A Mechanical Soundness Proof for Subtyping Over Recursive Types

Timothy Jones David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
{tim,djp}@ecs.vuw.ac.nz

ABSTRACT

Structural type systems provide an interesting alternative to the more common nominal typing scheme. Several existing languages employ structural types in some form, including Modula-3, Scala and various extensions proposed for Java. However, formalising a recursive structural type system is challenging. In particular, the need to use structural coinduction remains a hindrance for many. We formalise in Agda a simple recursive and structural type system with products and unions. Agda proves useful here because it has explicit support for coinduction and will raise an error if this is misused. The implementation distinguishes between inductively and coinductively defined types: the former corresponds to a finite representation, such as found in source code or the internals of a compiler, while the latter corresponds to a mathematical ideal with which we can coinductively define relations and proofs that are easily applied back to the inductive interpretation. As an application of this, we provide a mechanised proof of subtyping soundness against a semantic embedding of the types into Agda.

1. INTRODUCTION

Statically typed programming languages typically lead to programs which are more efficient and where errors are easier to detect ahead-of-time [1, 2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. One aspect affecting the flexibility of a static type system is the choice to employ a *nominal* or *structural* type system. In a nominal type system, relationships between types must be declared explicitly by the programmer. In a structural type system, on the other hand, relationships between types are implicit, based on their structure. The key advantage of structural typing is that the programmer need not identify all subtyping relationships beforehand [3, 4, 5]. This frees them from having to plan for all possible scenarios, and exposes the possibility of unanticipated reuse.

Whilst numerous mainstream languages employ nominal typing, there are relatively few which employ structural typing. Examples include OCaml [6], Modula-3 [7], Strongtalk [8], Scala [9], Whiley [10] and Grace [11]. One reason for this is that such languages are (typically) much harder to formalise and implement. In this paper, we focus on the former — that is, in the formalisation of recursive structural type systems.

The use of structural induction in proving properties of languages and their type systems is, by now, well established. However, formalising the subtype relation for a recursive type system requires some mechanism for ensuring termination. That is, since recursive types describe infinite structures, the subtype algorithm could (in principle) recurse forever. Coinduction offers a good solution here which has proved useful for defining and reasoning about properties of recursive type systems [12, 13, 14, 15]. Compared with structural induction, however, structural *coinduction* is not as widespread or as well understood. Kozen and Silva recently argued that [16]:

> *"Coinduction, on the other hand, is still mysterious and unfamiliar to many"*

Kozen and Silva were interested in promoting the use of coinduction in Computer Science, and identified a simple motto:

> *"A property holds by induction if there is good reason for it to hold; whereas a property holds by coinduction if there is no good reason for it not to hold"*

In particular, with induction one establishes a base case and inductive step but, with coinduction, one need only establish the *coinductive step*.

The main contribution of this paper is an Agda formalisation of a simple recursive structural type system involving products and unions.[1] Agda is particularly suited to this task due to its support for coinduction: an Agda program will not type check if coinduction is used incorrectly. With Agda, we establish a mechanised soundness proof of subtyping with respect to a semantic interpretation of types. However, the dual proof of completeness is not upheld by the relation and remains as future work. This improves upon the previous work of Danielsson and Altenkirch who also formalised a recursive type system with Agda, but did not establish soundness or completeness with respect to a semantic interpretation [17]. Furthermore, they did not consider products and unions, but only function types with $\top$ and $\bot$.

Our formalism distinguishes *inductively* defined types from *coinductively* defined types: the former corresponds to a finite representation, such as found in a program's source code or in the internals of a compiler, whereas the latter corresponds to a mathematical ideal with which we can coinductively formulate the subtype operator. With Agda, we establish a formal connection between these two representations of types, thereby ensuring that results obtained for coinductive types also hold for the inductive form.

[1] Available at https://github.com/zmthy/recursive-types/tree/ftfjp16

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. MOTIVATION

Our interest in reasoning about and implementing recursive types stems from our work on developing the Whiley [10, 18, 19] and Grace [11, 20, 21] programming languages.

2.1 Whiley

Whiley supports recursive structural types with records, unions, and more. As such, the compiler requires efficient representations of recursive types, and sound decision procedures for reasoning about *contractiveness*, *inhabitability*, and *subtyping*. The following illustrates the syntax for recursive types:

```
type Node is { any data, List next }
type List is null | Node
```

Here, the type `{any data, List next}` indicates a record with two fields, `data` and `next`, while the combinator `|` indicates a union type. Thus, a `List` is either `null` or a record with the structure `{any data, List next}`. A simple function operating over `List`s is given as follows:

```
function length(List list) → int:
    if list is null:
        return 0
    else:
        return 1 + length(list.next)
```

This counts the number of nodes in a list. The type test operator, `is`, distinguishes the two cases. Whiley employs *flow typing*, meaning `list` is retyped to `{any data, List next}` automatically on the false branch [22, 23]. To illustrate subtyping of recursive types, consider:

```
// A list with at least one node
type NonEmptyList is { any data, List next }

// A list containing integer data
type IntList is {int data, IntList next} | null
```

As data is immutable, both of the above types are implicit subtypes of `List`. Thus, any variable of type `NonEmptyList` or of type `IntList` can be passed into the function `length`. The Whiley Compiler accepts the above recursive types without trouble. However, it is possible to write recursive types which should be rejected by the compiler. For example:

```
type Invisible is Invisible | Invisible
```

This type does not sensibly describe any type, as it never actually describes the structure of a value, and the compiler should report an error. Likewise, consider another example:

```
type InfList is { int data, InfList next }

function get(InfList l) → (int d, InfList r):
   return l.data, l.next
```

In languages with lazy evaluation or implicit references, such a type would be perfectly reasonable [24]. However, in Whiley, all values are trees of *finite* depth and, hence, it is impossible construct a value of type `InfList`. Again, the compiler should report an error in such case.

2.2 Grace

As a language with structural typing, Grace uses recursive types in the standard manner to permit methods to accept and return values of the object they are defined in. The infinite list type from above is defined as:

```
type InfList = { data → Int; next → InfList }
```

Unlike Whiley, the body of a Grace type consists of method signatures rather than field declarations, so rather than being empty, this type includes any object which implements the appropriate methods. Field declarations with `def` implicitly introduce the appropriate getter method, so the following declaration is well-typed:

```
def ones : InfList = object {
    def data : Int = 1
    method next → InfList { self }
}
```

As values in an object-oriented language tend to be references, they make it simple to implement cyclic values. We can also implement the `InfList` type using only fields: the conceptual structure is infinite, but the actual runtime structure is a finite tree whose leaves are object references. The `next` method could be replaced with `def next : InfList = self`, and the resulting code would have the same behaviour, without requiring an infinite amount of time to construct the (conceptually) infinite structure.

3. SYNTAX, SEMANTICS & SUBTYPING

We now present our calculus for formalising recursive structural types and related operators (e.g. subtyping). The calculus is presented in the usual manner, alongside our mechanical Agda implementation. A key aspect of our calculus is the distinction between *inductive* and *coinductive* interpretations of recursive types.

3.1 Inductive Syntax

We begin by considering recursive types with a finite representation. Such types are those which have a physical machine representation. For example, they might be written in the source code of a programming language or implemented using a data structure within a compiler. Either way, they are described in some finite notation. The language of inductively defined types is as follows:

$$\hat{\mathtt{T}} ::= \mathtt{Int} \mid \hat{\mathtt{T}}_1 \mathbin{\hat{\times}} \hat{\mathtt{T}}_2 \mid \hat{\mathtt{T}}_1 \mathbin{\hat{\vee}} \hat{\mathtt{T}}_2 \mid \mu\mathtt{X}.\,\hat{\mathtt{T}} \mid \mathtt{X}$$

Unions of the form $\hat{\mathtt{T}}_1 \mathbin{\hat{\vee}} \hat{\mathtt{T}}_2$ represent types whose values are in at least one of the two component types. Recursive types are described syntactically using the common notation, $\mu\mathtt{X}.\,\hat{\mathtt{T}}$ [25, 26, 27, 13], with recursive variables $\mathtt{X}$ referring back to the type expressed by its binder. For example, $\mu\mathtt{X}.\,\mathtt{Int} \mathbin{\hat{\vee}} (\mathtt{Int} \mathbin{\hat{\times}} \mathtt{X})$ describes a non-empty list. The inductive form carries the artefacts of being a physical representation: the recursive type $\mu\mathtt{X}.\,\mathtt{Int} \mathbin{\hat{\vee}} (\mathtt{Int} \mathbin{\hat{\times}} \mathtt{X})$ is syntactically distinct from $\mu\mathtt{X}.\,\mathtt{Int} \mathbin{\hat{\vee}} \big(\mathtt{Int} \mathbin{\hat{\times}} (\mathtt{Int} \mathbin{\hat{\vee}} (\mathtt{Int} \mathbin{\hat{\times}} \mathtt{X}))\big)$, despite describing the same infinite tree.

When encoding syntactic types in Agda, our first challenge is to avoid issues of variable capture. Rather than using named type variables, we use De-Bruijn indices as is commonly done for the λ-calculus [28]. Our definition is:

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
  _×_ : (A B : InductiveType n) → InductiveType n
  _∨_ : (A B : InductiveType n) → InductiveType n
  μ_  : (A : InductiveType (suc n)) → InductiveType n
  Var : (x : Fin n) → InductiveType n
```

An InductiveType n represents a syntactic type nested within n enclosing μ terms. For example, $\hat{\mathtt{T}}$ in $\mu\mathtt{X}.\ \mathtt{Int} \mathbin{\hat{\vee}} \hat{\mathtt{T}}$ is nested within one enclosing μ term. Such a type corresponds to the Agda term $\mu\ \mathsf{Int} \vee T$, where T is the Agda encoding of $\hat{\mathtt{T}}$. The body of a μ term has type InductiveType (suc n) – i.e. is in the context of $n+1$ enclosing μ terms. A critical feature of our representation is that the body of a Var term (which represents type variables) is a natural x drawn from Fin n — that is, the finite set $\{0 \ldots n{-}1\}$. This feature ensures InductiveTypes never refer to unbound recursive variables.

Well-Formedness. Well-formed recursive types are required to be *contractive* to prohibit non-sensical types of the form $\mu\mathtt{X}.\ \mathtt{X}$ and $\mu\mathtt{X}.\ \mathtt{X} \mathbin{\hat{\vee}} \mathtt{X}$, etc [25]. A contractive tree may have infinite depth, but not infinite breadth, which applies to our types if we consider combinations of zero or more of the binary unions in a disjunctive normal form with arbitrary many branches: the types above respectively correspond to no branches and infinite branches, neither of which describe a sensible type.

DEFINITION 1 (INDUCTIVE WELL-FORMEDNESS). *A type $\hat{\mathtt{T}}$ is* well-formed *if every occurrence of a μ-bound variable in the body is separated from its binder by at least one $\hat{\times}$.*

This definition is adapted from Pierce [28], though union types were not part of the system he considered. It is useful to consider why $\hat{\times}$ is included, but not $\hat{\vee}$. We refer to $\hat{\times}$ as a *concrete constructor* and $\hat{\vee}$ as an *abstract constructor*. The distinction is that the former is represented in the language of runtime values, whilst the latter is not. Our definition does exclude some types that one might consider sensible: for example, $\mu\mathtt{X}.\ \mathtt{Int} \mathbin{\hat{\vee}} \mathtt{X}$ is not well-formed but is otherwise equivalent to $\mathtt{Int}$. We don't believe this exclusion is particularly concerning, and doing so enables a simple definition of well-formedness. Likewise, our definition also includes types which might not appear sensible: for example, $\mu\mathtt{X}.\ \mathtt{X} \mathbin{\hat{\times}} \mathtt{X}$ is well-formed but does not correspond to any runtime value (i.e. since they must have finite height).

A key challenge with our inductive Agda encoding of syntactic types lies in restricting them to be well-formed under Definition 1. To do this requires an encoding of contractive types as follows:

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
  int   : WF m Int
  pair  : ∀ {A B} → WF zero A → WF zero B → WF m (A × B)
  union : ∀ {A B} → WF m A → WF m B → WF m (A ∨ B)
  rec   : ∀ {A} → WF (suc m) A → WF m (μ A)
  ref   : ∀ {x} → m ≤ inject₁ x → WF m (Var x)
```

The WF type is indexed by an InductiveType, and an element of WF is a proof of well-formedness for that type. The intuition is that we introduce an additional counter m ranging over the set $\{0 \ldots n\}$ which indicates how many variables are currently invalid. That is, the number of enclosing μ terms encountered in a traversal (indicated by the use of suc in the rec rule) which have yet to reach a $\times$ term. When a $\times$ term is encountered the counter is reset to zero to indicate that all variables in scope are now valid.

Variable introductions such as $\forall\ \{A\ B\}$ indicate that the variables A and B will be introduced implicitly (i.e. they will not appear in introduction or elimination of these rules, with their values inferred from the other arguments) and will have their types inferred from their use later in the type of the rule. The inject_1 function trivially raises the upper bound of a Fin value: in order to compare m and x, they must be of the same type, so the upper bound of x (n) is raised to the upper bound of m (suc n).

WF is an important bridge between our inductive and coinductive interpretations: once you know that a type in its inductive form is contractive, you can switch to reasoning in the more convenient coinductive representation. Our mechanisation also demonstrates that contractivity of an inductively defined type is decidable.

3.2 Coinductive Syntax

We now consider the same syntax of types in a coinductive definition. As discussed earlier, there are types which are distinct in their inductive form, but describe the same infinite tree. In order to avoid having to reason about folding and unfolding the inductive representation, we present a coinductive form of the same syntax, with recursive references represented by infinite depth. The language of coinductive types is given as follows:

$$\mathtt{T} ::= \mathtt{Int} \mid \mathtt{T}_1 \times \mathtt{T}_2 \mid \mathtt{T}_1 \vee \mathtt{T}_2$$

As is common, we assume that coinductive types are finitely branching regular terms which may have infinite depth. The restriction to finitely branching regular terms is essential as it ensures every coinductive type can be mapped to one (or more) inductive types and vice versa. Furthermore, whilst inductive types are always represented finitely, coinductive types may have infinite depth. As an aside, coinductive types correspond to what is commonly referred to as *equi-recursive* types, whilst syntactic types are perhaps more similar to the notion of iso-recursive types [28].

Our encoding of coinductive types in Agda is as follows:

```
data CoinductiveType : Set where
  Int : CoinductiveType
  _×_ : (A B : ∞ CoinductiveType) → CoinductiveType
  _∨_ : (A B : CoinductiveType) → CoinductiveType
```

Here, CoinductiveType describes a set of tree-like structures which may be infinite in size. The ∞ CoinductiveType indicates the point at which this "infinity" can arise.

Well-Formedness. Recall from Definition 1 that types are restricted to being *contractive*. This definition cannot directly apply to our coinductive types as these do not include the concept of μ-bound variables. However, we can easily adapt it as follows:

DEFINITION 2 (COINDUCTIVE WELL-FORMEDNESS). *A coinductive type $\mathtt{T}$ is* well-formed *if every infinite path contains infinite occurrences of $\times$.*

An interesting observation is that, unlike for inductive types, we require no special treatment for well-formedness in the Agda encoding. This is because our encoding already prohibits infinite chains of $\vee$ terms. Thus, a CoinductiveType is, by construction, guaranteed to be well-formed according to Definition 2 above.

Unfolding. An important property for our system is that any well-formed inductive type can be infinitely unfolded to a well-formed coinductive type. This establishes the necessary connection between syntactic and coinductive types.

LEMMA 1 (SYNTACTIC UNFOLDING). *Let $\hat{\mathtt{T}}$ be a well-formed inductive type, and $\mathtt{T}$ the result of infinitely unfolding $\hat{\mathtt{T}}$. Then, $\mathtt{T}$ is well formed.*

PROOF. See ∞unfold in the Agda implementation. Because the coinductive types are well-formed by construction, an implementation of unfolding from WF to CoinductiveType which preserves the base constructors of the type suffices as a proof. As the unfolding is constructing a coinductive structure, the operation is not required to terminate, but it must be productive: all infinite operations must eventually appear behind a delay operator $\sharp$, which is responsible for the appearance of the ∞ type constructor in $\times$. □

3.3 Runtime Values

Following others, we consider a coinductive type $\hat{\mathtt{T}}$ as semantically representing the set of all runtime values which any variable of that type may hold [29, 26, 30, 31, 15]. To do this, we first define the language of runtime values as follows:

$$\psi ::= (\psi_1, \psi_2) \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$$

Unlike the coinductive type definition, we require that runtime values are finite in size. Although in principle we could support infinite regular terms here (e.g. for representing lazy non-terminating computations [24] or cyclic object structures [15, 32]), we choose not to for simplicity.

The encoding of runtime values in Agda is relatively straightforward as follows:

```
data Value : Set where
  int : ℤ → Value
  _,_ : (x y : Value) → Value
```

This simply represents values as trees terminated in the expected fashion with integer values. Since the ∞ operator is not used, we know that these are necessarily finite in size.

3.4 Semantic Interpretation

The semantic interpretation of coinductive types gives them meaning on which we can base subsequent operations (e.g. equivalence, subtyping, etc). We say that $[\![\mathtt{T}]\!]$ is the interpretation of type $\mathtt{T}$: the set of runtime values which inhabit this type.

DEFINITION 3 (SEMANTIC INTERPRETATION). *Every coinductive type* $\mathtt{T}$ *is characterized by the set of values it accepts, given by* $[\![\mathtt{T}]\!]$ *and which satisfies the following equation:*

$$\begin{aligned}[\![\mathtt{Int}]\!] &= \mathbb{Z} \\ [\![\mathtt{T}_1 \times \mathtt{T}_2]\!] &= [\![\mathtt{T}_1]\!] \times [\![\mathtt{T}_2]\!] \\ [\![\mathtt{T}_1 \vee \mathtt{T}_2]\!] &= [\![\mathtt{T}_1]\!] \cup [\![\mathtt{T}_2]\!]\end{aligned}$$

The semantic interpretation of types given above can be defined in either an *inductive* or *coinductive* fashion [24]. To better understand this, let us consider solutions to the following equation:

$$[\![\mathtt{T}]\!] = [\![\mathtt{Int} \times \mathtt{T}]\!]$$

Under an inductive interpretation of types we are interested in the *least solution* to the above equation (which is $[\![\mathtt{T}]\!] = \emptyset$ in this case). As such, the above type is said to be uninhabited under an inductive interpretation. In contrast, a coinductive interpretation corresponds to the *greatest solution* to the above equation (which, in this case, corresponds to the set of all lists of infinite length). For this paper, we are interested in the inductive interpretation as this is the natural choice for describing Whiley and Grace.

Our Agda encoding of the semantic interpretation is as follows:

```
mutual
  ⟦_⟧ : CoinductiveType → Set
  ⟦ Int ⟧     = ℤ
  ⟦ A × B ⟧ = ♭ A × ♭ B
  ⟦ A ∨ B ⟧ = ⟦ A ⟧ ⊎ ⟦ B ⟧

  data _×_ (A B : CoinductiveType) : Set where
    _,_ : ⟦ A ⟧ → ⟦ B ⟧ → A × B
```

This encoding transforms each CoinductiveType into a standard Agda type corresponding to the semantic embedding above. The transformation on its own will not terminate for infinitely-sized types, so we define it mutually with the $\times$ datatype, which naturally delays the application of $⟦\ A\ ⟧$ until the , constructor. This is a specialised form of the Rec type from $\Pi\Sigma$ [33]. The $\flat$ function reverses the $\sharp$, forcing the evaluation of the otherwise delayed A and B types.

The resulting types are disparate, and cannot be combined with a union operator as with our semantic sets above (the $\uplus$ is a tagged sum). We can use these types to embed values into a Value type.

```
data Value : Set where
  int : ℤ → Value
  _,_ : (x y : Value) → Value

embed : ∀ {A} → ⟦ A ⟧ → Value
```

This way we have an equivalent transformation from coinductive type to its underlying meaning, while ultimately interpreting the recursive types as types of Values, erasing unions in between.

3.5 Subtyping

We now consider subtyping between coinductive types. The intuition here is that we want $\mathtt{T}_1 \leq \mathtt{T}_2$ (a relation between coinductive types) to mean $[\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$ (the subset relation between semantic types). We face some challenges here because coinductive types can have infinite depth. That is, when comparing two types of infinite depth, it can be unclear whether they should be subtypes or not. However, since types are regular terms, it follows that any infinite derivation will repeat. Thus, the question of whether $\mathtt{T}_1 \leq \mathtt{T}_2$ holds for two infinite types $\mathtt{T}_1$ or $\mathtt{T}_2$ will either yield a finite proof that they do not, or will itself eventually reduce to the recursive query $\mathtt{T}_1 \leq \mathtt{T}_2$. To resolve this, we define our subtyping relation coinductively, and do so regardless of whether the underlying interpretation is inductive or coinductive. Intuitively, the use of coinduction means we can regard $\mathtt{T}_1 \leq \mathtt{T}_2$ to hold *unless we can prove it does not* [16].

The subtyping relation is give in Figure 1 where, following common convention, the use of a double line indicates a coinductive definition. We have restricted our rules to the minimum needed to show our main theorems (see below). In particular, there are no explicit rules for reflexivity and transitivity. This is particularly important for transitivity, because an explicit coinductive transitivity rule degenerates and, for example, can be used to show any two types are subtypes of each other [17].

The subtype relation is encoded in Agda as follows:

```
data _≤_ : (A B : CoinductiveType) → Set where
  int : Int ≤ Int
  pair : ∀ {A B C D} → ∞ (♭ A ≤ ♭ C) → ∞ (♭ B ≤ ♭ D)
       → A × B ≤ C × D
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
  union : ∀ {A B C} → A ≤ C → B ≤ C → A ∨ B ≤ C
```

This definition *does* mix inductive and coinductive rules, but only because this is natural in Agda. It is only necessary that the S-PAIR rule be coinductive: the other rules are either axioms or deconstruct finite structures, and so may be interpreted as either inductive or coinductive to the same effect. The definition does have to juggle delayed types: anywhere a delayed type A is used as a direct component of $\times$, it must be forced with $\flat$.

We can extend this subtyping on coinductive types to well-formed inductive types by applying the infinite unfolding of the inductive form and then considering the coinductive subtyping.

```
_<:_ : ∀ {A B} → WF zero A → WF zero B → Set
_<:_ p q = ∞unfold p ≤ ∞unfold q
```

With WF as a bridge, we get relations and proofs for the inductive syntax for free once they are defined for the coinductive syntax.

$$\frac{}{\texttt{Int} \leq \texttt{Int}}\ (\text{S-INT}) \qquad \frac{\texttt{T}_1 \leq \texttt{T}_3 \quad \texttt{T}_2 \leq \texttt{T}_4}{\texttt{T}_1 \times \texttt{T}_2 \leq \texttt{T}_3 \times \texttt{T}_4}\ (\text{S-PAIR}) \qquad \frac{\texttt{T}_1 \leq \texttt{T}_2}{\texttt{T}_1 \leq \texttt{T}_2 \vee \texttt{T}_3}\ (\text{S-LEFT}) \qquad \frac{\texttt{T}_1 \leq \texttt{T}_3}{\texttt{T}_1 \leq \texttt{T}_2 \vee \texttt{T}_3}\ (\text{S-RIGHT}) \qquad \frac{\texttt{T}_1 \leq \texttt{T}_3 \quad \texttt{T}_2 \leq \texttt{T}_3}{\texttt{T}_1 \vee \texttt{T}_2 \leq \texttt{T}_3}\ (\text{S-UNION})$$

Figure 1: Coinductively defined subtyping rules.

3.6 Subtype Soundness

As a precursor to showing our main theorems, we first establish reflexivity and transitivity. Our proofs rely on structural coinduction which is appropriate for a coinductively defined relation [16]. In essence, the key difference is that one need not establish a base case, as would be required for a regular proof-by-induction. This arises from the fact that $\texttt{T}_1 \leq \texttt{T}_2$ is assumed to be true unless shown otherwise (recall the motto of Kozen and Silva from §1).

LEMMA 2 (SUBTYPING IS REFLEXIVE). *Let* $\texttt{T}$ *be well-formed. Then, for all* $\texttt{T}$ *it follows that* $\texttt{T} \leq \texttt{T}$.

PROOF. See ≤-reflexive in the Agda implementation. □

LEMMA 3 (SUBTYPING IS TRANSITIVE). *Let* $\texttt{T}_1$, $\texttt{T}_2$ *and* $\texttt{T}_3$ *be well-formed types where* $\texttt{T}_1 \leq \texttt{T}_2$ *and* $\texttt{T}_2 \leq \texttt{T}_3$. *Then, it follows that* $\texttt{T}_1 \leq \texttt{T}_3$.

PROOF. See ≤-transitive in the Agda implementation. □

At this point, we can now establish the standard theorem of *soundness* with respect to our interpretation, $[\![\cdot]\!]$. Intuitively, soundness implies that whenever $\texttt{T}_1 \leq \texttt{T}_2$ by the rules of Figure 1 then this is correct with respect to the interpretation (i.e. $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$).

THEOREM 1 (SUBTYPING IS SOUND). *Let* $\texttt{T}_1$ *and* $\texttt{T}_2$ *be well-formed types where* $\texttt{T}_1 \leq \texttt{T}_2$. *Then,* $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$.

PROOF. See ≤-sound in the Agda implementation. □

As described above, all of these proofs are easily applied to <:.

3.7 Subtype (In)completeness

We now provide some discussion of the completeness theorem, which is currently not shown for our system. Intuitively, completeness requires that, whenever the interpretation implies $\texttt{T}_1 \leq \texttt{T}_2$ should hold (i.e. because $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$), then the rules of Figure 1 can establish this. We can state the theorem as follows:

THEOREM 2 (SUBTYPING IS COMPLETE). *Let* $\texttt{T}_1$ *and* $\texttt{T}_2$ *be well-formed types where* $[\![\texttt{T}_1]\!] \subseteq [\![\texttt{T}_2]\!]$. *Then,* $\texttt{T}_1 \leq \texttt{T}_2$.

The rules of Figure 1 are not sufficient to show this theorem. Unfortunately, there exist an infinite number of well-formed types whose interpretation is empty. One such type is $\mu\texttt{X}.\,\texttt{X} \,\hat{\times}\, \texttt{X}$. Such types are equivalent to $\bot$ and, for any type $\texttt{T}$, it should follow that $\bot \leq \texttt{T}$. Unfortunately, under the rules of Figure 1 we cannot show that, e.g. the unfolding of $\mu\texttt{X}.\,\texttt{X} \,\hat{\times}\, \texttt{X}$ is a subtype of $\texttt{Int}$. Extra rules are also required to handle distributivity over pairs: we cannot show that $(\texttt{Int} \vee \texttt{T}) \times \texttt{Int} \leq \texttt{T} \vee (\texttt{T} \times \texttt{Int})$ holds when $\texttt{T} = \texttt{Int} \times \texttt{Int}$.

To resolve these problems, we need rules of (roughly speaking) the following form (including a second distributivity rule for when the union is on the other side):

$$\frac{[\![\texttt{T}_1]\!] = \emptyset}{\texttt{T}_1 \leq \texttt{T}_2} \qquad \frac{\texttt{S} = (\texttt{T}_1 \times \texttt{T}_3) \vee (\texttt{T}_2 \times \texttt{T}_3)}{(\texttt{T}_1 \vee \texttt{T}_2) \times \texttt{T}_3 \leq \texttt{S}}$$

The bottom rule relies on the ability to determine whether a type is equivalent to $\bot$ or not. Whilst this is challenging, existing solutions are known [15, 24]. The challenge for Agda is that, as an intuitionistic logic, it is not enough to know that a type is either empty or not. Rather, it also requires a decidable operation to determine if it is one or the other. One solution to this problem would be to update the well-formedness definition to prohibit empty types. For now, this is left as future work.

4. RELATED WORK

Amadio and Cardelli were the first to show that subtype testing for recursive structural types was decidable [25]. Their system included function types, $\top$ and $\bot$, but did not distinguish between syntactic and algorithmic representations. However, they did separate semantic from algorithmic subtyping, and provided corresponding proofs of soundness and completeness. Amadio and Cardelli also did not exploit coinduction, instead preferring to define their relation axiomatically with explicit assumptions. Amadio and Cardelli established that subtyping in their system was decidable in exponential time. Kozen *et al.* improved on this by developing an $\mathcal{O}(n^2)$ algorithm [27]. Brandt and Henglein simplified the proof underlying the Amadio-Cardelli system by establishing a strong connection with coinduction [12]. Gapeyev *et al.* give an excellent overview of the relationship between subtyping and coinduction [13].

The work of Amadio and Cardelli established how to go about developing a subtyping algorithm: one sets out a semantic interpration of types, determines an appropriate subtyping algorithm and, finally, proves this algorithm sound and complete with respect to the semantic model. Numerous works have since followed this model. For example, the work of Damm [26] and similarly Aiken and Wimmers [29] considered recursive subtyping with union and intersection types. More recently, the XDuce system of Hosoya and Pierce for representing XML schemas [34].

The work of Danielsson and Alternkirch on formalising subtyping of recursive types is perhaps the closest related work [17]. They provided an Agda mechanisation of the recursive type system of Brandt and Henglein, which included only function types with $\top$ and $\bot$ [12]. Unlike us, they did not establish any soundness or completeness results with respect to a semantic interpretation. Instead, they showed how a mixed use of induction and coinduction allows the explicit transivity rule to be included. This contrasts with the more usual approach (as we have followed) of establishing transitivity as a property of the given rules, rather than as a rule itself.

With a semantic interpretation of types, a circularity can arise between the interpretation and the corresponding subtyping algorithm [14]. This occurs in the context of function types, whose natural interpretation are the functions themselves. If the definition of a function relies on the subtyping algorithm, the circularity is exposed. Frisch *et al.* addressed this problem in CDuce, which extended XDuce with function types [14, 35, 31]. Their solution was to provide a *bootstrapping* interpretation of function types. This does not interpret function types using terms of the enclosing programming language but, instead, views them simply as sets of tuples mapping inputs to outputs. Their system included function, union, intersection and negation types.

As part of their work on abstract compilation [36, 37, 38], Ancona and Corradi were concerned with typing programs written in untyped object-oriented programming languages (e.g. JavaScript, Python, etc) [15]. In this context, structural typing provides a flexible typing discipline which more closely matches the way dynamically typed programs are written. The authors chose a semantic-based approach to defining their subtyping relation which they argue is more intuitive. The key problem considered was the case for types which cannot be inductively interpreted. For example, a necessarily circular list cannot be soundly typed using an inductive

interpretation and, instead, requires a coinductive interpretation. The authors presented a top-down algorithm for semantic subtyping over coinductively interpreted types. Their system supported both record record and union types, and was shown to be sound and complete.

An alternative approach to formalising recursive types was investigated by Bonsangue *et al.* [24]. Their coalgebraic foundation provids a single framework for the formalisation of recursive types employing either an axiomatic approach, or that based on a semantic interpretation. This system included products and unions and supported a coinductive interpretation of types.

5. CONCLUSION

In this paper, we have presented an Agda formalisation of a simple recursive and structural type system with products and unions. Our formalism distinguishes between inductive and coinductive types. The former correspond to a finite representation, such as found in source code or in the internals of a compiler. The latter correspond to a mathematical ideal with which we can coinductively formulate the subtype operator. The benefit of using Agda is that we can exploit its support for coinduction to be sure that our formalisation is correct. Our main result is a mechanised proof of subtyping soundness with respect to an inductive interpretation of types. However, completeness is left for future work. The mechanisation is available at https://github.com/zmthy/recursive-types/tree/ftfjp16.

6. REFERENCES

[1] R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI*, pages 278–292. ACM Press, 1991.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64, 2007.

[3] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proc. ECOOP*, pages 260–284, 2008.

[4] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proc. ESOP*, pages 95–111, 2009.

[5] J. Gil and I. Maman. Whiteoak: introducing structural typing into java. In *Proc. OOPSLA*, pages 73–90, 2008.

[6] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon). *The Objective Caml system, Documentation and user's manual*, release 3.12 edition, 2011.

[7] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula-3 type system. In *Proc. POPL*, pages 202–212. ACM Press, 1989.

[8] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[9] The Scala programming language. http://lamp.epfl.ch/scala/.

[10] The Whiley Programming Language, http://whiley.org.

[11] The Grace programming language, http://gracelang.org.

[12] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proc. TLCA*, pages 63–81, 1997.

[13] V. Gapeyev, M. Y. Levin, and B. Pierce. Recursive subtyping revealed. *JFP*, 12(6):511–548, 2002.

[14] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proc. LICS*, pages 137–146. IEEE, 2002.

[15] D. Ancona and A. Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *Proc. ECOOP*, pages 282–307, 2014.

[16] D. Kozen and A. Silva. Practical coinduction. *Mathematical Structures in Computer Science*, pages 1–21, 2016.

[17] N. Danielsson and T. Altenkirch. Subtyping, declaratively. In *Proc. MPC*, pages 100–118, 2010.

[18] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proc. SLE*, pages 238–248, 2013.

[19] D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, pages 191–220, 2015.

[20] M. Homer, J. Noble, K. Bruce, A. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proc. DLS*, pages 17–28, 2012.

[21] T. Jones and J. Noble. Tinygrace: A simple, safe, and structurally typed language. In *Proc. FTfJP*, pages 3:1–3:6, 2014.

[22] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.

[23] D. J. Pearce. A calculus for constraint-based flow typing. In *Proc. FTfJP*, page Article 7, 2013.

[24] M. Bonsangue, J. Rot, D. Ancona, F. de Boer, and J. Rutten. A coalgebraic foundation for coinductive union types. In *Proc. ICALP*, pages 62–73, 2014.

[25] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15:575–631, 1993.

[26] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Proc. TACS*, pages 687–706, 1994.

[27] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. In *Proc. POPL*, pages 419–428, 1993.

[28] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA*, pages 31–41, 1993.

[30] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proc. ICALP*, pages 198–199, 2005.

[31] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):1–64, 2008.

[32] M. Servetto, J. Mackay, A. Potanin, and J. Noble. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In *Proc. ECOOP*, pages 205–229, 2013.

[33] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. In *Proc. FLOPS*, pages 40–55, 2010.

[34] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[35] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proc. ICFP*, pages 51–63, 2003.

[36] D. Ancona and G Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP*, pages 2–26, 2009.

[37] D Ancona, A Corradi, G Lagorio, and F Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): Can type inference meet verification? In *FoVeOOS*, pages 31–45, 2010.

[38] D Ancona and G Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into horn formulas. In *Proc. GANDALF*, pages 214–230, 2010.