An Algorithmic Framework for Recursive Structural Types

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, NZ
djp@ecs.vuw.ac.nz

October 2011

Abstract

Structural type systems provide an interesting alternative to the more common nominal typing scheme. Many existing languages employ structural types in some form, including Modula-3, Scala and various extensions proposed for Java. Previous work has addressed many aspects of structural types, such as efficient subtyping algorithms. Unlike nominal type systems, implementing a (recursive) structural type system remains a significant challenge. This is because a large gap exists between the formalisation of a recursive structural type system, and its algorithmic realisation. In this paper, we aim to reduce this gap by providing a generic framework that succinctly captures the important algorithmic issues. The framework is not tied to any particular structural type system, and can be instantiated in a variety of ways. Finally, by way of motivation, we illustrate some common instantiations of our framework.

1 Introduction

Statically typed programming languages typically lead to programs which are more efficient and where errors are easier to detect ahead-of-time [15, 4]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible in nature which helps reduce overheads and increase productivity [35, 41, 29, 8]. Indeed, recent times have seen a significant shift towards dynamically typed languages [36].

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 [6, 12, 30, 31, 24]. This frees the programmer from having to plan for all possible scenarios, and exposes the possibility of unanticipated reuse.

However, whilst numerous mainstream languages employ nominal typing, there are relatively few which employ structural typing. Examples include OCaml [28], Modula-3 [14], Strongtalk [10] and Scala [40]. One reason for this is that such languages are (typically) much harder to implement. We identify two specific tasks the would-be language designer must undertake:

1. **Syntactic Definition.** Here, one provides a syntactic definition of types in the language, along with an accompanying subtype relation. The latter is expressed in the usual manner with declarative inference rules, typically employing coinduction and special types of the form $\mu\mathtt{X}.\mathtt{T}$. One might also wish to demonstrate that subtyping is complete with respect to a semantic model [21, 16, 22].

2. **Algorithmic Definition.** Here, one turns the syntactic definition into an algorithm that can be implemented. The key challenge lies in determining a suitable underlying representation of types. Typically, a directed graph representation corresponding to a finite (tree) automaton is used.

Unfortunately, a significant gap exists between the syntactic and algorithmic definitions of a structural type system. For example, types of the form $\mu\mathtt{X}.\mathtt{T}$ do not exist in the algorithmic definition — rather, they are

back-edges in the automata. Furthermore, equivalent automata can have distinct underlying representations (e.g. isomorphic automata), making it difficult to determine when two types are the same (a point which the syntactic definition typically glosses over).

Many previous works exist on structural subtyping and related algorithms (e.g. [16, 22, 23, 13, 30, 18, 3, 27, 25, 7]). However, the majority focuses on the *syntactic* — rather than *algorithmic* — definition. In this paper, we aim to address this imbalance by providing an algorithmic framework for structural typing. This offers the would-be language designer a generic platform for implementing a range of structural type systems.

The contributions of this paper are as follows:

- We present a novel framework covering a range of recursive structural type systems. This focuses primarily on algorithmic aspects of the problem, and provides a generic platform for designing and implementing a structural type system.
- We provide several concrete instantiations of our framework, illustrating various aspects of structural type systems.

Finally, this work is motivated by our experiences with implementing a structural type system for the Whiley language [1, 38, 37].

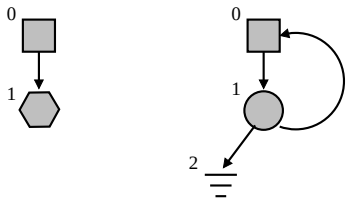
2 Overview

We now illustrate a simple structural type system, and informally introduce the main aspects of our framework. This example system will serve as a running example, and in subsequent sections we will flesh out more details.

The syntactic definition of types in our example system is given by the following:

$$\mathtt{T} ::= \mathtt{void} \mid \mathtt{any} \mid \mathtt{null} \mid \mathtt{[T]} \mid \mathtt{T}_1 \vee \mathtt{T}_2 \mid \mu\mathtt{X}.\mathtt{T} \mid \mathtt{X}$$

Here, `void` and `any` represent (respectively) $\bot$ and $\top$, `null` is used to terminate recursion and lists are given by `[T]`. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Example types in this language include `[any]` and $\mu\mathtt{X}.\mathtt{[null} \vee \mathtt{X]}$. These correspond (respectively) to the following automata in our framework:



Here, squares correspond to lists, circles to unions, hexagons to `any` and ground to `null`. Each *state* is given a unique numerical identifier, whilst edges correspond to state *transitions*. We immediately observe a difference between the syntactic representation of types (e.g. $\mu\mathtt{X}.\mathtt{[null} \vee \mathtt{X]}$) and their automata representation: namely that, unlike the other forms, $\mu\mathtt{X}.\mathtt{T}$ does not correspond to an automaton state; rather, it is implemented as a transition. Thus, whilst $\mu\mathtt{X}.\mathtt{[null} \vee \mathtt{X]}$ is syntactically distinct from $\mu\mathtt{Y}.\mathtt{[null} \vee \mathtt{Y]}$, they have an identical automata representation.

At this stage, we can now consider the main aspects of our framework: *semantic model*, *subtype algorithm*, *simplification*, *minimisation*, and *canonicalisation*

2.1 Semantic Model

An important issue in any programming language is the relationship between types and the runtime values they represent. This is often referred to as the *semantic model* of types [21, 16, 22]. A desirable property is that subtypes correspond to subsets in the model, and structural type systems lend themselves naturally to this.

To define a semantic model for our running example, we must first determine what constitutes a runtime values. In our case, this is straightforward:

$$\mathtt{V} ::= \mathtt{null} \mid \mathtt{[}\overline{\mathtt{V}}\mathtt{]}$$

Here, $\overline{\mathtt{V}}$ corresponds to a list of zero or more values. Thus, `[]`, `[null]`, `[[]]` and `[[], [null]]` are valid runtime values in our system. An *acceptance* relation determines the set of values which correspond to a given type:

$$\begin{array}{rll} \mathtt{any} & \models \mathtt{V} & \\ \mathtt{null} & \models \mathtt{null} & \\ \mathtt{[T]} & \models \mathtt{[}\overline{\mathtt{V}}\mathtt{]} & \textit{if } \forall \mathtt{V_i} \in \overline{\mathtt{V}}.\mathtt{T} \models \mathtt{V_i} \\ \mathtt{T}_1 \vee \mathtt{T}_2 & \models \mathtt{V} & \textit{if } \mathtt{T}_1 \models \mathtt{V} \textit{ or } \mathtt{T}_2 \models \mathtt{V} \end{array}$$

Thus, for example, $\mathtt{any} \models \mathtt{[[]]}$, $\mathtt{[null]} \models \mathtt{[]}$ and $\mu\mathtt{X}.\mathtt{[null} \vee \mathtt{X]} \models \mathtt{[[[]]]}$. We extend acceptance to a binary relation between types as follows:

Definition 1 (Type Subset) *Let* $\mathtt{T}_1 \models \mathtt{T}_2$ *denote* $\forall \mathtt{V}.(\mathtt{T}_2 \models \mathtt{V} \implies \mathtt{T}_1 \models \mathtt{V})$. *In other words, that the set of values accepted by* $\mathtt{T}_2$ *is a subset of those accepted by* $\mathtt{T}_1$.

The above definition gives a semantic notion of subtyping. The challenge now is to determine a corresponding algorithm.

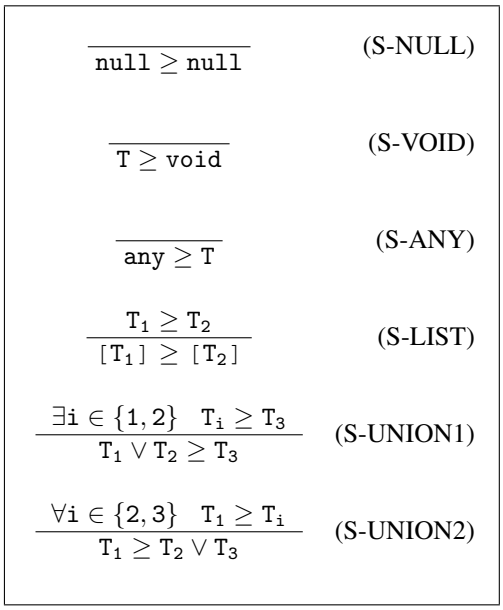


Figure 1: An algorithmic subtype operator for our running example. Note, reflexivitiy is implict and, hence, no rule for this is required.

2.2 Subtype Algorithm

Figure 1 gives an algorithmic notion of subtyping for our running example, defined in a coinductive fashion. These employ judgements of the form "$\texttt{T}_1 \leq \texttt{T}_2$", which are read as: $\texttt{T}_1$ is a subtype of $\texttt{T}_2$. For example, the following illustrates that $\texttt{T}_0 = \mu\texttt{X}.(\texttt{null} \vee [[\texttt{X}]])$ is a subtype of $\texttt{S}_0 = \mu\texttt{X}.(\texttt{null} \vee [\texttt{X}])$:

$$\begin{array}{ll} \texttt{S}_0 = \texttt{S}_1 \vee \texttt{S}_2 & \texttt{T}_0 = \texttt{T}_1 \vee \texttt{T}_2 \\ \texttt{S}_1 = \texttt{null} & \texttt{T}_1 = \texttt{null} \\ \texttt{S}_2 = [\texttt{S}_0] & \texttt{T}_2 = [\texttt{T}_3] \\ & \texttt{T}_3 = [\texttt{T}_0] \end{array}$$

1.	$\texttt{S}_0 \geq \texttt{T}_0$	(S-UNION2, 4+2)
2.	$\texttt{S}_0 \geq \texttt{T}_1$	(S-UNION1, 3)
3.	$\texttt{S}_1 \geq \texttt{T}_1$	(S-NULL)
4.	$\texttt{S}_0 \geq \texttt{T}_2$	(S-UNION1, 5)
5.	$\texttt{S}_2 \geq \texttt{T}_2$	(S-LIST, 6)
6.	$\texttt{S}_0 \geq \texttt{T}_3$	(S-UNION1, 7)
7.	$\texttt{S}_2 \geq \texttt{T}_3$	(S-LIST, 1)

In the above, we have specifically identified the subcomponents of each type, where each corresponds to a state from the (respective) automaton representation. We can see that S-INDUCT is used to prevent $\texttt{S}_0 \geq \texttt{T}_0$ from being shown a second time, as this would lead to an infinite derivation.

The remaining challenge is to show the subtyping algorithm is sound and complete with respect to the semantic model. In otherwords, that $\texttt{T}_1 \models \texttt{T}_2 \iff \texttt{T}_1 \geq \texttt{T}_2$ for any $\texttt{T}_1, \texttt{T}_2$. We return to discuss this in more detail in §**??**.

Finally, it is worth noting that we do not require a distributity rule for our example system, since one cannot distribute over lists. In §4.3.1 we will return to discuss this issue in more detail.

2.3 Simplification

In many type systems, there are immediate simplifications that can be applied. These are rewrite rules specific to the given type system, and should be *semantic preserving* (i.e. the set characterised by the type

should remain unchanged by applying the rewrite). In our running example, we have the following rewrite rules:

$$\begin{array}{rcll} \texttt{any} \vee \texttt{T} & \implies & \texttt{any} & \text{(R-ANY)} \\ \texttt{T}_1 \vee (\texttt{T}_2 \vee \texttt{T}_3) & \implies & \texttt{T}_1 \vee \texttt{T}_2 \vee \texttt{T}_3 & \text{(R-UNION1)} \\ \texttt{T}_1 \vee \texttt{T}_2 & \implies & \texttt{T}_1, \ \textbf{if}\ \texttt{T}_1 \geq \texttt{T}_2 & \text{(R-UNION2)} \\ \mu\texttt{X}.\texttt{T}_1 \vee \texttt{X} & \implies & \mu\texttt{X}.\texttt{T}_1 & \text{(R-REC1)} \\ \mu\texttt{X}.\texttt{X} & \implies & \texttt{void} & \text{(R-REC2)} \end{array}$$

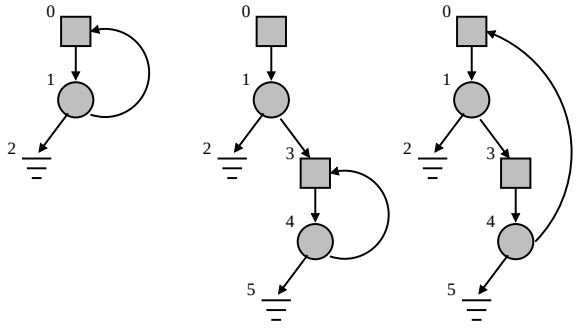
Whilst most of the above rules are straightforward, R-REC1 and R-REC2 merit some discussion. These are for dealing with so-called *contractive* types [3]. In particular, types such as $\mu\texttt{X}.(\texttt{X} \vee \texttt{X})$ and $\mu\texttt{X}.[\texttt{X}]$ are nonsensical in the context of a real programming language, as they cannot match any runtime values.

The distinction between the syntactic and algorithmic (i.e. automata) representation of our types is evident above. This is because the rewrite rules are more naturally expressed in a syntactic form, and yet they are applied to the algorithmic (i.e. automata) representation. For example, one might expect a rewrite rule such as $\texttt{T} \vee \texttt{any} \implies \texttt{any}$ which mirrors R-ANY; however, this is unnecessary since $(\texttt{any} \vee \texttt{T})$ and $(\texttt{T} \vee \texttt{any})$ have an identical algorithmic representation. Likewise, one might be concerned that producing $\mu\texttt{X}.\texttt{T}_1$ in R-REC1 leads to an "open" recursive type (e.g. $\mu\texttt{X}.\texttt{int}$). Again, this is not the case since $\mu\texttt{X}.\texttt{int}$ and $\texttt{int}$ have an identical algorithmic representation. Finally, whilst the type $\mu\texttt{X}.\texttt{X}$ is permissible under our syntactic definition of types, it cannot be represented as an automaton (because it contains no states). Such types are rejected by the compiler during parsing (i.e. before the syntactic representation is turned into an algorithmic one). Nevertheless, we include rule R-REC2 to capture how (for example) $\mu\texttt{X}.\texttt{X} \vee \texttt{X}$ reduces to $\texttt{void}$ via R-UNION2 and R-REC1.

2.4 Minimisation

The above simplification rules provide an obvious mechanism for reducing automata size. Minimisation builds on this to further reduce automata. The key difference is that, whilst simplification is domain-specific, minimisation is general and can be applied universally. Thus, the would-be language designer must provide appropriate simplification rules but, once in place, minimisation applies immediately.

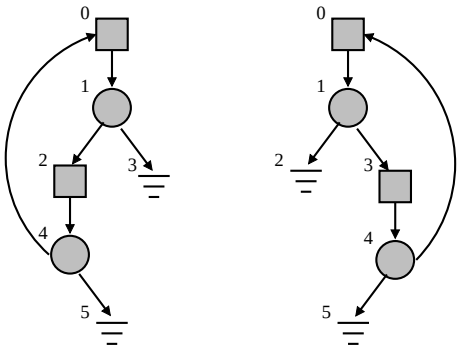
As an example, an infinite number of automata equivalent to $\mu\texttt{X}.[\texttt{null} \vee \texttt{X}]$ exist. The following illustrates three such automata:



In the rightmost automata, for example, states 1 and 4 are equivalent. That is, they accept the same set of values. The purpose of minimisation is to identify and merge such states together. As with simplification, it is important to ensure that minimisation is semantic preserving.

2.5 Canonicalisation

Canonicalisation is the process of ensuring that equivalent automata have identical representation on the machine. After simplification and minimisation, equivalent automata must have the same number and kind of states/transitions. However, equivalent automata may still be *isomorphic* and, hence, have different representations. The following illustrates two isomorphic automata for the type $\mu\texttt{X}.[\texttt{null} \vee \texttt{X}]$:



These two automata cannot be considered identical since they have a different labelling of states. The problem of determining whether automata are isomorphs reduces to the well-known graph isomorphism problem. Whilst the computational complexity of graph isomorphism remains in limbo (it is neither known to be NP-complete nor in P [26]), efficient algorithms exist for most practical purposes (see e.g. [32, 5, 33, 20]). Such algorithms operate by computing a *canonical labelling* of the graph and we apply the same approach to our automata. Intuitively, the idea behind a canonical labelling is to ensure the labels given to vertices are always the same for equivalent types.

2.6 Discussion

Whilst efficient algorithms exist for computing canonical labellings, they are likely to be more expensive than for simplification or minimisation. The cost of simplification, of course, depends on the exact rewrite rules required (but is typically $O(n)$). The minimisation algorithm, on the other hand, runs in $O(n^2)$ time.

Given this, one might wonder whether canonicalisation of types is really necessary. Indeed, in many cases, it may not be strictly necessary. However, we have found it useful for a number of reasons. As an example, consider the issue of name mangling. In compiling Whiley [1, 38, 37] to the JVM, we must employ naming mangling to encode types into function names to support overloading. This requires a unique string be generated for a type. Ideally, this remains impervious to changes elsewhere in the code or, for example, to being compiled with different compilers. That is, if recompiling the code resulted in a different string (i.e. because the new type was isomorphic to the old) then existing clients might break (because they relied on the old mangling).

3 Formalisation

In this section, we outline the core of our algorithmic framework for structural types. To help keep the presentation concrete, our example system will be used to illustrate how framework concepts map to type theory.

3.1 Automata

The core concept underlying our framework is the notion of a *automaton*. In this section, we formally define what an automaton is in the context of our framework. Our notion of an automaton is loosely based on the concept of a *tree automaton* (see e.g. [34, 17]), although there are a number of important differences.

An automaton, $\mathcal{A}$, is a map from state identifiers to state constructors. State identifiers are integers numbered contiguously from zero. State constructors are either *virtual* or *non-virtual*. The intuition is that non-virtual states corresponds to values from the programming language, whilst virtual states are only found in automata (this will become more apparent later). Virtual constructors have the form $\mathtt{c}\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$, where $\mathtt{c}$ is the constructor *kind* and $\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$ is a set of *children* (i.e. state identifiers). Non-virtual constructors have the form $\mathtt{c}(\mathtt{i}_0, \ldots, \mathtt{i}_n)$, where $\mathtt{c}$ is as before, and $\mathtt{i}_0, \ldots, \mathtt{i}_n$ is an array of zero or more child identifiers. Finally, the unique root of an automaton is always state zero.

Example. Consider the type $\mu\mathtt{X}.\,[\mathtt{null} \vee \mathtt{X}]$ for our example system. This corresponds to the following automaton:

$$\{0 \mapsto \mathtt{LIST}(1), 1 \mapsto \mathtt{UNION}\{0, 2\}, 2 \mapsto \mathtt{NULL}\}$$

Here, the kinds LIST, UNION and NULL correspond in the obvious manner to their type counterparts. We see that UNION states are virtual, whilst the remainder are non-virtual. In our example system, UNION states are the only kind of virtual state. However, other types of virtual state are possible, such as *intersections* and *negations*. □

3.2 Semantic Model

The meaning of automata is determined by the set of *terms* they accept. For a given automata $\mathcal{A}$, a term $\tau = \mathtt{c}(\tau_0, \ldots, \tau_n)$ is a tree where $\mathtt{c}$ is a constructor kind (as before) and $\tau_0, \ldots, \tau_n$ are sub-terms.

Example. Consider the runtime value `[[], []]` from our example system. This corresponds to the term $\mathtt{LIST}(\mathtt{LIST}(), \mathtt{LIST}())$. □

We now consider the acceptance relation, which is built-up by instantiating the following rules for the various constructors of the system in question:

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{c}_1, \quad \mathcal{A}(\mathtt{i}) \models \mathtt{c}_2(\tau_0, \ldots, \tau_n)}{} \text{(BASE)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{c}_1(\mathtt{j}), \quad \mathcal{A}(\mathtt{i}) \models \mathtt{c}_2(\tau_0, \ldots, \tau_n) \textbf{ if } \forall 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}(\mathtt{j}) \models \tau_\mathtt{k}}{} \text{(ONE-MANY)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{c}_1(\mathtt{j}_0, \ldots, \mathtt{j}_n), \quad \mathcal{A}(\mathtt{i}) \models \mathtt{c}_2(\tau_0, \ldots, \tau_n) \textbf{ if } \forall 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}(\mathtt{j}_\mathtt{k}) \models \tau_\mathtt{k}}{} \text{(ONE-ONE)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{c}_1\{\mathtt{j}_0, \ldots, \mathtt{j}_n\}, \quad \mathcal{A}(\mathtt{i}) \models \tau \textbf{ if } \exists 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}(\mathtt{j}_\mathtt{k}) \models \tau}{} \text{(UNION)}$$

Whilst the above rules have relatively limited structure, they can nevertheless characterise types from systems with any combination of the following constructs: *unions*, *sets*, *lists*, *tuples* and *records* and more (see §**??**). Extending to include other virtual kinds, such as *intersections* and *negations* is straightforward. However, dealing with more complex constructs, such as *functions* and *references* is more challenging.

Example. Recall the acceptance relation given for our example system in §2.1. We can now provide corresponding instantiations of the above (generic) rules:

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{ANY}, \quad \mathcal{A}(\mathtt{i}) \models \tau}{} \text{(BASE)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{NULL}, \quad \mathcal{A}(\mathtt{i}) \models \mathtt{NULL}}{} \text{(BASE)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{LIST}(\mathtt{j}), \quad \mathcal{A}(\mathtt{i}) \models \mathtt{LIST}(\tau_0, \ldots, \tau_n) \textbf{ if } \forall 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}(\mathtt{j}) \models \tau_\mathtt{k}}{} \text{(ONE-MANY)}$$

$$\frac{\mathcal{A}(\mathtt{i}) = \mathtt{UNION}\{\mathtt{j}_0, \ldots, \mathtt{j}_n\}, \quad \mathcal{A}(\mathtt{i}) \models \tau \textbf{ if } \exists 0 \leq \mathtt{k} \leq \mathtt{n} \textbf{ where } \mathcal{A}(\mathtt{j}_\mathtt{k}) \models \tau_\mathtt{k}}{} \text{(UNION)}$$

From this, we observe that no instantiation of the ONE-ONE rule is required for our example system. As we will see, this has important consequences, since it means no distributivity over subtyping is required. □

$$\frac{\mathcal{A}_1(\mathtt{i}) = \mathtt{c}_1 \quad \mathcal{A}_2(\mathtt{j}) = \mathtt{c}_2}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(BASE)}$$

$$\frac{\mathcal{A}_1(\mathtt{l}) \sqsupseteq \mathcal{A}_2(\mathtt{m}) \quad \mathcal{A}_1(\mathtt{i}) = \mathtt{c}(\mathtt{l}) \quad \mathcal{A}_2(\mathtt{j}) = \mathtt{c}(\mathtt{m})}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(ONE-MANY)}$$

$$\frac{\mathcal{A}_1(\mathtt{i}) = \mathtt{c}_1(\mathtt{l}_0, \ldots, \mathtt{l}_n) \quad \mathcal{A}_2(\mathtt{j}) = \mathtt{c}_2(\mathtt{m}_0, \ldots, \mathtt{m}_n) \quad \forall 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}_1(\mathtt{l}_k) \sqsupseteq \mathcal{A}_2(\mathtt{m}_k)}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(ONE-ONE)}$$

$$\frac{\mathcal{A}_1(\mathtt{i}) = \mathtt{c}_1\{\mathtt{l}_0, \ldots, \mathtt{l}_n\} \quad \exists 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}_1(\mathtt{l}_k) \sqsupseteq \mathcal{A}_2(\mathtt{j})}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(UNION1)}$$

$$\frac{\mathcal{A}_2(\mathtt{j}) = \mathtt{c}_1\{\mathtt{l}_0, \ldots, \mathtt{l}_n\} \quad \forall 0 \leq \mathtt{k} \leq \mathtt{n} \,.\, \mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{l}_k)}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(UNION2)}$$

$$\frac{\begin{array}{c}\mathcal{A}_2(\mathtt{j}) = \mathtt{c}_1(\mathtt{l}_0, \ldots, \mathtt{l}_i, \ldots, \mathtt{l}_n) \\ \mathcal{A}_2(\mathtt{l}_i) = \mathtt{c}_2\{\mathtt{m}_0, \ldots, \mathtt{m}_n\} \\ \mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathtt{c}_1(\mathtt{l}_0, \ldots, \mathtt{m}_0, \ldots, \mathtt{l}_n) \\ \ldots \\ \mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathtt{c}_1(\mathtt{l}_0, \ldots, \mathtt{m}_n, \ldots, \mathtt{l}_n)\end{array}}{\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})} \quad \text{(DISTRIBUTE)}$$

Figure 2: Illustrating the generic semantic subsumption rules.

We now extend out notion of acceptance to allowing comparison between automatons. An automaton $\mathcal{A}_1$ *subsumes* another automaton $\mathcal{A}_2$ if $\mathcal{A}_1$ accepts all input values accepted by $\mathcal{A}_2$ (and possibly more). Subsumption is closely related to the notion of *subtyping*, and is formally defined as follows:

Definition 2 (Semantic Subsumption) *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata. Then,* $\mathcal{A}_1$ subsumes $\mathcal{A}_2$*, denoted $\mathcal{A}_1 \models \mathcal{A}_2$ iff $\forall \tau . \mathcal{A}_2 \models \tau \implies \mathcal{A}_1 \models \tau$.*

Essentially, semantic subsumption between automata gives an idealised view of subtyping and we desire that any subsumption algorithm implements this.

3.3 Algorithmic Subsumption

Whilst the above definition provides an idealised view of subsumption, the challenge lies in constructing an algorithm to check when one automata subsumes another. To address this, we take $\mathcal{A}_1 \sqsupseteq \mathcal{A}_2$ to denote that $\mathcal{A}_1$ has been shown *algorithmically* to subsume $\mathcal{A}_2$. In the general case, one must construct a subsumption algorithm from scratch, and provide an adhoc proof of soundness and completeness. However, as with the acceptance relation, a small number of generic rules cover the majority of cases.

The generic rules for determining subsumption are shown in Figure 2. Whilst these are given in a declarative fashion, it is straightforward to construct an algorithm from these rules. The rules employ judgements of the form "$\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})$", which are read as: state $\mathtt{i}$ from $\mathcal{A}_1$ subsumes state $\mathtt{j}$ from $\mathcal{A}_2$.

Example. The subtype rules for our example system are shown in Figure 1. These represent various instantiations of the generic subsumption rules from Figure 2. For example, S-NULL, S-VOID and S-ANY are instantiations of the BASE rule. Likewise, S-LIST is an instantation of the ONE-MANY rule.

Finally, S-INDUCT, S-UNION1 and S-UNION2 correspond in the obvious manner to INDUCT, UNION1 and UNION2. □

3.3.1 Soundness and Completeness

Finally, it is desirable to establish a strong correspondence between the semantic and algorithmic notions of subsumption. This can help identify any weakness or mistakes in the system, and is formulated as *soundness* and *completeness* theorems:

Theorem 1 (Soundness) *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be automata where $\mathcal{A}_1 \sqsupseteq \mathcal{A}_2$. Then, $\mathcal{A}_1 \models \mathcal{A}_2$.*

Proof 1 *By induction over the structure of $\mathcal{A}_1$ and $\mathcal{A}_2$. The induction hypothesis states that if $\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})$ holds for some $\mathtt{i}, \mathtt{j} > \mathtt{0}$, then $\mathcal{A}_1(\mathtt{i}) \models \mathcal{A}_2(\mathtt{j})$. Each case corresponds to a rule from Figure 2:*

- *Case $\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{0})$ holds under an instance of the* BASE *subsumption rule from Figure 2. Then, $\mathcal{A}_1 \models \mathcal{A}_2$ should follow immediately from a corresponding* BASE *acceptance rule.*
- *Case $\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{0})$ holds under an instance of the* ONE-MANY *rule subsumption from Figure 2. Then, $\mathcal{A}_1(\mathtt{0}) = \mathtt{c}(\mathtt{i})$, $\mathcal{A}_2(\mathtt{0}) = \mathtt{c}(\mathtt{j})$ and $\mathcal{A}_1(\mathtt{i}) \sqsupseteq \mathcal{A}_2(\mathtt{j})$. By inductive hypothesis $\mathcal{A}_1(\mathtt{i}) \models \mathcal{A}_2(\mathtt{j})$ and, hence, $\mathcal{A}_1 \models \mathcal{A}_2$ follows from a corresponding* ONE-MANY *acceptance rule(s).*
- *Case $\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{0})$ holds under an instance of the* ONE-ONE *subsumption rule from Figure 2. Then, $\mathcal{A}_1(\mathtt{0}) = \mathtt{c}\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$, $\mathcal{A}_2(\mathtt{0}) = \mathtt{c}\{\mathtt{j}_0, \ldots, \mathtt{j}_n\}$ and $\forall \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}.\mathcal{A}_1(\mathtt{i}_k) \sqsupseteq \mathcal{A}_2(\mathtt{j}_k)$. By inductive hypothesis $\forall \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}.\mathcal{A}_1(\mathtt{i}_k) \models \mathcal{A}_2(\mathtt{j}_k)$ and, hence, $\mathcal{A}_1 \models \mathcal{A}_2$ follows from a corresponding* ONE-ONE *acceptance rule.*
- *Case $\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{0})$ holds under an instance of the* UNION1 *subsumption rule from Figure 2. Then, $\mathcal{A}_1(\mathtt{0}) = \mathtt{c}\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$ and $\exists \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}.\mathcal{A}_1(\mathtt{i}_k) \sqsupseteq \mathcal{A}_2(\mathtt{0})$. By inductive hypothesis $\mathcal{A}_1(\mathtt{i}_k) \models \mathcal{A}_2(\mathtt{0})$ and, hence, $\mathcal{A}_1 \models \mathcal{A}_2$ follows from a corresponding* UNION *acceptance rule.*
- *Case $\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{0})$ holds under an instance of the* UNION2 *subsumption rule from Figure 2. Then, $\mathcal{A}_2(\mathtt{0}) = \mathtt{c}\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$ and $\forall \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}.\mathcal{A}_1(\mathtt{0}) \sqsupseteq \mathcal{A}_2(\mathtt{i}_k)$. By inductive hypothesis we have $\forall \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}.\mathcal{A}_1(\mathtt{0}) \models \mathcal{A}_2(\mathtt{i}_k)$ and, hence, $\mathcal{A}_1 \models \mathcal{A}_2$ follows from a corresponding* UNION *acceptance rule.*

Theorem 2 (Completeness) *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be automata where $\mathcal{A}_1(\mathtt{i}) \models \mathcal{A}_2(\mathtt{j})$ Then, $\mathcal{A}_1 \sqsupseteq \mathcal{A}_2$.*

3.4 Simplification

Simplification is the most domain-specific aspect of our framework. The would-be language designer must carefully determine appropriate rewrite rules for his/her type system. Furthermore, the rules must be shown to be *confluent* (i.e. that they are unambiguous and terminating) and that they produce *simplified form*:

Definition 3 (Equivalence) *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata where $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ and $\mathcal{A}_1 \sqsupseteq \mathcal{A}_2$. Then, $\mathcal{A}_1$ and $\mathcal{A}_2$ are* equivalent*, written $\mathcal{A}_1 \equiv \mathcal{A}_2$.*

Definition 4 (Simplified Form) *An automaton, $\mathcal{A}$, is in* simplified form *if, for all $\mathtt{i}, \mathtt{j}$ where $\mathcal{A}(\mathtt{i}) \equiv \mathcal{A}(\mathtt{j})$, we have $\mathcal{A}(\mathtt{i}) = \mathtt{c}(\mathtt{l}_0, \ldots, \mathtt{l}_n)$, $\mathcal{A}(\mathtt{j}) = \mathtt{c}(\mathtt{m}_0, \ldots, \mathtt{m}_n)$ and $\forall \mathtt{0} \leq \mathtt{k} \leq \mathtt{n}\ .\ \mathcal{A}(\mathtt{l}_k) \equiv \mathcal{A}(\mathtt{m}_k)$.*

The above states that, in simplified form, any equivalent states must have the same kind, the same number of children and, furthermore, those children in the same argument position themselves be equivalent. The aim of simplification is to ensure equivalent states have identical form. This prepares the automaton for minimisation, where equivalent states are merged together.

Example. The rewrite rules for our example system are given in §2.3. Consider the type $\mathtt{any} \vee \mathtt{void}$, whose automaton is $\{0 \mapsto \text{UNION}\{1, 2\}, 1 \mapsto \text{ANY}, 2 \mapsto \text{VOID}\}$. It is clear that states 0 and 1 accept the same set of terms (namely, all possible terms). However, they have different shape and, hence, the automaton is not in simplified form. □

3.5 Minimisation

The aim of minimisation is to eliminate redundancy from our automata. That is, so they do not contain equivalent states and, hence, are as small as possible. More specifically, we desire the following:

Definition 5 (Minimisation) *Let $\mathcal{A}$ be an automaton. Then, $\mathcal{A}$ is in* minimised form *if $\forall i, j$ where $\mathcal{A}(i) \equiv \mathcal{A}(j)$, it follows that $i = j$.*

In fact, for an automaton in simplified form it is very straightforward to reduce it to minimised form as follows: for any set of equivalent states pick a representative and replace all occurences of the others in argument positions with the representative; finally, compact the automaton by eliminating unreachable states. Definition 4 ensures this operation is semantically preserving, since we know all equivalent states have identical form.

Example. Consider the type $\mu\mathtt{X}.\,[\,[\mathtt{X}]\,]$ from our example system, whose automaton is $\{0 \mapsto \text{LIST}(1), 1 \mapsto \text{LIST}(0)\}$. This accepts terms such as: `[]`, `[[]]`, `[[],[]]` and `[[[]]]`. In fact, states `0` and `1` are equivalent and, hence, this automaton is in simplified (but not minimised) form. If we pick `0` as the representative of these two equivalent states then, by replacing occurences of `1` with `0`, we arrive at: $\{0 \mapsto \text{LIST}(0), 1 \mapsto \text{LIST}(0)\}$ Here, state `1` is no longer reachable and should be eliminated. Thus, we have reduced $\mu\mathtt{X}.\,[\,[\mathtt{X}]\,]$ to the (minimised) automaton $\mu\mathtt{X}.\,[\mathtt{X}]$. □

3.6 Canonicalisation

Computing the canonical form for an automaton requires finding a *canonical labelling* of states. Whilst different canonical labelings can be employed, we follow the graph isomorphism literature (e.g.[32, 5, 33, 20]) and use the so-called *lexicographical ordering*. To make the connection more explicit, we employ a graph-like view of automata:

- For a non-virtual state, the edge $\mathtt{i} \xrightarrow{\mathtt{k}} \mathtt{j} \in \mathcal{A}$ denotes that `j` is the $\mathtt{k}^{\text{th}}$ child identifier of state `i` in $\mathcal{A}$. For example, if $\mathcal{A}(\mathtt{i}) = \mathtt{c}(\mathtt{i}_0, \ldots, \mathtt{i}_n)$ and $\mathtt{i} \xrightarrow{\mathtt{k}} \mathtt{j}$, then $\mathtt{j} = \mathtt{i}_k$.
- For a virtual state, the edge $\mathtt{i} \xrightarrow{\infty} \mathtt{j} \in \mathcal{A}$ indicates that `j` is a child of state `i` in $\mathcal{A}$. For example, if $\mathcal{A}(\mathtt{i}) = \mathtt{c}\{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$ and $\mathtt{i} \xrightarrow{\infty} \mathtt{j}$, then $\mathtt{j} \in \{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$.

In this manner, we are viewing our automata as coloured, labelled digraphs (where colour corresponds to kind, and edge labels to argument positions). We now build up towards defining the lexicographic ordering:

Definition 6 *Let $\mathtt{e}_1 = \mathtt{i} \xrightarrow{\mathtt{k}_1} \mathtt{j}$ and $\mathtt{e}_2 = \mathtt{v} \xrightarrow{\mathtt{k}_2} \mathtt{w}$ be edges. Then, we say $\mathtt{e}_1 < \mathtt{e}_2$, if $\mathtt{i} < \mathtt{v}$ or $\mathtt{i} = \mathtt{v} \wedge \mathtt{j} < \mathtt{w}$ or $\mathtt{i} = \mathtt{v} \wedge \mathtt{j} = \mathtt{w} \wedge \mathtt{k}_1 < \mathtt{k}_2$.*

Definition 7 (Edge Sequence) *Let $\mathcal{A}$ be an automaton. Then, Δ is a total ordering of all $\mathtt{i} \xrightarrow{\mathtt{k}} \mathtt{j} \in \mathcal{A}$ such that, for all $m < n$, it follows that $\Delta(m) < \Delta(n)$.*

Definition 8 (Lexicographic Ordering) *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata with corresponding edge sequences Δ_1 and Δ_2. Then, $\mathcal{A}_1$ is* lexicographically lower *than $\mathcal{A}_2$, denoted $\mathcal{A}_1 < \mathcal{A}_2$ if $\exists \mathtt{i}$ where $\Delta_1(\mathtt{i}) < \Delta_2(\mathtt{i})$ and $\forall \mathtt{j} < \mathtt{i}.\Delta_1(\mathtt{j}) = \Delta_2(\mathtt{j})$.*

The lexiographic ordering defines an ordering between automata. The intuition is that the canonical form of an automaton is an isomorphism with lowest lexicographic ordering. Finally, we can define the canonical form and give the canonicalisation theorem, which simply states that any two isomorphic automata have the same canonical form:

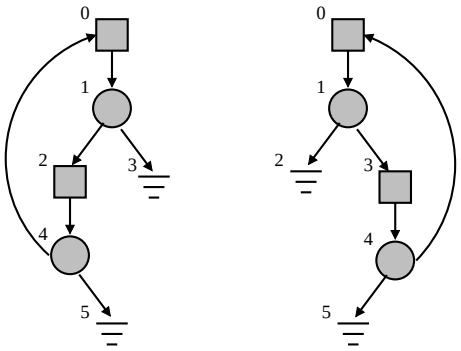
Definition 9 (Canonical Form) *Let $\mathcal{A}_1$ be an automaton. Then, its canonical form, denoted $\hat{\mathcal{A}}_1$, is an isormopism where $\hat{\mathcal{A}}_1 \leq \mathcal{A}_2$ holds for all isomorphisms $\mathcal{A}_2$ of $\mathcal{A}_1$.*

Theorem 3 (Canonicalisation) *For two automata $\mathcal{A}_1 \equiv \mathcal{A}_2$ it follows that $\hat{\mathcal{A}}_1 = \hat{\mathcal{A}}_2$.*

Proof 2 *Straightforward.*

Constructing an algorithm to find the canonical form of an automaton $\mathcal{A}$ is easy enough: simply enumerate all isomorphisms of $\mathcal{A}$ and select the lexiographically lowest. Of course, such an algorithm is not particularly efficient! Nevertheless, in practice we have found it sufficient for our purposes, given that structural types tend to be relatively small. Furthermore, significanly more efficient algorithms have been developed in the context of graph isomorphism (see e.g. [32, 5, 33, 20]).

Example. Consider again the following isomorphic automata $\mu\texttt{X}.\texttt{[null} \vee \texttt{X]}$ from §2.5:



Let Δ_1 denote the edge ordering for the left automaton, and Δ_2 that for the right automaton. Then, we have:

Δ_1	=	$0 \xrightarrow{0} 1$	$1 \xrightarrow{\infty} 2$	$1 \xrightarrow{\infty} 3$	$2 \xrightarrow{0} 4$	$4 \xrightarrow{\infty} 0$	$4 \xrightarrow{\infty} 5$
Δ_2	=	$0 \xrightarrow{0} 1$	$1 \xrightarrow{\infty} 2$	$1 \xrightarrow{\infty} 3$	$3 \xrightarrow{0} 4$	$4 \xrightarrow{\infty} 0$	$4 \xrightarrow{\infty} 5$
		0	1	2	3	4	5

Here, we can see that the orderings are identical upto index 3. At this point, it becomes clear that $\Delta_1 < \Delta_2$ and, hence, that $\mathcal{A}_1 < \mathcal{A}_2$. In fact, $\mathcal{A}_1$ is in canonical form as only two other isomorphs are possible and, in both, $4 \xrightarrow{\infty} 5$ comes before $4 \xrightarrow{\infty} 0$. □

3.6.1 Efficiency

An interesting observation is that virtual states are the sole reason that computing a canonical form is computationally challenging. In particular, for an automaton without any virtual states, the canonical form can be computed immediately as follows: starting from the root state, allocate states successive identifiers in a breadth-first fashion where children are allocated in order of appearance. Virtual nodes cause problems because they do not have an explicit ordering of children and, hence, we must search for the lowest.

This observation has interesting performance implications for a system which represents all types in canonical form. This is because the computational complexity is not exponential in the number of states; rather, it is exponential in the number of *virtual* states.

4 Extensions

4.1 Sets

As a simple example, consider instantiating our framework to support set types of the form $\{\mathtt{T}\}$. These accept corresponding set values of the form $\{\overline{\mathtt{V}}\}$ and are treated in an identical fashion to list types. To do this, we introduce a constructor kind SET, and instantiate the corresponding acceptance and subsumption rules (which are essentially identical to those for LIST).

4.2 Tuples

Instantiating our framework to support tuple types of the form $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ is also relatively straightforward. For example, $(\mathtt{any}, \mathtt{any})$ and $(\mathtt{[any]}, \mathtt{null})$ are tuple types. Tuple values of the form $(\mathtt{V}_1, \ldots, \mathtt{V}_n)$ are defined analogously. The corresponding acceptance and subtype relations are:

$$(\mathtt{T}_1, \ldots, \mathtt{T}_n) \models (\mathtt{V}_1, \ldots, \mathtt{V}_n) \quad \textit{if } \forall 0 \leq \mathtt{i} \leq \mathtt{n}.\mathtt{T}_\mathtt{i} \models \mathtt{V}_\mathtt{i}$$

$$\frac{\forall 0 \leq \mathtt{i} \leq \mathtt{n} \;.\; \mathtt{T}_\mathtt{i} \geq \mathtt{S}_\mathtt{i} \mid \mathcal{C} \cup \{(\mathtt{T}_1, \ldots, \mathtt{T}_n) \geq (\mathtt{S}_1, \ldots, \mathtt{S}_n)\}}{(\mathtt{T}_1, \ldots, \mathtt{T}_n) \geq (\mathtt{S}_1, \ldots, \mathtt{S}_n) \mid \mathcal{C}} \qquad \text{(S-TUPLE)}$$

To represent this in our framework, we require an appropriate constructor kind, TUPLE. Thus, type $(\mathtt{any}, \mathtt{null})$ is represented by the automaton $\{0 \mapsto \text{TUPLE}(1,2), 1 \mapsto \text{ANY}, 2 \mapsto \text{NULL}\}$. Now, we observe that the acceptance rule above corresponds to an instantiation of the generic ONE-ONE rule from §3.2. Furthermore, the subtype rule above corresponds to an instantiation of the generic ONE-ONE subsumption rule from Figure 2. Finally, in doing this, it becomes immediately apparent that we may distribute over tuples.

4.3 Records

Record types are similar to tuples, except they have *named* fields rather than *unnamed* fields (as for tuples). This complicates the issue of instantiating them within our framework. Let a record type have the form $\{\mathtt{T}_0\ \mathtt{f}_0, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}$ where $\mathtt{f}_k$ represents a field name. For example, $\{\mathtt{[any]\ f1}, \mathtt{null\ f2}\}$ is a record type and, furthermore, it should be indistinguishable from $\{\mathtt{null\ f2}, \mathtt{[any]\ f1}\}$. As a starting point, we have the following acceptance and subtype rules for records:

$$\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \models \{\mathtt{V}_1\ \mathtt{f}_1, \ldots, \mathtt{V}_n\ \mathtt{f}_n\} \quad \textit{if } \forall 0 \leq \mathtt{i} \leq \mathtt{n}.\mathtt{T}_\mathtt{i} \models \mathtt{V}_\mathtt{i}$$

$$\frac{\forall 0 \leq \mathtt{i} \leq \mathtt{n} \;.\; \mathtt{T}_\mathtt{i} \geq \mathtt{S}_\mathtt{i} \mid \mathcal{C} \cup \{\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \geq \{\mathtt{S}_1\ \mathtt{f}_1, \ldots, \mathtt{S}_n\ \mathtt{f}_n\}\}}{\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \geq \{\mathtt{S}_1\ \mathtt{f}_1, \ldots, \mathtt{S}_n\ \mathtt{f}_n\} \mid \mathcal{C}} \qquad \text{(S-RECORD)}$$

At this stage, things look remarkably similar as for tuples. However, a problem remains in that we must encode field names. For example, suppose we introduce a constructor kind, RECORD. Furthermore, when creating a RECORD(...) constructor, we allocate fields to argument positions in alphabetical order (as this ensures declaration order doesn't matter). However, this is still not sufficient because it doesn't encode field names. For example, $\{\mathtt{[any]\ f1}, \mathtt{any\ f2}\}$ is indistinguishable from $\{\mathtt{[any]\ f2}, \mathtt{any\ f3}\}$ as they both correspond to the automaton $\{0 \mapsto \text{RECORD}(1,1), 1 \mapsto \mathtt{ANY}\}$.

In fact, we can encode record types within our framework. The key is to encode field names into the constructor kind itself. Thus, rather than having a single kind RECORD, we have an infinite family of kinds $\text{RECORD}\langle \mathtt{f}_1, \ldots, \mathtt{f}_n \rangle$. Under this encoding, $\{\mathtt{[any]\ f1}, \mathtt{any\ f2}\}$ corresponds to the automaton $\{0 \mapsto \text{RECORD}\langle \mathtt{f}_1, \mathtt{f}_2 \rangle(1,1), 1 \mapsto \mathtt{ANY}\}$. Note that this is distinct from $\{\mathtt{[any]\ f2}, \mathtt{any\ f3}\}$ which corresponds to $\{0 \mapsto \text{RECORD}\langle \mathtt{f}_2, \mathtt{f}_3 \rangle(1,1), 1 \mapsto \mathtt{ANY}\}$. Finally, we instantiate the ONE-ONE acceptance and subsumption rules in a similar manner as for tuples.

At this point, we start to reach the limits of our framework. Record types often support the so-called *width* subtyping rule:

$$\frac{\begin{array}{c}\mathtt{m} \leq \mathtt{n} \\ \forall 0 \leq \mathtt{i} \leq \mathtt{m} \;.\; \mathtt{T}_\mathtt{i} \geq \mathtt{S}_\mathtt{i} \mid \mathcal{C} \cup \{\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_m\ \mathtt{f}_m\} \geq \{\mathtt{S}_1\ \mathtt{f}_1, \ldots, \mathtt{S}_n\ \mathtt{f}_n\}\}\end{array}}{\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_m\ \mathtt{f}_m\} \geq \{\mathtt{S}_1\ \mathtt{f}_1, \ldots, \mathtt{S}_n\ \mathtt{f}_n\} \mid \mathcal{C}} \qquad \text{(S-WIDTH)}$$

This rule is difficult to encode in our framework because fields (i.e. argument positions) are not all compared in a one-one fashion; rather, some are compared in a one-one fashion, whilst others are ignored. Of course, it's not difficult to see how one could extend our framework with generic rules to cover this case. However, it's not clear how generic such rules would be — that is, whether they could be used for situations other than width subtyping.

4.3.1 Distributivity.

Distributivity is a common issue facing structural type systems in the presence of unions. For example, with tuple types we have the following equivalence $(\texttt{null}, \texttt{any}) \vee (\texttt{[any]}, \texttt{any}) \equiv (\texttt{null} \vee \texttt{[any]}, \texttt{any})$. A rule such as the following is common for handling this:

$$\frac{\begin{array}{c}\texttt{T} = \{\texttt{T}_1\ \texttt{f}_1 \ldots, \texttt{T}_\texttt{i} \vee \texttt{T}'_\texttt{i}\ \texttt{f}_\texttt{i} \ldots, \texttt{T}_\texttt{n}\ \texttt{f}_\texttt{n}\} \\ \{\texttt{T}_1\ \texttt{f}_1 \ldots, \texttt{T}_\texttt{i}\ \texttt{f}_\texttt{i} \ldots, \texttt{T}_\texttt{n}\ \texttt{f}_\texttt{n}\} \leq \texttt{S} \lfloor \mathcal{C} \cup \{\texttt{T} \leq \texttt{S}\} \\ \{\texttt{T}_1\ \texttt{f}_1 \ldots, \texttt{T}'_\texttt{i}\ \texttt{f}_\texttt{i} \ldots, \texttt{T}_\texttt{n}\ \texttt{f}_\texttt{n}\} \leq \texttt{S} \lfloor \mathcal{C} \cup \{\texttt{T} \leq \texttt{S}\}\end{array}}{\texttt{T} \leq \texttt{S} \lfloor \mathcal{C}} \qquad \text{(S-DIST)}$$

This rule is problematic for our framework because it relies on types (namely $\{\texttt{T}_1\ \texttt{f}_1 \ldots, \texttt{T}_\texttt{i}\ \texttt{f}_\texttt{i} \ldots, \texttt{T}_\texttt{n}\ \texttt{f}_\texttt{n}\}$ and $\{\texttt{T}_1\ \texttt{f}_1 \ldots, \texttt{T}'_\texttt{i}\ \texttt{f}_\texttt{i} \ldots, \texttt{T}_\texttt{n}\ \texttt{f}_\texttt{n}\}$) which may not correspond to states in the automaton representing T. In other words, these two types are constructed specifically as part of this rule and will not correspond to components of T (except by coincidence). To resolve this, we rely on rewrite rules as part of simplification (see next section) to factor such types and, thus, ensure such a rule is not required.

5 Related Work

Amadio and Cardelli were the first to show that subtype testing for recursive structural types was decidable [3]. 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. A notion of canonical form was also given, but this is quite different from our notion (in fact, it's more comparable to our *simplified form*).

Amadio and Cardelli established that subtyping in their system was decidable in exponential time. Kozen *et al.* improved 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 [11]. Gapeyev *et al.* give an excellent overview of the relationship between subtyping and coinduction [23].

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 [18] and similarly Aiken and Wimmers [2] considered recursive subtyping with union and intersection types. More recently, the XDuce system of Hosoya and Pierce for representing XML schemas [25].

One problem with this approach is that a circularity can arise between the semantic interpretation and the corresponding subtyping algorithm [21]. 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 [21, 7, 22]. 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 making it extremely powerful.

5.1 Programming Languages.

The are many examples of programming languages which support structural type systems. Strongtalk prioneered the use of structural typing for describing structures from an untyped world (i.e. SmallTalk) [10]. The essential idea is succinctly captured in the following quote (from [10]):

> *"Smalltalk is an unusually flexible and expressive language. Any type system for Smalltalk should place a high priority on preserving its essential flavor."*

The paper hints that some form of recursive structural typing was supported, although few details are given and no formalisation is included. Unfortunately, structural subtyping was subsequently dropped from Strongtalk [9], in part due to the complexity of error messages produced. Modula-3 [14] and OCaml [28] provide other well known examples of programming languages which support structural types. OCaml supports structural typing and value semantics. Furthermore, mutable structures (e.g. arrays and mutable records) can be updated in place. However, this is not done in accordance with value semantics — rather, mutable structures are allocated in the heap and passed by reference [39].

Proposals have been made to extend Java with structural types [30, 24]. Such approaches attempt to mix nominal and structural typing in an effort to gain the best of both worlds. More recently, Scala has emerged as a mainstream language which supports limited forms of structural subtyping [40]. One challenge here, is that of implementing structural types on a system designed for nominal types (i.e. the JVM) [19].

Finally, Whiley is an experimental language supporting a full structural type system [1, 38, 37]. The framework presented in this paper emerged from the author's efforts to implement Whiley's type system.

6 Conclusions

Previous work has addressed many aspects of structural types, such as efficient subtyping algorithms. Unlike nominal type systems, implementing a recursive structural type system remains a significant challenge. This is because a large gap exists between the formalisation of a recursive structural type system, and its algorithmic realisation.

In this paper, we have presented a generic framework that succinctly captures the important algorithmic issues. The framework is not tied to any particular structural type system, and can be instantiated in a variety of ways. We have illustrated our framework using a number of concrete instantiations.

References

[1] The whiley programming language, http://whiley.org.

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

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15:575–631, 1993.

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

[5] Babai and Luks. Canonical labeling of graphs. In *Proc. STOC*, pages 171–183, 1983.

[6] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM TOPLAS*, 19(1):153–187, 1997.

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

[8] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proc. OOPSLA*, pages 117–136, 2009.

[9] G. Bracha. The strongtalk type system for smalltalk.

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

[11] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proc. Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 1210 of *LNCS*, pages 63–81. Springer-Verlag, 1997.

[12] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into java. In *Proc. ECOOP*, volume 3086 of *LNCS*, pages 389–413. Springer-Verlag, 2004.

[13] L. Cardelli. Structural subtyping and the notion of power type. In *Proc. POPL*, pages 70–79. ACM Press, 1988.

[14] 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.

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

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

[17] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[18] F. M. Damm. Subtyping with union types, intersection types and recursive types. volume 789 of *LNCS*, pages 687–706. 1994.

[19] G. Dubochet and M. Odersky. Compiling structural types on the jvm: a comparison of reflective and generative techniques from scala's perspective. In *Proc. ICOOOLPS*, pages 34–41. ACM Press, 2009.

[20] J. Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs. *JCICS*, 38(3):432–444, 1998.

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

[22] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.

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

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

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

[26] J. Kobler, U. Schoning, and J. Toran. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhauser, Boston, 1993.

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

[28] 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.

[29] R. P. Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41(7):22–26, 2008.

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

[31] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proc. ESOP*, volume 5502 of *LNCS*, pages 95–111. Springer-Verlag, 2009.

[32] B. D. McKay. Practical graph isomorphism. In *NMC*, volume 30, pages 45–87, 1981.

[33] T. Miyazaki. The complexity of mckay's canonical labelling algorithm. *Groups and Computation, II*, 28:239–256, 1997.

[34] M. Nivat and A. Podelski, editors. *Tree Automata and Languages*. North-Holland, 1992.

[35] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[36] L. D. Paulson. Developers shift to dynamic programming languages. *IEEE Computer*, 40(2):12–15, 2007.

[37] D. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. In *Proc. BYTECODE*, 2011.

[38] D. J. Pearce and J. Noble. Flow-sensitive types for whiley. Technical Report ECSTR10-23, Victoria University of Wellington, 2010.

[39] D. Rémy. Using, understanding, and unraveling the OCaml language. from practice to theory and vice versa. In *Proc. APPSEM*, pages 413–536. Springer-Verlag, 2000.

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

[41] D. Spinellis. Java makes scripting languages irrelevant? *IEEE Software*, 22(3):70–71, 2005.