Structural Refinement Types

David Binder
University of Tübingen
Germany
david.binder@uni-tuebingen.de

Ingo Skupin
University of Tübingen
Germany
skupin@informatik.uni-tuebingen.de

David Läwen
University of Tübingen
Germany
david.laewen@gmx.de

Klaus Ostermann
University of Tübingen
Germany
klaus.ostermann@uni-tuebingen.de

Abstract

Static types are a great form of lightweight static analysis. But sometimes a type like **List** is too coarse – we would also like to work with its *refinements* like non-empty lists, or lists containing exactly 42 elements. Dependent types allow for this, but they impose a heavy proof burden on the programmer. We want the checking and inference of refinements to be fully automatic.

In this article we present a simple refinement type system and inference algorithm which uses only variants of familiar concepts from constraint-based type inference. Concretely, we build on the algebraic subtyping approach and extend it with typing rules which combine properties of nominal and structural type systems in a novel way. Despite the simplicity of our approach, the resulting type system is very expressive and allows to specify and infer non-trivial properties of programs.

CCS Concepts: • **Theory of computation → Type theory**.

Keywords: Structural Types, Nominal Types, Refinement Types, Algebraic Subtyping

ACM Reference Format:
David Binder, Ingo Skupin, David Läwen, and Klaus Ostermann. 2022. Structural Refinement Types. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '22), September 11, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3546196.3550163

TyDe '22, September 11, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9439-0/22/09…$15.00
https://doi.org/10.1145/3546196.3550163

1 Introduction

Consider the following simple predecessor function on Peano numbers:

$$\textbf{def}\ \texttt{pred} \coloneqq \lambda x.\textbf{case}\ x\ \textbf{of}\ \{S(n) \Rightarrow n\}$$

According to most type systems and implementations, this function is partial. How programming languages handle this partiality may differ; some implementations will crash at runtime while others generate a warning or an error at compile-time. But there is another alternative: This function is a perfectly well-behaved total function on the domain of *non-zero* Peano numbers. Expressing and inferring such *refinements* of data types is the subject of this article.

We are certainly not the first to have made this observation. More fine-grained types of this sort are commonly known as *refinement types* [6]. In distinction to dependent types, which can express almost arbitrary subsets of types, refinement types are much more lightweight. A heavy emphasis on automation makes it so that little to no additional annotations are required by the programmer to benefit from their expressiveness.

As soon as we introduce refinement types into a type system, some notion of subtyping comes into play: If a function expects a number as an argument, it is clearly also valid to pass a subtype such as a non-zero number. So we need subtyping, but the design of a global type inference algorithm with nice properties such as principal types for a system with both subtyping and parametric polymorphism is known to be a hairy problem. Luckily for us, we don't have to solve this problem ourselves, since Dolan and Mycroft [3, 4] as well as Parreaux [14] have already done so for us. The *algebraic subtyping* approach they have developed can be used as a blueprint to design a type system which fulfills the desired properties, similar to how the basic Hindley-Milner algorithm can be used as a blueprint for further type system extensions.

While experimenting with an implementation of the algebraic subtyping type inference algorithm, we made the following discovery, which we present as our central contribution in this paper: We implemented both ordinary nominal algebraic data types, as well as purely structural data types in the form of polymorphic variants [7]. We then realized

that if we combine the typing rules for polymorphic variants with the typing rules for nominal data types, we get a form of refinement types for free! Before we go into further detail, let us briefly review the spectrum of nominal and structural type systems, one of the principal ways in which type systems can be classified.

A nominal type system can be recognized by the presence of a sublanguage for specifying types and the relationships between them. In inheritance-based, object-oriented languages, for example, this sublanguage is used to specify classes and their subtyping relationships, while in data-oriented languages, it is used to specify a wide variety of algebraic data types, such as enums and structs. By contrast, structural type systems consider the *structure* of types to reason about type compatibility. In systems for structural records and polymorphic variants [7, 8], for example, the types reflect which fields are present in records, or which variants may be present in a sum type. Nominal subtyping can be seen as a subset of structural subtyping in which a nominal subtyping declaration must be present in addition to a structural subtyping relation, which on one hand facilitates unplanned extensibility but on the other hand leads to undesirable subtype relationships [13].

In practice, a lot of systems combine nominal and structural aspects within the same system. To illustrate this, consider the following examples:

$$\{\ a = 2, b = \texttt{True}\ \} : \{\ a : \mathbb{N}, b : \mathbb{B}\ \}$$
$$\texttt{`IoError}(5) : \langle\ \texttt{`IoError}(\mathbb{N})\ \rangle$$

In the first line, both fields a and b are reflected in the structural record type. The fields themselves, on the other hand, are typed using nominal types $\mathbb{N}$ and $\mathbb{B}$. The second example illustrates the use of polymorphic variants (or structural sum types), using syntax inspired by OCaml. The constructor `` `IoError `` is not part of some algebraic data type declared elsewhere in the program or the standard library. It is introduced in this term, so its name and argument type is reflected in the inferred type $\langle\ \texttt{`IoError}(\mathbb{N})\ \rangle$.

The combination of such structural variants and records with equi-recursive types is surprisingly expressive. Equi-recursive types have the form $\mu\alpha.\tau$ and are for typechecking purposes indistinguishable from their unfolding $\tau[\mu\alpha.\tau/\alpha]$[1]. As an example for this expressivity, consider the following function:

$$\textbf{def}\ \texttt{isEven} \coloneqq \lambda x.\textbf{case}\ x\ \textbf{of}\ \{\texttt{`Z} \Rightarrow \texttt{True};\ \texttt{`S}(x) \Rightarrow \texttt{not}(\texttt{isEven}(x))\}$$

In a system which can infer equi-recursive types, this function has the inferred type $\mu\alpha.\langle\ \texttt{`Z}, \texttt{`S}(\alpha)\ \rangle \to \mathbb{B}$. This type correctly reflects both the constructors matched against and the recursive nature of the function. But while this type is hardly more informative than the type $\mathbb{N} \to \mathbb{B}$, the following example shows how structural types in combination with equi-recursive types can give more information than the corresponding nominal types can. Consider the following function which doubles its argument:

$$\textbf{def}\ \texttt{double} \coloneqq \lambda x.\textbf{case}\ x\ \textbf{of}\ \{\texttt{`Z} \Rightarrow \texttt{`Z};\ \texttt{`S}(x) \Rightarrow \texttt{`S}(\texttt{`S}(\texttt{double}(x)))\}$$

The inferred type of this function is $\mu\alpha.\langle\ \texttt{`Z} \mid \texttt{`S}(\alpha)\ \rangle \to \mu\beta.\langle\ \texttt{`Z} \mid \texttt{`S}(\texttt{`S}(\beta))\ \rangle$. This type reveals that the function only returns even numbers!

But, alas, not everything is fine with these structural types. Consider the predecessor function which we used as our first example, but this time with a polymorphic variant:

$$\textbf{def}\ \texttt{pred} \coloneqq \lambda x.\textbf{case}\ x\ \textbf{of}\ \{\texttt{`S}(n) \Rightarrow n\}$$

Its inferred type scheme is the rather surprisingly polymorphic $\forall\alpha.\ \langle\ \texttt{`S}(\alpha)\ \rangle \to \alpha$. This type states that the predecessor function can be applied to any typeable term as long as it is wrapped in the successor constructor, even such nonsensical terms as `` `S(True) ``. Under the typing rules of a structural system this is correct and doesn't violate the soundness of the type theory. But it is clearly confusing for a programmer who expects that the application of the successor constructor to a boolean should result in a type error.

The main idea presented in this paper is that if we combine the ordinary typing rules for data types with the typing rules for polymorphic variants, we obtain an interesting refinement type system. We call these types *structural refinement types*. In this system, structural types are used as the refinements of nominal types which are declared by the user. For instance, given the nominal type $\mathbb{N}$ of Peano numbers, we can use various structural refinement types, such as

$$\text{all natural numbers} : \mu\alpha.\langle\ \mathbb{N} \mid Z, S(\alpha)\ \rangle$$
$$\text{non-zero numbers} : \langle\ \mathbb{N} \mid S(\mu\alpha.\langle\ \mathbb{N} \mid Z, S(\alpha)\ \rangle)\ \rangle$$
$$\text{even numbers} : \mu\alpha.\langle\ \mathbb{N} \mid Z, S(\langle\ \mathbb{N} \mid S(a)\ \rangle)\ \rangle$$

With these refinement types, the predecessor function can be typed with $\langle\ \mathbb{N} \mid S(\mu\alpha.\langle\ \mathbb{N} \mid Z, S(\alpha)\ \rangle)\ \rangle \to \mu\alpha.\langle\ \mathbb{N} \mid Z, S(\alpha)\ \rangle$[2]. That is, the function expects a non-zero number as an argument and returns some natural number.

Existing approaches to refinement types support extensive automation, but they make various trade-offs. In the original system of Freeman and Pfenning [6], the programmer was required to manually specify all refinements they want to use. We only require the user to specify the original data type, but not its refinements, which are discovered automatically. With liquid refinement types [18, 19], an external solver is

[1] This distinguishes equi-recursive types from iso-recursive types, where the isomorphism between $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$ has to be made explicit using folding and unfolding functions.

[2] There are many ways in which the presentation of these refinement types can be improved when shown to the user. We use a presentation which is close to the form used in the formalization.

used to solve the proof obligations for the programmer. By contrast, we use only familiar techniques from constraint-based type inference. The system of Jones and Ramsay [12] is very similar in spirit to our system, but the refinements they allow are less expressive. They support refinements which can be expressed as the removal of constructors from the definition of a type, but require these removals to be hereditary. As such, they cannot express the type of non-empty lists, which requires the removal of the `Nil` constructor only at the top level. Our system lifts this limitation, supporting all refinements that can be expressed as a regular sublanguage of the original type.

The rest of this paper is structured as follows:

- In Section 2 we present our central ideas using a simplified version of our type system.
- In Section 3 we present the full declarative type system with user defined and parameterized data types.
- In Section 4 we present the type inference algorithm, and the algorithms used to simplify types. The type inference algorithm uses a variant of the biunification algorithm introduced by Dolan and Mycroft [3, 4] with some modifications inspired by Parreaux [14]. Type simplification is achieved by encoding types in finite automata and using familiar simplification techniques from automata theory.
- We discuss related work in Section 5, future work in Section 6 and conclude in Section 7.

2 The Main Idea

Our main idea is to build upon the algebraic subtyping approach [3, 4, 14] and to extend it with types which combine the typing rules for nominal and structural types. In Section 3 and Section 4 we will present these rules with all the gory details. Since we show how to implement structural refinement types for arbitrary user-defined data types, the resulting rules are quite complex. In order to make them more palatable, we specialize the rules in this section to two examples: Peano numbers and lists. In Section 2.1 we show how to work with refinements of Peano numbers, since they are the simplest example involving recursive types. We show how to deal with refinements of lists in Section 2.2. Parameterized types like lists pose an interesting design question. Should we compute refinements of the spine of the list and its elements separately, or together? We chose to refine them separately, and motivate that choice in that subsection.

2.1 Peano Numbers

Natural numbers $\mathbb{N}$ can be represented by a data type with two constructors: zero (Z) and successor (S). The typing rules for the constructors and the pattern match are familiar:

$$\frac{}{\Gamma \vdash Z : \mathbb{N}}\ \mathrm{Z}_{Nominal} \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash S(e) : \mathbb{N}}\ \mathrm{S}_{Nominal}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \qquad \Gamma \vdash e_Z : \tau \qquad \Gamma, x : \mathbb{N} \vdash e_S : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{Z \Rightarrow e_Z, S(x) \Rightarrow e_S\} : \tau}\ \mathrm{Case}^{\mathbb{N}}_{Nominal}$$

Nothing about these rules is surprising. But note that even in these very simple rules one of the essential characteristics of all static analyses is already present. The types that we use are a conservative approximation, as the single type $\mathbb{N}$ is used for all natural numbers.

In contrast to this, let us consider what would happen if we used polymorphic variants [7, 8] for natural numbers. The typing rule for zero and the successor would then be the following:

$$\frac{}{\Gamma \vdash \texttt{`Z} : \langle\ \texttt{`Z}\ \rangle}\ \mathrm{Z}_{Structural} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \texttt{`S}(e) : \langle\ \texttt{`S}(\sigma)\ \rangle}\ \mathrm{S}_{Structural}$$

If we compare them to the previous rules, we can make the following observations: The inferred type $\langle\ \texttt{`Z}\ \rangle$ for zero is much more informative, it is even specific enough to deduce the only possible inhabitant, namely the number 0! And the same holds true for the rule for the successor; from the type in the conclusion of the rule we can deduce that the outermost constructor must be `` `S ``. But the rule $\mathrm{S}_{Structural}$ is also somewhat disappointing: It does not impose any restriction on the type of the term that `` `S `` is applied to. This is why a term like `` `S(True) `` is typeable with this rule.

Similar to how the typing rules for constructors reflect the constructor used to construct the term in the type, the rule for pattern matches reflects the constructors which are matched against. For example, a pattern match which only matches against the constructor `` `S `` is typed as follows:

$$\frac{\Gamma \vdash e : \langle\ \texttt{`S}(\tau)\ \rangle \qquad \Gamma, x : \tau \vdash e_S : \rho}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\texttt{`S}(x) \Rightarrow e_S\} : \rho}\ \mathrm{Case}^{S}_{Structural}$$

In this rule we don't require e to be a natural number. We only place the minimal requirement on the type e, namely to be wrapped with the constructor `` `S ``.

So what is the result when we combine the nominal and structural rules above? In our system, the typing rules for zero and the successor look as follows:

$$\frac{}{\Gamma \vdash Z : \langle\ \mathbb{N} \mid Z\ \rangle}\ \mathrm{Z}_{Refinement}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma <: \mu\alpha.\langle\ \mathbb{N} \mid Z, S(\alpha)\ \rangle}{\Gamma \vdash S(e) : \langle\ \mathbb{N} \mid S(\sigma)\ \rangle}\ \mathrm{S}_{Refinement}$$

In the typing rule $\mathrm{S}_{Refinement}$ we have combined the properties of both systems. In the conclusion of the rule, we can still deduce from the type $\langle\ \mathbb{N} \mid S(\sigma)\ \rangle$ which constructor was used to build the term. And in the second premise we still put a constraint on the permissible arguments of S, namely that the argument to S must be some subtype of the Peano natural numbers, i.e. a refinement of $\mathbb{N}$. The situation is similar in the case of pattern matches:

$$\frac{\langle\, \mathbb{N} \mid \emptyset \,\rangle <: \tau <: \mu\alpha.\langle\, \mathbb{N} \mid Z, S(\alpha) \,\rangle \quad \Gamma \vdash e : \langle\, \mathbb{N} \mid S(\tau) \,\rangle \quad \Gamma, x : \tau \vdash e_S : \rho}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{S(x) \Rightarrow e_S\} : \rho}\ \textsc{Case}^{S}_{\textit{Refinement}}$$

We require the term e to be of type $\langle\, \mathbb{N} \mid S(\tau) \,\rangle$, since we only match against the successor constructor. But we also require the argument of S to be *some* natural number, which we express here with a lower and an upper bound on τ. We might learn more about the requirements that τ must satisfy in the body e_S, for example that τ must be non-zero, too.

2.2 Lists

In the previous section we used the example of natural numbers, whose definition in the formal syntax of Section 3 will look like this:

$$\textbf{data}\ \text{Nat} : (;) \to * \{\ Z() : \text{Nat}(;),\ S(\textbf{rec}@(;)) : \text{Nat}(;)\ \}$$

After the name of the type constructor Nat we have to specify its kind. Since natural numbers are not parameterized, the kind is simply $(;) \to *$, which is isomorphic to the kind $*$ of inhabited types. The recursive occurrence in the argument of the constructor S is written using the special **rec** symbol, which has to be applied to zero type arguments, using the syntax $\textbf{rec}@(;)$. This syntax makes more sense once we see the definition of a parameterized type. The type of lists of some element type α is defined as follows:

$$\begin{aligned}&\textbf{data}\ \text{List} : (*;) \to * \{\\ &\quad \forall\alpha.\texttt{Nil}() : \textbf{List}(\alpha;),\\ &\quad \forall\alpha.\texttt{Cons}(\alpha, \textbf{rec}@(\alpha;)) : \textbf{List}(\alpha;)\ \}\end{aligned}$$

This definition introduces the type constructor **List** of kind $(*;) \to *$. In the algebraic subtyping system we have to keep covariant and contravariant parameters of a type strictly separate. The element type of a list is a covariant argument, and covariant arguments are written to the left of the semicolon in the kind of **List**.

There is one central design question concerning parameterized types like lists. The problem can be illustrated with the list `Cons(True, Cons(False, Nil))`. What should be the type of this list? As we see it, there are two possible choices:

1. We infer the refinement as if the programmer had written a data type declaration of lists of booleans. We call this the *monomorphising* approach. With this approach, we infer the type of a two-element list which contains the term `True` at the first position and the term `False` at the second position.
2. We infer separate refinements for the spine of the list and the elements of the list. Using this approach, we infer the type of a two element list (that is, a refinement on lists) whose elements are among the set containing both `True` and `False` (that is, a refinement of booleans). On a technical level, this corresponds to introducing a single unification variable for all of the elements of the list.

The types that are inferred using the first approach are more precise, but this precision comes at a cost. Type inference has to keep track of a lot more information, making it much harder to scale to realistic programs. Secondly, the inferred types are much harder to decipher for the user, and the approach is overall less modular. For these reasons we have decided to specify and implement the second approach.

An example for functions on lists can be found in Figure 1. Note that all type annotations in Figure 1 are not necessary, but they are checked can be used as documentation. The `mapNonEmpty` function operates on the type `NonEmpty` of non-empty lists, which is a subtype of the type `FullList` of list of arbitrary length. This illustrates how we can keep reasoning about the *shape* of the list separate from reasoning about the lists *contents*: `mapNonEmpty` keeps track of the shape and allows us to recover the shape of the input in the output, while at the same time remaining parametric over the types of the elements of the lists involved. This allows us to use its output as an input to the `max` function. At the same time, we would still be able to use its output in any parametric function on lists like a `listLength` function since `NonEmpty` is a subtype of `FullList`.

2.3 The Expressivity of Structural Refinement Types

How expressive are the structural refinement types that we present here? In their paper, Freeman and Pfenning [6] already made the observation that they can only express refinements that correspond to a regular sublanguage of the original type, and we have the same restriction in our system. For example, it is possible to express the refinement type of even natural numbers, but it is not possible to express the refinement type of natural numbers that are prime.

This restriction of the expressive power is essential in several different respects. The favorable closure properties of regular languages are crucial when we compute and simplify unions and intersections of refinement types. The close correspondence between types and finite automata, which explains the restriction to regular sublanguages, is also the basis of the simplification algorithms described by Dolan [3], which we modify for our purposes in Section 4.4.

3 Formalization

In this section we describe the declarative type system; the type inference algorithm will then be described in Section 4. We introduce terms in Section 3.1, kinds in Section 3.2 and most types in Section 3.3. We introduce the syntax and rules of structural refinement types in Section 3.4. We use the notation $\overline{e}$ to denote a (possibly empty) list of elements e, following the conventions of Igarashi et al. [10].

```
data List : (∗;) → ∗ {
  ∀α.Nil() : List(α;),
  ∀α.Cons(α, rec@(α;)) : List(α;) }

type FullList : (∗;) → ∗ :=
  μδ.⟨ List(γ;) | Nil, Cons(γ, δ) ⟩

map : (α → β) →
    FullList@(α;) → FullList@(β;)
map := λf xs. case xs of {
    Nil ⇒ Nil;
    Cons(y, ys) ⇒ Cons(f y, map f ys)}

type NonEmpty : (∗;) → ∗ :=
    ⟨ List(α;) | Cons(α, FullList@(α;)) ⟩

mapNonEmpty : (α → β) →
    NonEmpty@(α;) → NonEmpty@(β;)
mapNonEmpty := λf xs. case xs of {
    Cons(y, ys) ⇒ Cons(f y, map f ys)}

max : NonEmpty@(ℕ;) → ℕ
max := λns. case ns of {
    Cons(m, ms) ⇒ case ms of {
        Nil ⇒ m
        Cons(o, os) ⇒ case (max Cons(o, os)) ≤ m of {
            True ⇒ m
            False ⇒ max Cons(o, os)}}}
maxLength : NonEmpty@(String;) → ℕ
maxLength := λss. max (mapNonEmpty length ss)
```

Figure 1. Mapping on non-empty lists. The type annotations are for documentation purposes; principal types can always be inferred.

3.1 Terms

The term system, presented in Figure 2, is entirely standard for a language with functions and data types. Terms e can be variables x, which are taken from the set of term variables Var. The two syntactic forms for functions are lambda abstractions $\lambda x.e$ and function applications $e\ e$. Elements of an algebraic data type can be constructed with constructors applied to arguments $C(\overline{e})$, where the names of constructors are from the set CtorName. In order to deconstruct an element of an algebraic data type one uses a pattern match **case** e **of** $\{\overline{C(\overline{x}) \Rightarrow e}\}$.

We do not specify any operational semantics for the terms of Figure 2. The type theory we present is equally suitable for both strict and non-strict evaluation orders.

$$x, y \in \textsc{Var} \quad C \in \textsc{CtorName}$$

$$\begin{array}{rcl} e, e_i & ::= & x \mid \lambda x.e \mid e\ e \\ & \mid & C(\overline{e}) \mid \textbf{case}\ e\ \textbf{of}\ \{\overline{C(\overline{x}) \Rightarrow e}\} \end{array}$$

Figure 2. The syntax of terms.

3.2 Kinds

We formalize a system with higher kinds whose syntax is given in Figure 3.

The kind $*$ classifies inhabited types. Higher kinds have the form $(\overline{\kappa}; \overline{\kappa'}) \to *$. They have two lists of arguments $\overline{\kappa}$ and $\overline{\kappa'}$ and always return the kind $*$. The two argument lists are needed to keep track of variance; the first list stands for covariant arguments and the second list stands for contravariant arguments. As an example, the list type constructor has kind $(*;) \to *$, taking one covariant argument of kind $*$ and no contravariant arguments.

$$\kappa \quad ::= \quad * \quad | \quad (\overline{\kappa}; \overline{\kappa}) \to * \qquad \textit{Kinds}$$

Figure 3. The syntax of kinds.

3.3 Types

We will first discuss the types of the core system, which are given in Figure 4. Type variables are taken from a set TyVar and occur as both skolem variables α and unification variables $\alpha^?$ once we consider type inference in Section 4. The function type $\sigma \to \tau$ is the only built-in concrete type, all other concrete types are specified by the programmer. We postpone the discussion of user-specified types until Section 3.4. The application of a higher-kinded type constructor to type arguments is written $\tau@(\overline{\sigma}; \overline{\rho})$; in this example the type constructor τ is applied to the covariant arguments $\overline{\sigma}$ and the contravariant arguments $\overline{\rho}$. Since we are in a subtyping system we also have union and intersection types $\sigma \sqcup \tau$ and $\sigma \sqcap \tau$, as well as top and bottom types $\top$ and $\bot$. Equirecursive types are written $\mu\alpha.\tau$ and are indistinguishable from their unfolding $\tau[\mu\alpha.\tau/\alpha]$.

Since we have type constructors with higher kinds in our system, we have to specify rules for checking the kinds of types. These rules are given in Figure 5. The judgement $\Delta \vdash \tau : \kappa$ states that the type τ has kind κ in a kind environment Δ which assigns kinds to type variables. The syntax of kind environments is specified in Figure 4.

Note that in the kinding rules in Figure 5 the function type can only be applied to types of the inhabited kind $*$, but all other constructions like unions, intersections, the top and bottom type and recursive types can occur at arbitrary kinds. This is an essential prerequisite for the correct treatment of structural refinement types: For example, we need to be able

$$\alpha, \beta, \gamma, \delta, \rho \in \text{TyVar}$$

$$\begin{array}{rcll}
\sigma, \tau & ::= & \alpha & \textit{Skolem variable} \\
& | & \alpha^? & \textit{Unification variable} \\
& | & \tau \to \tau & \textit{Function type} \\
& | & \tau @ (\overline{\tau}; \overline{\tau}) & \textit{Type Application} \\
& | & \tau \sqcup \tau \quad | \quad \tau \sqcap \tau & \textit{Union, Intersection} \\
& | & \top \quad | \quad \bot & \textit{Top and Bottom} \\
& | & \mu\alpha.\tau & \textit{Recursive type} \\
\Delta & ::= & \epsilon \quad | \quad \alpha : \kappa, \Delta & \textit{Kind environments} \\
\Sigma & ::= & \epsilon \mid \sigma <: \tau, \Sigma \mid \triangleright (\sigma <: \tau), \Sigma & \textit{Hypotheses context}
\end{array}$$

Figure 4. Syntax of types for the core system.

Kinding rules: $\Delta \vdash \tau : \kappa$

$$\frac{\Delta(\alpha) : \kappa}{\Delta \vdash \alpha : \kappa}\ \text{K-Var} \qquad \frac{\Delta, \alpha : \kappa \vdash \tau : \kappa}{\Delta \vdash \mu\alpha.\tau : \kappa}\ \text{K-Mu}$$

$$\frac{}{\Delta \vdash \top : \kappa}\ \text{K-Top} \qquad \frac{}{\Delta \vdash \bot : \kappa}\ \text{K-Bot}$$

$$\frac{\Delta \vdash \sigma : \kappa \qquad \Delta \vdash \tau : \kappa}{\Delta \vdash \sigma \sqcup \tau : \kappa}\ \text{K-Union}$$

$$\frac{\Delta \vdash \sigma : \kappa \qquad \Delta \vdash \tau : \kappa}{\Delta \vdash \sigma \sqcap \tau : \kappa}\ \text{K-Inter}$$

$$\frac{\Delta \vdash \sigma : * \qquad \Delta \vdash \tau : *}{\Delta \vdash \sigma \to \tau : *}\ \text{K-Fun}$$

$$\frac{\Delta \vdash \sigma : (\overline{\kappa}; \overline{\kappa'}) \to * \qquad \overline{\Delta \vdash \tau : \kappa} \quad \overline{\Delta \vdash \tau' : \kappa'}}{\Delta \vdash \sigma @ (\overline{\tau}; \overline{\tau'}) : *}\ \text{K-TyApp}$$

$$\frac{\Delta \vdash \tau : (;) \to *}{\Delta \vdash \tau : *}\ \text{K-Star}$$

Figure 5. Kinding rules.

to talk about the union of the type constructors of lists and non-empty lists. The K-Star rule witnesses the fact that a higher order kind with no parameters is the same as kind $*$.

The declarative typing rules for the core system are presented in Figure 6. The presentation of these rules is adapted from that in [14]. The judgement $\Gamma \vdash e : \tau$ states that the expression e has type τ in the variable context Γ which assigns types to term variables. Note that in this form the rule T-Sub is not syntax directed, since it allows to change the type of an expression at an arbitrary point in a typing derivation.

The last set of rules concern the formalization of the subtyping lattice, and are presented in Figure 7. The judgement $\Sigma \vdash \sigma <: \tau$ says that σ is a subtype of τ under the assumption that the hypotheses in Σ hold. There are two different kinds of hypotheses, guarded hypotheses $\triangleright(\sigma <: \tau)$ and unguarded hypotheses $\sigma <: \tau$. Unguarded hypotheses can be used directly using the rule S-Hyp, but guarded hypotheses must first be unlocked with the $\triangleleft$ operation before they can be used. Hypotheses are unguarded every time we pass

Typing rules: $\Gamma \vdash e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{T-App}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ \text{T-Lam} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}\ \text{T-Sub}$$

Figure 6. Declarative typing rules

a type constructor in the rule S-Fun. The additional complication of hypotheses contexts is necessary for the correct treatment of recursive types. Without them, we would not be able to determine that $\mu\alpha.\textbf{Bool} \to \alpha$ is a subtype of $\mu\alpha.\textbf{Bool} \to \textbf{Bool} \to \alpha$, even though they have the same infinite unfolding. This example is well-explained in [14].

Subtyping rules: $\Sigma \vdash \sigma <: \tau$

$$\frac{}{\vdash \tau <: \tau}\ \text{S-Refl} \qquad \frac{H \in \Sigma}{\Sigma \vdash H}\ \text{S-Hyp} \qquad \frac{\Sigma, \triangleright H \vdash H}{\Sigma \vdash H}\ \text{S-Assum}$$

$$\frac{\vdash H}{\Sigma \vdash H}\ \text{S-Weaken} \qquad \frac{}{\vdash \tau <: \top}\ \text{S-Top} \qquad \frac{}{\vdash \bot <: \tau}\ \text{S-Bot}$$

$$\frac{\Sigma \vdash \tau <: \tau' \qquad \Sigma \vdash \tau' <: \tau''}{\Sigma \vdash \tau <: \tau''}\ \text{S-Trans}$$

$$\frac{\triangleleft\Sigma \vdash \sigma' <: \sigma \qquad \triangleleft\Sigma \vdash \tau <: \tau'}{\Sigma \vdash \sigma \to \tau <: \sigma' \to \tau'}\ \text{S-Fun}$$

$$\frac{\Sigma \vdash \tau <: \tau' \qquad \Sigma \vdash \overline{\sigma <: \sigma'} \qquad \Sigma \vdash \overline{\rho' <: \rho}}{\Sigma \vdash \tau @ (\overline{\sigma}; \overline{\rho}) <: \tau' @ (\overline{\sigma'}; \overline{\rho'})}\ \text{S-App}$$

$$\frac{\forall j \exists i : \Sigma \vdash \tau_i <: \sigma_j}{\Sigma \vdash \sqcap_i \tau_i <: \sqcap_j \sigma_j}\ \text{S-Meet}$$

$$\frac{\forall i \exists j : \Sigma \vdash \tau_i <: \sigma_j}{\Sigma \vdash \sqcup_i \tau_i <: \sqcup_j \sigma_j}\ \text{S-Join}$$

$$\frac{}{\Sigma \vdash \tau[\mu\rho.\tau/\rho] <: \mu\rho.\tau}\ \text{S-}\mu\text{-R}$$

$$\frac{}{\Sigma \vdash \mu\rho.\tau <: \tau[\mu\rho.\tau/\rho]}\ \text{S-}\mu\text{-L}$$

where

$$\triangleleft\overline{\Sigma} = \overline{\triangleleft\Sigma} \qquad \triangleleft(\triangleright H) = H \qquad \triangleleft(\tau <: \sigma) = \tau <: \sigma$$

Figure 7. Declarative subtyping rules

3.4 Structural Refinement Types

We now extend the system with user defined data types, which are specified in Figure 8. A program contains a set of data type declarations of the form $\textbf{data}\ N : (\overline{\kappa}; \overline{\kappa'}) \to * \{\ \overline{ct}\ \}$, which introduce a new algebraic data type N with kind $(\overline{\kappa}; \overline{\kappa'}) \to *$ whose name is taken from the set TyName, together with its constructors $\overline{ct}$. For example, the type of

lists is defined with the following declaration:

$$\begin{aligned}&\textbf{data}\ \mathrm{List} : (*;) \to * \ \{\\ &\quad \forall\alpha.\mathtt{Nil}() : \mathrm{List}(\alpha;),\\ &\quad \forall\alpha.\mathtt{Cons}(\alpha, \textbf{rec}@(\alpha;)) : \mathrm{List}(\alpha;)\ \}\end{aligned}$$

The types of the arguments inside data type declarations are restricted to *non-polar types* v, which are the subset of all types that do not contain any lattice constructors $\sqcap$, $\sqcup$, $\top$ or $\bot$. We also require all occurrences of **rec** to be in positive positions.

The additional kinding, typing and subtyping rules are presented in Figure 9.

$$N, M \in \textsc{TyName}$$

d	::=	$\textbf{data}\ N :\ (\overline{\kappa};\overline{\kappa}) \to * \ \{\ \overline{ct}\ \}$	*Type declaration*
ct	::=	$\forall\overline{\alpha},\overline{\alpha'}.\, C(\overline{v}) : N(\overline{\alpha};\overline{\alpha'})$	*Constructor declaration*
v	::=	α	*Skolem variable*
	|	$v \to v$	*Function type*
	|	$v@(\overline{v};\overline{v})$	*Type application*
	|	$\mu\alpha.v$	*Recursive type*
	|	$\langle\ N(\overline{\alpha};\overline{\alpha})\ \mid \overline{C(\overline{v})}\ \rangle$	*Refinement type*
	|	**rec**	*Recursive occurrence*
σ, τ	::=	$\ldots$	(From Figure 4)
	|	$\langle\ N(\overline{\alpha};\overline{\alpha})\ \mid \overline{C(\overline{\tau})}\ \rangle$	*Refinement type*

Figure 8. The syntax of refinement types and data declarations.

Kinding rules: $\Delta \vdash \tau : \kappa$

$$\frac{\textbf{data}\ N : \kappa\{\overline{ct}\}\ \mathrm{OK} \qquad \overline{\forall\overline{\beta},\overline{\beta'}.\, C(\overline{\tau}[\beta/\alpha,\beta'/\alpha']) : N(\overline{\beta};\overline{\beta'}) \subseteq \overline{ct}}}{\Delta \vdash \langle\ N(\overline{\alpha};\overline{\alpha'})\ \mid \overline{C(\overline{\tau})}\ \rangle : \kappa}\ \textsc{K-Refine}$$

Well-formedness rules: d OK

$$\frac{\overline{\overline{\alpha:\kappa}, \overline{\alpha':\kappa'} \vdash \overline{\tau} : *}}{\textbf{data}\ N : (\overline{\kappa};\overline{\kappa'}) \to * \ \left\{\overline{\forall\overline{\alpha},\overline{\alpha'}.\, C(\overline{\tau}) : N(\overline{\alpha};\overline{\alpha'})}\right\}\ \mathrm{OK}}\ \textsc{K-Data}$$

Typing rules: $\Gamma \vdash e : \tau$

$$\frac{\forall\, i : \Gamma \vdash e_i : \sigma_i[\overline{\alpha'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}] \qquad \forall\overline{\alpha},\overline{\beta}.\, C(\overline{\sigma_i}) : N(\overline{\alpha};\overline{\beta}) \in \mathit{Ctors}}{\Gamma \vdash C(\overline{e_i}) : \langle\ N(\overline{\alpha};\overline{\beta})\ \mid C(\overline{\sigma_i})\ \rangle@(\overline{\alpha'};\overline{\beta'})}\ \textsc{T-Ctor}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \langle\ N(\overline{\alpha};\overline{\beta})\ \mid \overline{C_i(\overline{\sigma_{ij}})}\ \rangle@(\overline{\alpha'};\overline{\beta'}) \\ \forall\, i :\ \Gamma, \overline{x_{ij} : \sigma_{ij}[\overline{\alpha'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}]} \vdash e_i : \tau[\overline{\alpha'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}] \\ \left\{\overline{\forall\overline{\alpha},\overline{\beta}.\, C_i(\overline{\sigma_{ij}}) : N(\overline{\alpha};\overline{\beta})}\right\} = \mathit{Ctors}(N)\end{array}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\overline{C_i(\overline{x_{ij}}) \Rightarrow e_i}\} : \tau}\ \textsc{T-Case}$$

Subtyping rules: $\Sigma \vdash \sigma <: \tau$

$$\frac{\triangleleft\Sigma \vdash \overline{s <: t}}{\Sigma \vdash \langle\ N(\overline{\alpha};\overline{\beta})\ \mid \overline{C(\overline{s})}\ \rangle <: \langle\ N(\overline{\alpha};\overline{\beta})\ \mid \overline{C(\overline{t})}, \varphi\ \rangle}\ \textsc{S-Refine}$$

Figure 9. Extension with structural refinement types.

4 Type Inference

In this section we present the algorithm which we use to infer structural refinement types. The overall architecture of the type inference algorithm follows Pottier and Rémy [16]. In their approach, one introduces a language of constraints (Section 4.1) and then implements separate phases for first generating constraints (Section 4.2) and then solving them (Section 4.3). The resulting types might be very verbose and can be compressed, which is why a type simplification pass is needed. We will present automata-based type simplification for structural refinement types in Section 4.4.

4.1 Constraints and Polar Types

At the heart of Hindley-Milner based type inference is the unification algorithm which solves a set of type equalities of the form $\sigma \sim \tau$. Starting from the initial set of generated constraints, the unification algorithm successively applies one of the following three operations, until no further constraints remain: (1) Remove equations of the form $\tau \sim \tau$. (2) Decompose constraints like $\sigma \to \tau \sim \sigma' \to \tau'$ into simpler constraints $\sigma \sim \sigma'$ and $\tau \sim \tau'$. (3) Solve equations between a unification variable and another type (like $\alpha^? \sim \tau$) by substituting τ for all occurrences of $\alpha^?$ in the remaining constraints, possibly after performing an occurs-check. The result of the unification algorithm is a type substitution which maps unification variables to types, such that all constraints are satisfied.

In the algebraic subtyping approach [3, 4, 14] this scheme is modified in order to work with type inequality constraints $\sigma <: \tau$ instead of type equality constraints $\sigma \sim \tau$. There are two important problems one encounters when trying to adapt Hindley-Milner based type inference to type inequality constraints. First, the presence of equi-recursive types (and the corresponding absence of an occurs-check) make special precautions necessary in order to guarantee the termination of the inference algorithm. This is solved by adding a cache of solved constraints, which we will discuss in Section 4.3. The second problem is that unions and intersections as well as top and bottom types can now occur in constraints. To see why this can be a problem, consider the two constraints

$\sigma_1 \sqcup \sigma_2 <: \tau$ and $\sigma <: \tau_1 \sqcup \tau_2$. The first constraint is equivalent to the conjunction of the two constraints $\sigma_1 <: \tau$ and $\sigma_2 <: \tau$, and can easily be handled by the constraint solver. The second constraint $\sigma <: \tau_1 \sqcup \tau_2$, on the other hand, is equivalent to a *disjunction* of constraints which cannot be easily handled. The solution to this problem, proposed by [3], is to exclude constraints of the second kind by introducing a notion of *polar types* which restrict where unions and intersections, top and bottom types can occur within types.

$$
\begin{array}{lcll}
p & ::= & + \mid - & \textit{Polarities} \\
\Pi & ::= & \epsilon \mid \alpha : p, \Pi & \textit{Polarity context} \\
q & ::= & \sigma^+ <: \tau^- & \textit{Constraints}
\end{array}
$$

$$\neg(+) = - \quad \neg(-) = + \quad \textit{(Polarity switch)}$$

Polarity check: $\Pi \vdash \tau : p$

$$\frac{\Pi(\alpha) = p}{\Pi \vdash \alpha : p}\ \text{P-Bound} \qquad \frac{\alpha \notin \Pi}{\Pi \vdash \alpha : p}\ \text{P-Free}$$

$$\frac{}{\Pi \vdash \top : -}\ \text{P-Top} \qquad \frac{}{\Pi \vdash \bot : +}\ \text{P-Bot}$$

$$\frac{\Pi \vdash \sigma : + \qquad \Pi \vdash \tau : +}{\Pi \vdash \sigma \sqcup \tau : +}\ \text{P-Union}$$

$$\frac{\Pi \vdash \sigma : - \qquad \Pi \vdash \tau : -}{\Pi \vdash \sigma \sqcap \tau : -}\ \text{P-Inter}$$

$$\frac{\Pi \vdash \sigma : \neg(p) \qquad \Pi \vdash \tau : p}{\Pi \vdash \sigma \to \tau : p}\ \text{P-Fun}$$

$$\frac{\Pi, \alpha : p \vdash \tau : p}{\Pi \vdash \mu\alpha.\tau : p}\ \text{P-Mu}$$

$$\frac{\Pi \vdash \tau : p \qquad \overline{\Pi \vdash \sigma : p} \qquad \overline{\Pi \vdash \rho : \neg(p)}}{\Pi \vdash \tau@(\overline{\sigma}; \overline{\rho}) : p}\ \text{P-TyApp}$$

$$\frac{\overline{\Pi, \overline{\alpha : p}, \overline{\beta : \neg(p)} \vdash \tau : p}}{\Pi \vdash \langle\ N(\overline{\alpha}; \overline{\beta}) \mid \overline{C(\overline{\tau})}\ \rangle : p}\ \text{P-Refine}$$

Figure 10. The restriction to polar types.

The sub-syntax of polar types is formalized in Figure 10. We distinguish two different polar types, positive types τ^+ and negative types τ^-. The polarity of a type is checked with the help of the judgement $\Pi \vdash \tau : p$, where a type τ is checked to have polarity p in the polarity context Π which assigns polarities to type variables. The only rules which require types to have concrete polarities are the rules P-Union, P-Inter, P-Top and P-Bot; this is the essential restriction of polar types. The rules P-Fun, P-TyApp and P-Refine implement the usual rules for type constructors which deal with co- and contravariance, and make use of the helper function $\neg(p)$ for switching the polarity of contravariant arguments. The rule P-Mu extends the context with the recursive variable; this is needed to guarantee that the unfolding of a polar recursive type is always a polar type with the same polarity. Type variables which occur in the context Π must have the polarity assigned to them in the context (rule P-Bound), whereas type variables not contained in Π can be given any polarity (rule P-Free). Without the latter rule, we would not be able to assign a polarity to the sensible type $\alpha \to \alpha$. Note that types which don't contain any unions and intersections or top and bottom types, can always be checked with both polarities.

Constraints q must always have the form $\sigma^+ <: \tau^-$. This restriction excludes the problematic example $\sigma <: \tau_1 \sqcup \tau_2$ we considered earlier in this section.

4.2 Constraint Generation

$$
\begin{aligned}
[\![\alpha]\!]^\bot_\tau &:= \alpha & [\![\alpha^?]\!]^\bot_\tau &:= \alpha^? \\
[\![\alpha]\!]^\top_\tau &:= \alpha & [\![\alpha^?]\!]^\top_\tau &:= \alpha^?
\end{aligned}
$$

$$
\begin{aligned}
[\![\nu \to \nu']\!]^\bot_\tau &:= [\![\nu]\!]^\top_\tau \to [\![\nu']\!]^\bot_\tau \\
[\![\nu \to \nu']\!]^\top_\tau &:= [\![\nu]\!]^\bot_\tau \to [\![\nu']\!]^\top_\tau \\
[\![\nu@(\overline{\nu'}; \overline{\nu''})]\!]^\bot_\tau &:= [\![\nu]\!]^\bot_\tau@(\overline{[\![\nu']\!]^\bot_\tau}; \overline{[\![\nu'']\!]^\top_\tau}) \\
[\![\nu@(\overline{\nu'}; \overline{\nu''})]\!]^\top_\tau &:= [\![\nu]\!]^\top_\tau@(\overline{[\![\nu']\!]^\top_\tau}; \overline{[\![\nu'']\!]^\bot_\tau}) \\
[\![\mu\alpha.\nu]\!]^\bot_\tau &:= \mu\alpha.[\![\nu]\!]^\bot_\tau \\
[\![\mu\alpha.\nu]\!]^\top_\tau &:= \mu\alpha.[\![\nu]\!]^\top_\tau \\
[\![\langle\ M(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{\nu})}\ \rangle]\!]^\bot_\tau &:= \langle\ M(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{[\![\nu]\!]^\bot_\tau})}\ \rangle \\
[\![\langle\ M(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{\nu})}\ \rangle]\!]^\top_\tau &:= \langle\ M(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{[\![\nu]\!]^\top_\tau})}\ \rangle
\end{aligned}
$$

And, given **data** $N : (\overline{\kappa}, \overline{\kappa'}) \to * \{\overline{C(\overline{\nu})}\}$:

$$
\begin{aligned}
[\![\mathbf{rec}]\!]^\bot_N &:= \langle\ N(\overline{\alpha}; \overline{\alpha'}) \mid \emptyset\ \rangle \\
[\![\mathbf{rec}]\!]^\top_N &:= \mu\rho.\langle\ N(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{[\![\nu]\!]^\top_\rho})}\ \rangle \\
[\![\mathbf{rec}]\!]^\bot_\rho &:= \rho \\
[\![\mathbf{rec}]\!]^\top_\rho &:= \rho
\end{aligned}
$$

Figure 11. Definitions of lower and upper bound translation

During constraint generation we take a term e (from Figure 2) and a typing context Γ, and return a positive type τ^+, together with a set of constraints Ξ. This is expressed in the corresponding judgement form $\Gamma \vdash e : \tau \rightsquigarrow \Xi$. The type τ that is generated contains unification variables, whose precise type will be determined in the subsequent constraint solving step. Constraint generation uses the *upper* and *lower bound translation* functions, which are given in Figure 11. These functions translate the types used inside a data type declaration to the *greatest* and respectively *least* types with respect respect to the surrounding declaration.

The constraint generation rules are given in Figure 12a, we will now discuss the rules in detail.

The type of a variable x is determined in rule G-Var by looking up the type assigned to x in the variable context Γ.

The terms for function abstraction and application are handled in rules G-Lam and G-App. In order to generate a type for a lambda abstraction $\lambda x.e$, a fresh unification variable $\beta^?$ is generated for the variable x. The body of the function e is then checked in the extended context $\Gamma, x : \beta^?$, resulting in the type τ, and the type $\beta^? \to \tau$ is returned. For function applications $e_1\ e_2$ the types σ_1 and σ_2 are inferred for the function and argument, respectively. We generate a fresh unification variable $\beta^?$ for the result of the function application, and add the constraint $\sigma_1 <: \sigma_2 \to \beta^?$.

Constructors are handled in rule G-Ctor. We first look up the signature of the constructor C in the program; assume that this results in $\forall \overline{\alpha}, \overline{\alpha'}.\, C(\overline{\sigma}) : N(\overline{\alpha}; \overline{\alpha'})$. We then generate fresh unification variables $\overline{\beta^?}$ and $\overline{\beta'^?}$ for the covariant and contravariant arguments of the type N to which the constructor C belongs. We return $\langle\ N(\overline{\alpha}; \overline{\alpha'}) \mid C(\overline{\sigma})\ \rangle @ (\overline{\beta^?}; \overline{\beta'^?})$, that is the refinement type of N containing only the constructor C, applied to the freshly generated unification variables. But we still have to generate constraints for the arguments of the constructor! For every term e in the argument list of the constructor, we infer a type τ. We then add a constraint between the inferred type τ and the type σ declared in the data type declaration. Before we do this, we have to replace the type variables α and α' in the type declaration with the fresh unification variables. We also have to replace the recursive occurrences of N in σ by the fully refined type N. This results in the constraint $\tau <: [\![\sigma]\!]^\top_N[\overline{\beta^?}/\overline{\alpha}, \overline{\beta'^?}/\overline{\alpha'}]$ which is ultimately emitted.

Pattern matches are handled in rule G-Case. First, we typecheck the term on which the pattern matching is done. The result of this typecheck is then constrained from above by the refinement type which contains all the constructors that appear in the patterns. Then we generate new unification variables $\beta^?_{ij}$ for all variables in the patterns and check the cases e_i of the pattern match using these new unification variables, resulting in types τ_i. The resulting type of the whole pattern match is a new unification variable $\gamma^?$ which is an upper bound of the types of all the cases, thus we add constraints $\tau_i <: \gamma$ for all i. Finally, we need to ensure that the pattern variables are only used in a way that is *compatible* with the refinement type of the pattern match. This is done via the G-Compat rule which constrains the unification variables $\beta^?_{ij}$ to be above the *lower bound translation* $[\![\cdot]\!]^\bot_N$ of the corresponding type in declaration of the current refinement type N, and below the *upper bound translation* $[\![\cdot]\!]^\top_N$ of the same type. The compatibility rule is also indexed by the name of the current refinement type to allow translating occurrences of **rec**. Adding constraints involving the lower bound translation is necessary to allow rejection of terms like $\lambda n.$ **case** n **of** $\{S(m) \Rightarrow$ **case** m **of** $\{\texttt{True} \Rightarrow \texttt{True}\}\}$. In this instance, the constraint $[\![\mathbb{N}]\!]^\bot_{\mathbb{N}} <: \beta^?$ for the unification variable $\beta^?$ of m which is generated in the pattern match for n is required to later reject this program when this constraint clashes with the constraint $\beta^? <: \langle$ **Bool** $\mid$ `True` $\rangle$. This leads to the constraint $\langle\ \mathbb{N} \mid \emptyset\ \rangle <: \langle$ **Bool** $\mid$ `True` $\rangle$ by transitivity.

4.3 Constraint Solving

The constraint solver takes a list of constraints qs as input, and produces a list of variable bounds B as output, or else fails if the set of constraints is not satisfiable. A variable bound B assigns to a unification variable $\alpha^?$ a list of lower and upper bounds. Constraint solving is defined as a state transition system on constraint solver states S. A constraint solver state S consists of a cache of already solved constraints, a list of constraints to be solved, and a list of computed variable bounds. If a constraint is encountered that cannot be solved, the constraint solver transitions into the special **Fail** state.

$$\begin{array}{lcll} B & \coloneqq & \overline{\sigma^+} <: \alpha^? <: \overline{\sigma^-} & \textit{Variable Bounds} \\ S & \coloneqq & \{Q\}; \overline{Q} \vdash \overline{B} \quad \mid \quad \textbf{Fail} & \textit{Constraint Solver State} \end{array}$$

The algorithm terminates when there are no constraints left to be processed. The cache is necessary to ensure termination and in order to avoid exponential runtime, as it is possible for the algorithm to add an already processed constraint back to its list of constraints. Initially, the cache is empty and the list of constraints to be processed is given by the output of the constraint generator. The initial configuration of the constraint solver for a list of constraints qs is therefore:

$$\emptyset; qs \vdash \text{empty}$$

As long as the list of constraints to be processed is not empty, one of the rules of Figure 12b will determine the next state of the constraint solver. These rules are applied repeatedly until the list of constraints is empty. In the end, this gives us a constraint solver state that correctly reflects all the upper and lower bounds implied by the set of initial constraints.

The rule CacheHit will remove the constraint if it is found in the cache, and has therefore already been processed.

The rules UpperBound and LowerBound solve *atomic* constraints whose left or the right hand side consist of a unification variable. If both σ_1 and σ_2 are unification variables, the rule UpperBound will fire, ensuring determinism[3]. We only discuss the rule UpperBound, since the rule LowerBound works similarly. The constraint $\alpha^? <: \sigma$ is solved by adding σ to the list of upper bounds of $\alpha^?$. But, as discussed in [14], we must also make sure that this new bound is consistent with the existing lower bounds of $\alpha^?$. We therefore add one new constraint between σ and each of the existing lower bounds to the constraints to be processed. These are the subconstraints that are implied by transitivity of the subtyping relation. Note that no occurs check is performed; it is

[3]This is analogous to the situation in Hindley-Milner, where to solve a constraint like $\alpha \sim \beta$, we are free to either substitute α for β or β for α.

Constraint generation: $\Gamma \vdash e : \tau \rightsquigarrow \Xi$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow \emptyset}\ \text{G-Var} \qquad \frac{\Gamma, x : \beta^? \vdash e : \tau \rightsquigarrow \Xi \qquad \mathit{Fresh}(\beta^?)}{\Gamma \vdash \lambda x.e : \beta^? \to \tau \rightsquigarrow \Xi}\ \text{G-Lam}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rightsquigarrow \Xi_1 \qquad \Gamma \vdash e_2 : \sigma_2 \rightsquigarrow \Xi_2 \qquad \mathit{Fresh}(\beta^?)}{\Gamma \vdash e_1\ e_2 : \beta^? \rightsquigarrow \{\sigma_1 <: \sigma_2 \to \beta^?\} \cup \Xi_1 \cup \Xi_2}\ \text{G-App}$$

$$\frac{\overline{\Gamma \vdash e : \tau \rightsquigarrow \Xi} \qquad \forall \overline{\alpha}, \overline{\alpha'}.\, C(\overline{\sigma}) : N(\overline{\alpha}; \overline{\alpha'}) \in \mathit{Ctors} \qquad \mathit{Fresh}(\overline{\beta^?}, \overline{\beta'^?})}{\Gamma \vdash C(\overline{e}) : \langle\, N(\overline{\alpha}; \overline{\alpha'}) \mid C(\overline{\sigma}) \,\rangle@(\overline{\beta^?}; \overline{\beta'^?}) \rightsquigarrow \left\{\overline{\tau <: [\![\sigma]\!]^\top_N[\overline{\beta^?}/\overline{\alpha}, \overline{\beta'^?}/\overline{\alpha'}]}\right\} \cup (\bigcup_i \Xi_i)}\ \text{G-Ctor}$$

$$\frac{\begin{array}{c}\overline{\Gamma, x : \beta^? \vdash e : \tau \rightsquigarrow \Xi} \\ \Gamma \vdash e : \sigma \rightsquigarrow \Xi\end{array} \qquad \mathit{Fresh}(\overline{\beta^?}, \gamma^?, \overline{\delta^?}, \overline{\delta'^?}) \qquad \begin{array}{c}\overline{C(\overline{\beta^?}) \heartsuit_N C(\overline{\sigma}[\overline{\delta^?}/\alpha, \overline{\delta'^?}/\alpha']) \rightsquigarrow \Xi_\heartsuit} \\ \overline{\forall \overline{\alpha}, \overline{\alpha'}.\, C(\overline{\sigma}) : N(\overline{\alpha}; \overline{\alpha'}) \subseteq \mathit{Ctors}}\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{C(\overline{x}) \Rightarrow e}\} : \gamma^? \rightsquigarrow \left\{\sigma <: \langle\, N(\overline{\alpha}; \overline{\alpha'}) \mid \overline{C(\overline{\sigma})} \,\rangle@(\overline{\delta^?}; \overline{\delta'^?}),\ \overline{\tau <: \gamma^?},\right\} \cup \Xi \cup \{\bigcup_i \Xi_i\} \cup \left\{\bigcup_i \Xi_{\heartsuit,i}\right\}}\ \text{G-Case}$$

$$\frac{}{C(\overline{\beta^?}) \heartsuit_N C(\overline{\tau}) \rightsquigarrow \left\{\overline{[\![\tau]\!]^\bot_N <: \beta^? <: [\![\tau]\!]^\top_N}\right\}}\ \text{G-Compat}$$

(a) Constraint generation rules. Inputs are contexts and terms, outputs are types and constraint sets.

Constraint solver step: $S \twoheadrightarrow S'$

$$\frac{q \in ca}{ca; q, qs \vdash bs \twoheadrightarrow ca; qs \vdash bs}\ \textsc{CacheHit}$$

$$\frac{q \notin ca \qquad q = \alpha^? <: \sigma \qquad bs(\alpha^?) = lbs <: \alpha^? <: ubs}{ca; q, qs \vdash bs \twoheadrightarrow ca, q; \{lb <: \sigma\}_{lb \in lbs}, qs \vdash bs[\alpha^? \mapsto lbs <: \alpha^? <: \{ubs, \sigma\}]}\ \textsc{UpperBound}$$

$$\frac{q \notin ca \qquad q = \sigma <: \alpha^? \qquad \sigma \notin \textsc{TyVar} \qquad bs(\alpha^?) = lbs <: \alpha^? <: ubs}{ca; q, qs \vdash bs \twoheadrightarrow ca, q; \{\sigma <: ub\}_{ub \in ubs}, qs \vdash bs[\alpha^? \mapsto \{\sigma, lbs\} <: \alpha^? <: ubs]}\ \textsc{LowerBound}$$

$$\frac{q \notin ca \qquad q = \sigma_1 <: \sigma_2 \qquad \sigma_1 \notin \textsc{TyVar} \qquad \sigma_2 \notin \textsc{TyVar} \qquad \mathbf{Sub}(q) = qs'}{ca; q, qs \vdash bs \twoheadrightarrow ca, q; qs' +\!\!+ qs \vdash bs}\ \textsc{SubOk}$$

$$\frac{q \notin ca \qquad q = \sigma_1 <: \sigma_2 \qquad \sigma_1 \notin \textsc{TyVar} \qquad \sigma_2 \notin \textsc{TyVar} \qquad \mathbf{Sub}(q) = \mathbf{Fail}}{ca; q, qs \vdash bs \twoheadrightarrow \mathbf{Fail}}\ \textsc{SubFail}$$

(b) The biunification algorithm.

Decomposing non-atomic constraints: $\mathbf{Sub}(-) : q \to \overline{q}/\mathbf{Fail}$

$$\begin{array}{rcl@{\qquad}rcl}
\mathbf{Sub}(\tau <: \top) & := & \emptyset & \mathbf{Sub}(\bot <: \sigma) & := & \emptyset \\
\mathbf{Sub}(\tau_1 \sqcup \tau_2 <: \sigma) & := & \{\tau_1 <: \sigma, \tau_2 <: \sigma\} & \mathbf{Sub}(\tau <: \sigma_1 \sqcap \sigma_2) & := & \{\tau <: \sigma_1, \tau <: \sigma_2\} \\
\mathbf{Sub}(\tau <: \mu\alpha.\sigma) & := & \{\tau <: \sigma[\mu\alpha.\sigma/\alpha]\} & \mathbf{Sub}(\mu\alpha.\tau <: \sigma) & := & \{\tau[\mu\alpha.\tau/\alpha] <: \sigma\} \\
\multicolumn{3}{r@{\qquad}}{\mathbf{Sub}(\alpha <: \alpha) := \emptyset} & & & \\
\multicolumn{3}{r@{\qquad}}{\mathbf{Sub}(\sigma_1 \to \tau_1 <: \sigma_2 \to \tau_2) := \{\sigma_2 <: \sigma_1, \tau_1 <: \tau_2\}} & & & \\
\multicolumn{3}{r@{\qquad}}{\mathbf{Sub}(\tau@(\overline{\sigma}; \overline{\rho}) <: \tau'@(\overline{\sigma'}; \overline{\rho'})) := \{\tau <: \tau', \overline{\sigma <: \sigma'}, \overline{\rho' <: \rho}\}} & & & \\
\multicolumn{3}{r@{\qquad}}{\mathbf{Sub}(\langle\, N(\overline{\alpha}; \overline{\alpha'}) \mid \emptyset \,\rangle <: \langle\, N(\overline{\beta}; \overline{\beta'}) \mid \psi \,\rangle) := \emptyset} & & & \\
\multicolumn{3}{r@{\qquad}}{\mathbf{Sub}(\langle\, N(\overline{\alpha}; \overline{\alpha'}) \mid C(\overline{\tau}), \varphi \,\rangle <: \langle\, N(\overline{\beta}; \overline{\beta'}) \mid C(\overline{\sigma}), \psi \,\rangle) := \{\tau_i <: \sigma_i[\overline{\alpha}/\overline{\beta}]\} \cup \mathbf{Sub}(\langle\, N(\overline{\alpha}; \overline{\alpha'}) \mid \varphi \,\rangle <: \langle\, N(\overline{\beta}; \overline{\beta'}) \mid \psi \,\rangle)} & & & \\
\end{array}$$

(c) Decomposing subtyping constraints.

Figure 12. Type inference.

possible that type variables refer to themselves recursively through their lower or upper bounds.

If neither σ_1 nor σ_2 are type variables, we invoke the **Sub**(−) function to decompose the constraint into a list of subconstraints. This function is specified in Figure 12c. If none of the rules in that figure match, then the function **Sub**(−) returns **Fail**.

4.4 Type Simplification

The principal types in vanilla Hindley Milner type inference have the useful property that they are syntactically unique[4]. For this reason, it is not necessary to simplify or normalize the principal types that have been inferred in such a system. The situation is different in a subtyping system, where there are usually many equivalent but syntactically distinct principal types for any typeable term. For example, the term $\lambda x\ y\ z.\mathbf{case}\ x\ \mathbf{of}\ \{\texttt{True} \Rightarrow y, \texttt{False} \Rightarrow z\}$ can be given the syntactically distinct but equivalent types $\forall\alpha\beta.\mathbf{Bool} \to \alpha \to \beta \to \alpha \sqcap \beta$ and $\forall\alpha.\mathbf{Bool} \to \alpha \to \alpha \to \alpha$. In this case, we would prefer to display the second type to the user, since it is syntactically shorter and easier to understand.

There are two main approaches to simplify types. The direct approach presented by Parreaux [14, Section 4] works directly on types and has a simpler implementation. The alternative is the automata-based approach of Dolan [3]. It uses a close correspondence between types and their encoding in finite automata, witnessed by a *representation theorem* [3, Section 7]. This theorem states that any two type automata represent the same type *iff* they accept the same language. Therefore, any algorithm for simplifying finite automata can be used to simplify types, which permits the use of generic, well-known algorithms from automata theory.

For our purposes, we assume the same equivalence for our type automata, using them to simplify our inferred types, though we do not prove the corresponding theorem.

As an example, let us observe how the type of the Boolean negation function is simplified. The negation function is given by

$$\mathbf{def}\ \texttt{not} := \lambda b.\ \mathbf{case}\ b\ \mathbf{of}\ \{\texttt{True} \Rightarrow \texttt{False}, \texttt{False} \Rightarrow \texttt{True}\}.$$

The inferred type is somewhat convoluted. Figure 13 depicts the type automata as generated initially along with the type automata after the determinisation step. The simplified inferred type of `not` is—as one would expect—

$$\langle\ \mathbf{Bool} \mid \texttt{True}, \texttt{False}\ \rangle \to \langle\ \mathbf{Bool} \mid \texttt{True}, \texttt{False}\ \rangle.$$

Meet and join types correspond to non-deterministic transitions, while equi-recursive types correspond to cyclical type automata, as shown in Figure 14. Here we can see the necessity of the polarity restriction for recursive types discussed in Section 4.1: Occurrences of μ-bound variables are represented by cyclical transitions back to the state corresponding to the recursive type; as such, they must possess the same polarity.

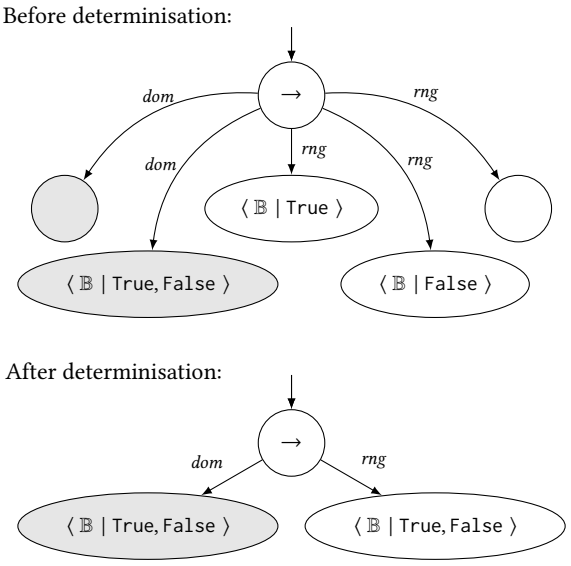

Figure 13. Determinisation of type automata for type $\langle\ \mathbf{Bool} \mid \texttt{True}, \texttt{False}\ \rangle \to \langle\ \mathbf{Bool} \mid \texttt{True}, \texttt{False}\ \rangle$. The shading of the states signifies polarity, where gray is negative and white is positive.

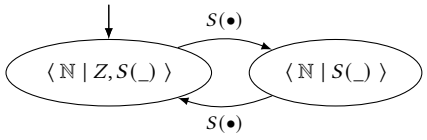

Figure 14. Type automata for refinement type of even Peano numbers $\mu\alpha.\langle\ \mathbb{N} \mid Z, S(\langle\ \mathbb{N} \mid S(\alpha)\ \rangle)\ \rangle$

Higher-kinded types are represented in the same way as types of kind $*$, except that the kind annotations (not visible in the automata figures) are different. For meets and joins, only nodes with the same kind may be merged during the determinisation. Type application is handled analogously to the representation of the function type. An example for an automaton involving higher-kinded types can be found in Figure 15 which depicts the type of a singleton list containing only the value `True`.

5 Related Work

We have identified the following areas of related work:

Algebraic Subtyping: The elegant combination of type inference, parametric polymorphism and subtyping has been a longstanding problem. The algebraic subtyping approach was developed as a solution to this problem by Dolan [3]

[4] Unique up to α-equivalence, of course.

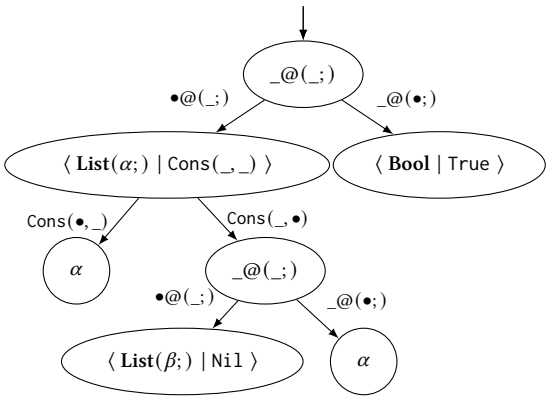

Figure 15. Type automaton for type of one-element **Bool** list $\langle\ \mathbf{List}(\alpha;)\ |\ \mathtt{Cons}(\alpha, \langle\ \mathbf{List}(\beta;)\ |\ \mathtt{Nil}\ \rangle @(\alpha;))\ \rangle @(\langle\ \mathbf{Bool}\ |\ \mathtt{True}\ \rangle;)$

and Dolan and Mycroft [4]. Algebraic subtyping is the central foundation on which our paper builds. We have also profited from the presentation of the underlying algorithm and ideas of the algebraic subtyping approach by Parreaux [14]. Parreaux and Chau [15] have recently been working on extending the algebraic subtyping approach by lifting the polarity restriction and allowing unions and intersections in arbitary positions in types. We plan to investigate the possible interaction of their extended system with our structural refinement types in future work.

Polymorphic variants: The algebraic subtyping approach is, in general, not particularly concerned with the concrete types available in the system, only with the algebraic properties of the subtyping lattice. For example, [3, 4] illustrate their approach with a system which only contains booleans, functions and records. In this paper we study the combination of ordinary algebraic data types, which all languages in the ML lineage support, with polymorphic variants. Polymorphic variants [7, 8] allow to program with the familiar tools of functional programming: building up data with constructors and decomposing with pattern matching. But they don't require the programmer to declare the data types and their constructors in advance. Structural refinement types combine the typing rules for algebraic data types with the typing rules of polymorphic variants.

Record subtyping and codata types: There is a well-known duality between data types and codata types [1, 5, 9, 17]. A special instance of this duality is the duality of polymorphic variants and extensible records [8]. In fact, we have developed and implemented the ideas presented in this paper in the more general context of algebraic data and codata types. We have specialized the formalization to data types and polymorphic variants in order to simplify the presentation, but the extension to the more general setting is straightforward.

Refinement types: Refinement types in our sense were introduced in a seminal paper by Freeman and Pfenning [6]. They conceive of refinement types as abstract domains, which have to be manually specified by the programmer in advance. They require a finite number of refinement types in order to keep type inference decidable within the intersection type system they use.

Another recent system with a very similar aim are the intensional refinement types of Jones and Ramsay [12]. In distinction to our system, they can only express the complete absence of a constructor in a refinement type. For example, they cannot express the type of nonempty lists, since that type requires to exclude the `nil` constructor only as the outermost constructor. A huge benefit of their paper, on the other hand, is the detailed study of the computational cost of tracking type refinements. We have not yet attempted to characterize the computational cost of our refinements theoretically or empirically.

Variance of type parameters The idea of co- and contravariant type parameters has been explored in-depth in the literature on object-oriented subtyping [11]. A significant trade-off in the design of a variance mechanism is the difference between use-site and definition-site variance [2]. With regard to this distinction, structural refinement types as presented here offer definition-site variance.

6 Future Work

We have implemented the refinement types described in this paper in the Duo language, and we plan to provide both a theoretical and empirical evaluation of that system. We plan to provide proofs of the soundness of the system as well as the principal types property in the future. We conjecture that it is relatively straight-forward to modify the existing proofs provided by Dolan [3] and Parreaux [14]. For an empirical validation we plan to implement larger case studies which can help evaluate whether the expressivity of the refinement types is useful in practice, and whether the type inference algorithm scales well with larger programs.

7 Discussion

We have presented a type inference algorithm for *structural refinement types*, which combines properties of both algebraic data types and polymorphic variants. These novel types allow us to express refinements on algebraic data types which correspond to regular sublanguages. In contrast to other refinement type systems, our approach uses only familiar techniques from constraint based type inference. As a result, the addition of these types to a language which already implements the algebraic subtyping approach is comparatively simple. We have implemented *structural refinement types* in the Duo language[5].

[5]Publically available at github.com/duo-lang.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming infinite structures by observations. In *Proceedings of the Symposium on Principles of Programming Languages* (Rome, Italy). ACM, New York, 27–38. https://doi.org/10.1145/2480359.2429075

[2] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. 2011. Taming the Wildcards: Combining Definition- and Use-Site Variance. In *Proceedings of the Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, 602–613. https://doi.org/10.1145/1993498.1993569

[3] Stephen Dolan. 2017. *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR.

[4] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, 60–72. https://doi.org/10.1145/3009837.3009882

[5] Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *Proceedings of the European Symposium on Programming*. Springer, 119–146. https://doi.org/10.1007/978-3-030-17184-1_5

[6] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the Symposium on Principles of Programming Languages* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, 268–277. https://doi.org/10.1145/113445.113468

[7] Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13. Baltimore. https://cir.nii.ac.jp/crid/1571698600108365312

[8] Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants*. Technical Report. Citeseer.

[9] Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629–650. https://doi.org/10.1016/S0747-7171(89)80065-3

[10] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[11] Atsushi Igarashi and Mirko Viroli. 2006. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *Transactions on Programming Languages and Systems* 28, 5 (sep 2006), 795–847. https://doi.org/10.1145/1152649.1152650

[12] Eddie C Jones and Steven J. Ramsay. 2021. Intensional datatype refinement: with application to scalable verification of pattern-match safety. *Proceedings of the Symposium on Principles of Programming Languages* 5 (2021), 1–29. https://doi.org/10.1145/3434336

[13] Klaus Ostermann. 2008. Nominal and Structural Subtyping in Component-Based Programming. *Journal of Object Technology* 7, 1 (2008), 121–145. https://doi.org/10.5381/jot.2008.7.1.a4

[14] Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). In *Proceedings of the International Conference on Functional Programming*, Vol. 4. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3409006

[15] Lionel Parreaux and Chun Yin Chau. 2022. MLscript: Principal Type Inference in a Boolean Algebra of Structural Types. https://www.cse.ust.hk/~parreaux/papers/mlscript-boolean-algebra-structural-types Conditionally accepted (OOPSLA '22).

[16] François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

[17] Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem. In *Proceedings of the International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP '15)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/2784731.2784763

[18] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[19] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161