A Portable, Simple, Embeddable Type System

Jim Newton, Adrien Pommellet

A Portable, Simple, Embeddable Type System

Jim E. Newton
jnewton@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

Adrien Pommellet
adrien@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We present a simple type system inspired by that of Common Lisp. The type system is intended to be embedded into a host language and accepts certain fundamental types from that language as axiomatically given. The type calculus provided in the type system is capable of expressing union, intersection, and complement types, as well as membership, subtype, disjoint, and habitation (non-emptiness) checks. We present a theoretical foundation and two sample implementations, one in Clojure and one in Scala.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*.

ACM Reference Format:
Jim E. Newton and Adrien Pommellet. 2021. A Portable, Simple, Embeddable Type System. In *Proceedings of the 14th European Lisp Symposium (ELS'21).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.4709777

1 INTRODUCTION

1.1 Long-term Motivations

The motivation for this research is two-fold. In the larger sense we will lay the groundwork, so that in a future publication we will be able reason about the regularity of heterogeneous sequences [28] in programming languages which support them. By heterogeneous sequence, we mean a sequence of arbitrary, finite length for which elements are of various types such as `["Alice" 12 "Bob" 54 "Eve" -3]`. Typically, such a sequence does not contain completely random data, but rather data which follows a pattern: each element of the sequence must in turn be of a *type* determined by some sort of specification.

By *reason about the regularity* of these sequences we mean somehow specifying regular (in the sense of regular expressions) sequences of types, and to ask questions such as whether a given sequence (at run-time) belongs to this set. The theory of finite automata [20] describes how such a question can be answered in linear time, regardless of the complexity of the expression, provided that membership of the so-called *alphabet* can be determined in constant time for any element of the sequence in question. Other forms of reasoning might be to ask (at compile-time) whether an arbitrary sequence in one such set also belongs to another similarly specified set, or to compute a specifier for the intersection of such sets and ask whether that intersection is inhabited or empty. These latter questions can be answered by exploiting an algebra of operations defined on deterministic finite automata (DFA).

Novice, non-Java-savvy Scala [33, 34] users may be surprised that the pattern matcher can distinguish between `Int` and `Double` and between `List` and `Array`, but cannot distinguish between `List[Int]` and `List[Double]` as this distinction is obviated by Java type-erasure [6]. We see sequences of constant type as a special case of type patterns which may also appear in input data such as JSON [37] or as s-expression [27].

The theory of deterministic finite automata is a powerful tool for such verification: however, it can only handle sequences on a finite alphabet instead of arbitrary values from the infinite set of objects representable in a programming language. A reasonable intuition would thus be to consider a finite alphabet of *decidable types*: each type representing a potentially infinite set of values. However, these types may intersect and thus violate the *determinism* requirement if a value in the sequence belongs to two overlapping types that lead to different branches of a given algorithm. Because of their nice algebraic properties, we wish to employ deterministic finite automata (DFA), as opposed to non-deterministic (NFA). We must thus avoid such non-determinism at DFA construction time.

In a smaller sense, we want to be able to compute a proper partition of the type space (defined formally in Definition 3.1) in order to decompose *large* types into *smaller* non-overlapping types. By *compute a partition of the type space*, we mean to express the set of all objects in the programming language as a disjoint union of subsets, where each subset is expressed in terms of intersections, unions, and complements of more *fundamental* types.

We state clearly that in this article, we will not describe how to recognize sequences of heterogeneous types yet such as `Array<Int>`: this goal would be premature. In [29] we addressed a generalization of this problem, but only for Common Lisp. Our long term goal is to extend that approach to work with any programming language for which SETS has been implemented.

To do so, we need to define beforehand a simple embeddable type system (SETS) which can be used in such a recognition flow. This type system is intended to be simple, and not intended to replace that of Common Lisp, nor is it intended to be powerful enough to implement the Common Lisp type system. The type system encapsulates just enough of the Common Lisp type system needed to implement the regular sequence recognition flow as described in [28] to other languages.

1.2 Goals and Contributions

We will therefore present in this article the first necessary step to address the goal of type-based sequence recognition. We introduce a simple type system (SETS) that defines *fundamental* types, and

ELS'21, May 03–04 2021, Online, Everywhere

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.4709777

which makes it possible to compute a partition of a type space using a given set of types and set-theoretic operations ($\cup, \cap, \overline{\ }, \in, \subset$). This simple type system will be used as a building block for later development in a future publication, but deserves careful consideration in its own right.

Our previous work addressed this problem for Common Lisp, as described in [28]; we begin in this article our goal of generalizing the method to a larger class of languages. We demonstrate the SETS type system by implementing it in three programming languages.

- *Clojure* [18, 19] was chosen because it is a lisp, and we hoped that it would provide a logical transition for a theory based in Common Lisp.
- *Scala* runs on the same JVM [25] as does Clojure, thus allows similar sequences and type reflection operations despite syntactic differences.
- Python [48] is a dynamic language that makes different assumptions about types than do JVM based languages.

All three of these languages support heterogeneous sequences, and provide a type system with reflection. The implementation in Python is in its infancy, and the details of its implementation are discussed sparingly in this article.

The current article lays a foundation by defining a portable, Simple, Embeddable Type System (SETS). The principles of SETS, defined formally in Section 3.2, are inspired by the type system of Common Lisp, which we outline in Section 2.4. We have identified some limitations of the Common Lisp type system in Section 4.5 and propose solutions in SETS. We then present two sample implementations of SETS, one in Clojure and one in Scala in Sections 4.2 and 4.3 respectively.

2 PREVIOUS WORK

2.1 Type-based Sequence Recognition

In [28] we discussed the flow for developing type based sequence recognition in the explicit context of Common Lisp [2], as Common Lisp provides a built-in type system capable of expression Boolean combinations of types of arbitrary complexity. We presented several algorithms for computing Maximum Disjoint Type Decomposition (MDTD) [30], but those algorithms make heavy use of the assumption that the underlying type system is that of Common Lisp.

A notable implementation detail of rational type expressions in Common Lisp [29] is that the type definition makes reference to the `satisfies` type. This use of `satisfies` means that in many cases, `subtypep` cannot reason about the type. In the current work, we define the building blocks of a type system which can be extended (without resorting to `satisfies`) in a future publication which will be able to more easily reason about such types.

2.2 Types as Boolean Combinations

Castagna *et al.* [7] argue that many programmers are drawn to the flexibility and development speed of dynamically typed languages, but that even in such languages, compilers may infer types from the program if certain language constructs exist. The authors argue that adding set theoretical operations (union, intersection, complement) to a type system facilitates transition from dynamic typing to static typing while giving even more control to the programmer.

Researchers in early programming languages experimented with set semantics for types. Dunfield [13] discusses work related to intersection types, but omits acknowledging the Common Lisp type system. He mentions the Forsythe [39] language from the late 1980s, which provides intersection types, but not union types. Dunfield claims that intersections were first developed in 1981 by Coppo *et al.* [9], and in 1980 by Pottinger [38] who acknowledges discussions with Coppo and refers to Pottinger's paper as forthcoming. Work on union types began later, in 1984 by MacQueen *et al.* [26].

Languages which support intersection and union types [7, 8, 36], should also be consistent with respect to subtype relations. Frisch *et al.* [15] referred to this concept as semantic subtyping. In particular, the union of types A and B should be a supertype of both A and B, the intersection of types A and B should be a subtype of both A and B, and if A is a subtype of B, then the complement of B should be a subtype of the complement of A. While Coppo and Dezani [10] discuss intersection types in 1980, according to a private conversation with one of the authors, Mariangiola Dezani, theoretical work on negation types originates in Castagna's semantic subtyping.

The theory of intersection types seems to have influenced modern programming language extensions, at least in Java [16] and Scala. Bettini *et al.* [4] (including Dezani, mentioned above) discuss the connection of Java λ-expressions to intersection types.

The standard library of Scala language [33, 34] supports a type called `Either`. This type serves some of the purpose of a union type. `Either[A,B]` is a composed type, but has no subtype relation neither to A nor to B. `Either` lacks many of the mathematical properties of union; it is neither associative $Either[Either[A, B], C] \neq Either[A, Either[B, C]]$, nor commutative $Either[A, B] \neq Either[B, A]$. Sabin [44] discusses user level extensions to the Scala type system to support intersection and union types. Yet, Sabin [43] recommends that no one use his implementation in *real code*. Doeraene [12, 22] introduces pseudo-union types in Scala.js. Scala 3 [1, 31, 42] supports intersection and union types in a type lattice, but not complementary types. However, according to the Scala 3 specification[1], the type inferencer may not consider union types while computing the least upper bound of two types.

2.3 Difficulties with the Subtype Relation

Grigore [17] addresses subtype decidability in Java [16]: deciding whether one type is a subtype of another is equivalent to the halting problem. Kennedy[2] and Pierce [24] identify several restrictions which when imposed, make the question decidable. They investigate the effect of restricting these capabilities in different ways in Java [16], Scala [32, 33], C#, and .NET Intermediate Language.

In Scala, the operator `<:<` attempts to compute the subtype relation, returning *true* if this relation can be proven, else *false*. Thus, the Scala `<:<` conflates *cannot compute* with *computed to be false*. The method `isAssignableFrom` defined on `java.lang.Class` is also available to the Scala programmer. This (decidable) method

[1] See https://dotty.epfl.ch/docs/reference/new-types/union-types.html for a description of the behavior of Union Types in Scala 3

[2] Curiously, Kennedy [24] from Microsoft Research gives citations for Java and Scala, but not for C# and .NET Intermediate Language. Perhaps authors believe the reader is more familiar with C# and .NET IL than with other languages.

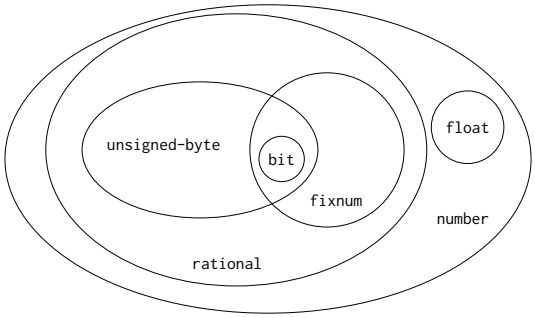

Figure 1: Example Common Lisp types with intersection and subset relations

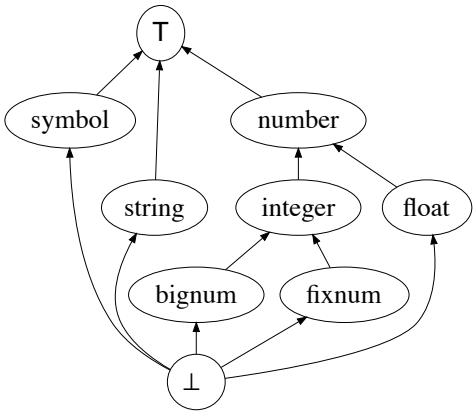

Figure 2: Example hierarchy of hosted types. Arrows point from subtype to supertype.

computes the subtype relation, returning *true* if the subtype relation is provided by the class hierarchy, and *false* otherwise.

Bonnaire [5] presents *typed Clojure*, but does not rigorously define a Clojure *type*, and does not implement anything as ambitious as Common Lisp's `subtypep`. The Clojure [18, 19] language offers an `isa?` predicate, which is simple and decidable but is ultimately based on `isAssignableFrom` which is provided by the JVM [16].

In Common Lisp [2] `subtypep` can be computationally intensive in general. Even in cases where the subtype relation is decidable, a Common Lisp implementation is allowed to return *don't know* in cases deemed to be *too computationally intensive*. [3, 47]

2.4 Types in Common Lisp

A detailed discussion of Common Lisp types can be found in the Common Lisp specification [2, Section 4.2.3]. A summary thereof emphasizing peculiarities of function types can be found in [28]. We present here a shorter version of that exposé, emphasizing features potentially portable to other programming languages.

In Common Lisp, a *type* is a (possibly infinite) set of objects at a particular point of time during the execution of a program [2]. As illustrated in Figure 1, Common Lisp programmers may take many ideas about types from the intuition they already have from set algebra. An object can belong to more than one type. Two given types may intersect, such as the case of `unsigned-byte` and `fixnum` in the figure, (unsigned-byte ∩ fixnum) ≠ ∅. Types may be disjoint such as `float` and `fixnum`, (float ∩ fixnum) = ∅. Types may have a subtype relation such as `fixnum` and `number`, (fixnum ⊂ number) or more complicated relations such as (bit ⊂ (fixnum ∩ unsigned-byte) ⊂ rational).

Types are never explicitly represented as objects by Common Lisp. Instead, a type is referred to indirectly by so-called *type specifiers*, which designate types. The symbol `string` and the list `(and number (not bit))` are type specifiers [2]. Since types are not objects specified in the language, but rather designated by specifiers, it becomes challenging to reason about types. *I.e.*, programmers must construct algorithms to determine whether designated types are inhabited *vs.* empty, and to efficiently represent types designated by composing type specifiers in various ways.

Two important Common Lisp functions pertaining to types are `typep` and `subtypep`. The function `typep`, a set membership test, is used to determine whether a given object is of a given type. Type specifiers indicating compositional types are often used on their own, such as in the expression `(typep x '(or string (eql 42)))`, which evaluates to `true` either if `x` is a string, or is the integer 42. The function `subtypep`, a subset test, is used to determine whether a given type is a *recognizable* subtype of another given type. The function call `(subtypep T1 T2)` distinguishes three cases: (1) that `T1` is a subtype of `T2`, (2) that `T1` is not a subtype of `T2`, or (3) that the subtype relationship cannot be determined. Newton [28] and Valais [46] discuss scenarios which fall into this third case.

3 PORTABLE AND EMBEDDABLE

Here we present a type system which exhibits some notable features of the Common Lisp type system, in particular, union, intersection, and complement types, type membership and subtype predicates. The type system we present is intended to be implementable in other languages, and should allow run-time code to reason about sets of values. We present our assumptions with respect to the host language in Section 3.1, an abstract definition of SETS in Section 3.2, explain some subtleties of the subset relation in Section 3.4, then discuss different subset algorithms in Section 3.5. Sections 4.2 and 4.3 present sample implementations in Clojure and Scala respectively.

3.1 Hosted Types

We are interested in embedding a simple type system into a host language. We will assume that said host language provides a set of *fundamental* types which we will accept as given as well as a distinct set Υ_0 of designators for those fundamental types. Just as Common Lisp distinguishes between types and type specifiers, we will also distinguish between the types (sets of values) and their designators (syntactical elements). *E.g.* consider $\Upsilon_0 = \{symbol, string, number, integer, float, bignum, fixnum\}$ as shown in Figure 2. See Section 4.1 for a discussion of parameterized types such as `List<Int>`.

We further assume that the host language provides membership and subset relations with respect to Υ_0. The membership relation must be decidable and properly implemented in the host programming language. For example, we must be able to determine that `"hello world"` $\in$ *string* and that $1.2 \notin$ *integer*.

We also require that the programming language provide a subtype query mechanism with respect to Υ_0. For example, in Figure 2, we see that *fixnum* $\subset$ *integer* $\subset$ *number* and that *string* $\not\subset$ *integer*. Even if the subtype relation is sometimes undecidable, we intend to exploit the cases when it is decidable. See Section 2.3 for further discussion of the decidability of the subtype relation.

In Common Lisp, the `typep` and `subtypep` functions are built-in, and symbols implement type designators (called type specifiers in Common Lisp) for built-in types.

In Clojure, the functions `instance?` and `isa?` correspond respectively to the Common Lisp functions `typep` and `subtypep` as far as low level types are concerned. In Python, the functions are `isinstance` and `issubclass`.

In Scala, a class is an object that belongs to a sort of meta-class, `java.lang.Class`, and that class has methods named `isInstance` and `isAssignableFrom`. We have access to the (meta-)class object via that `classOf[]` syntax.

```
// membership test in Scala
classOf[java.lang.String]
  .isInstance("hello world")
// subtype test in Scala
classOf[java.lang.Number]
  .isAssignableFrom(classOf[java.lang.Integer])
```

3.2 Formal definition of SETS

Different programming languages make different assumptions about types. We assume a type system in accord with Definition 3.2.

Definition 3.1 (type, type space). Let Σ denote the set of all values expressible in a given programming language. A *type space* is the set 2^Σ of subsets of Σ. Each subset of Σ is called a *type*.

Definition 3.2 (simple embedded type system). A *simple embedded type system* (SETS) consists in a set Υ of *type designators*, two operators $\in$ and $\subset$, and a *type designation* function $[\![\,]\!] : \Upsilon \to 2^\Sigma$. Intuitively, each $v \in \Upsilon$ designates a type which is the set $[\![v]\!]$. Formally, the set Υ and the function $[\![\,]\!]$ are defined inductively, using the following atomic elements:

- **Hosted types**: let Υ_0 be a set of *native type designators* of the host language, such that to each $v \in \Upsilon_0$, a *native type* σ_v can be matched. Then $\Upsilon_0 \subset \Upsilon$ and $\forall v \in \Upsilon_0$, $[\![v]\!] = \sigma_v$.[3]
- **Universal type**: there is a symbol $\top \in \Upsilon$, and $[\![\top]\!] = \Sigma$.
- **Empty type**: there is a symbol $\bot \in \Upsilon$, and $[\![\bot]\!] = \emptyset$.
- **Singletons**: $\forall a \in \Sigma$, there is a matching element $\{a\} \in \Upsilon$, and $[\![\{a\}]\!] = \{a\}$.
- **Predicates**: for any decidable function f mapping Σ to $\{true, false\}$, there is a symbol $\mathfrak{Sat}_f \in \Upsilon$, and $[\![\mathfrak{Sat}_f]\!] = \{x \in \Sigma \mid f(x) = true\}$.

And the following constructors:

- **Union**: if $v_1, v_2 \in \Upsilon$, then $v_1 \cup v_2 \in \Upsilon$ and $[\![v_1 \cup v_2]\!] = [\![v_1]\!] \cup [\![v_2]\!]$.
- **Intersection**: if $v_1, v_2 \in \Upsilon$, then $v_1 \cap v_2 \in \Upsilon$ and $[\![v_1 \cap v_2]\!] = [\![v_1]\!] \cap [\![v_2]\!]$.
- **Complement**: if $v \in \Upsilon$, then $\overline{v} \in \Upsilon$ and $[\![\overline{v}]\!] = \Sigma \setminus [\![v]\!]$.

Moreover, the relations $\in$ and $\subset$ on Υ should verify the following properties:

(1) The operator $\in$ defines a *decidable* membership relation between values and type designators. Formally, given $v \in \Upsilon$ and $a \in \Sigma$, $a \in v$ holds if only if $a \in [\![v]\!]$, and $a \in \overline{v}$ if and only if $a \notin [\![v]\!]$.
(2) There is a partial subset relation $\subset$ on types. Given $v_1, v_2 \in \Upsilon$, $[\![v_1]\!] \subset [\![v_2]\!]$ if and only if $v_1 \subset v_2$. However, this subset relation may be *undecidable*. See Sections 2.3 and 3.4. △

The $\mathfrak{Sat}_f$ type is capable of defining many of the same sets definable by other types in SETS by clever enough choices of f. However, doing so will limit reasoning power. For example, given two predicates f and g it is not possible in general to determine whether $\mathfrak{Sat}_f \subset \mathfrak{Sat}_g$ or whether $\overline{\mathfrak{Sat}_f} \cap \overline{\mathfrak{Sat}_g}$ is inhabited.

We do not specify how the programming language represents type designators: possibly as part of the language specification [45, Chapter 4] as in Common Lisp, or as an Algebraic Data Type (ADT) [21] implemented as an add–on library.

3.3 Relating SETS to Common Lisp

SETS as described in Section 3.2 is based on the type system of Common Lisp but simplified, and omitting some features which make the subtype relation difficult to compute. For example, we omit interval (such as `(float -1.0 (2.0))`), `mod`, and `member` types as well as composed forms such as `(vector double-float 100)`. Whereas Common Lisp defines a `member` type, SETS represents such types as a finite union of singleton types.

The hosted types of SETS correspond to the types designated by the atomic forms of the type specifiers mentioned Figure 2-4 of Section 4.2.3 in the Common Lisp specification [2] as well symbolic names of Common Lisp classes, conditions, and structures. List forms of types mentioned in Figure 4-3 of Common Lisp specification Section 4.2.3 are omitted in SETS, such as `(cons float)`, `(function (t))`, and `(simple-array * 4)`.

The universal and empty types of SETS are respectively designated by `t` and `nil` in Common Lisp. Singleton types in SETS are represented by `eql` in Common Lisp. Predicates in SETS are represented by `satisfies` in Common Lisp. Union, intersection, and complement in SETS are represented respectively by `or`, `and`, and `not` in Common Lisp.

The $\in$ operation in SETS is implemented by the Common Lisp `typep` function. The $\subset$ operation in SETS is implemented by the Common Lisp `subtypep` function, which returns two values. A second such value of `nil` means the subtype relation undecidable; otherwise a first value of `true` or `false` correspond to the decidable true and false.

3.4 A Challenging Subtype Relation

Note that the subtype relation is the atomic function for detecting intersection, disjointness, and emptiness [3, Sec 6]. We need these

[3] Recall from Section 3.1 that we axiomatically assume that this makes sense.

checks in order to minimize a partition of the type space. Although we use functions such as `disjoint?` and `inhabited?` in our code, these are merely semantic wrappers around `subtypep?`, at least from the API perspective, if not from the internal implementation perspective.

However, the subset relation may not always be *decidable*. Thus, the decision function in SETS should be allowed to return *dont-know* in a manner appropriate to the host language, although we would like it to be as decisive as possible. We discuss here three scenarios when such an *indecision* might occur.

The most obvious situation involves $\mathfrak{Sat}$. Given two arbitrary decidable predicates f and g, it is not possible to know whether $\mathfrak{Sat}_f \subset \mathfrak{Sat}_g$. This is a direct consequence of not being able to decide whether a given recursive language is a subset of another.

A second scenario involves a hidden empty set. Consider disjoint sets A and B. $A \cap C \not\subset B$ if and only if $A \cap C \neq \emptyset$. However, it may be impossible to decide whether $A \cap C$ is empty, as it is undecidable to determine whether a given recursive language is empty.

Consider a third scenario featuring two types A and B that are disjoint such as the number types `Double` and `Long` in the JVM. We wish to confirm that $Long \subset \overline{Double}$. Note that $A \subset \overline{B}$ if $A \cap B = \emptyset$ and $\overline{A} \neq B$. If we want to determine whether $A \subset \overline{B}$, testing whether $A = \overline{B}$ would be a mistake: we would have to test both $A \subset \overline{B}$ and $\overline{B} \subset A$, thus introducing an infinite loop in our subtype procedure.

We consider the subtype procedure inaccurate if it returns *dont-know* in a decidable case. We use the term *inaccurate* deliberately after considering alternative terminology. The only way we have of knowing whether the subtype relation is satisfied is to determine so (or not so) algorithmically. Given two algorithms, X and Y, if X determines that $A \subset B$ or that $A \not\subset B$, but Y returns *dont-know* then we consider Y less accurate then X in this regard. Thus, in some cases, it is algorithmically impossible to distinguish between an *inaccurate* answer and an *undecidable* problem.

Our current solution to the question of whether $Long \subset \overline{Double}$ is to include an admittedly inaccurate special case in the subtype procedure which checks whether it is dealing with hosted types and their complements. We may then use known properties of hosted types, as in Algorithm 1 of Section 4.4, to determine that $\overline{Double} \not\subset Long$. We may however not be able to directly handle relations involving more complex types such as $\overline{\{0\}} \subset \{1\} \cup \overline{\{0\}}$.

Undetermined cases can cause a cascade effect in increasing the complexity of a procedure. Indeed, consider two types A and B. Their standard partition is defined as $\{A \cap B, A \cap \overline{B}, \overline{A} \cap B, \overline{A} \cap \overline{B}\} \setminus \{\emptyset\}$. The automata-theoretic approach, alluded to in Section 5.2, heavily relies on such partitions. The maximum size of this partition is 4, but various subtype relations may decrease it: as an example, if $A \subset B$ then $A \cap \overline{B} = \emptyset$. Given that the size of standard partitions can grow exponentially with the number of types involved, it is of the utmost importance to keep it as small as possible. Thus an inaccuracy in the algorithm may cause unnecessary performance problems when constructing finite automata.

3.5 Computing the Subtype Relation

The specification of SETS does not impose any particular algorithm. We discuss here three known approaches which have been used to compute `subtypep` in Common Lisp: (1) Baker's algorithm, (2) Binary Decision Diagrams, and (3) Symbolic Boolean manipulation.

Baker's [3] algorithm (approach 1) was presented anew by Valais [47] in attempt to make it more understandable. At a high level, the algorithm attempts to represent every type designator by a bit-mask, allowing union, intersection, and complement to be computed by Boolean operations of bit-vectors. Equivalent types have the identical bit-masks; the empty type has an all zero bit-mask. Baker's algorithm decides whether A is a subtype of X by computing the bit-wise `AND` of a and $\overline{x}$ given a bit-vector a of A and a bit-vector x of X. If the result is 0, then $A \subset X$. The ECL [41] Common Lisp compiler uses a variant of Baker's algorithm.

We [30] presented Binary Decision Diagrams (BDDs) (approach 2) as computation tools for manipulating Common Lisp type specifiers. BDDs are a powerful tool for manipulating Boolean functions. A BDD can represent a type specifier, and Boolean operations between BDDs compute canonical forms for union, intersection, and complement types.

Approach 3 employs a symbolic Boolean manipulation approach. A type designator represents an abstract syntax tree (AST) for a Boolean expression. Computing the subtype relation involves checking a long but inexhaustive series of conditions, some necessary, some sufficient, and some both necessary and sufficient. If a necessary condition fails, `false` is returned. If a sufficient condition succeeds, `true` is returned. If all these conditions fail to decide the result, then `dont-know` is returned. The SBCL[40] Common Lisp compiler's implementation of `subtypep` relies on a similar yet augmented procedure [23].

The sample implementations of SETS presented in Section 4 rely on the third approach. In the Scala implementation (Section 4.3) of SETS, dynamic dispatch is used on the class of the type designator object to direct the computation away from tests known to be irrelevant. In the Clojure implementation (Section 4.2), type designators are represented by s-expressions, which are examined by an ad-hoc pattern matcher which determines according to the first element (`and`, `or`, `not`, etc) which irrelevant checks it should eliminate. Examples of such conditions are:

- *necessary and sufficient*: $\overline{A} \subset \overline{X}$ if and only if $X \subset A$.
- *sufficient*: $X \subset A \cup B$ if $X \subset A$ or $X \subset B$.
- *sufficient*: $A \cap B \subset X$ if $A \subset X$ or $B \subset X$.
- *necessary*: $X \subset A \cap B$ only if $X \subset A$ and $X \subset B$.

We must also take into account the computational complexity of the solution. Our representation of a type designator is an expression tree. Some operations on such a tree have exponential complexity in terms of the height of the tree. It is possible to minimize the depth of the tree by converting the expressions to DNF (disjunctive normal form) or CNF (conjunctive normal form), but the conversion itself has an exponential worst case complexity. Yet, conversion to a normal form can significantly increase the accuracy of the decision procedures, as we will demonstrate in Section 5.1.

4 IMPLEMENTATIONS OF SETS

Three implementations of SETS are currently available, one in Clojure as described in Section 4.2, a second in Scala as described in Section 4.3, and a third which is currently under development in Python. The Python implementation (called Genus) is publicly available at https://gitlab.lrde.epita.fr/jnewton/python-rte, but we do not discuss specific details thereof in this article.

4.1 Java Subclass vs. SETS Subtype

In order to avoid confusion later, we clarify here that types in SETS are not the same as what a Java programmer might otherwise think of as a type, nor the Scala or Clojure programmer.

The Clojure and Scala implementations of SETS both fundamentally manipulate the same low-level objects within the JVM, albeit the derived classes of objects expressible in each language are different. Both at the Scala level and the Clojure level, no substantial difference is made between classes and interfaces—a fact which might be unfamiliar to the Java programmer. The Scala and Clojure implementations of SETS treat interfaces and classes identically in the sense that a type (in the SETS sense) is a set of objects which share a particular `java.lang.Class` in a particular list of super classes.

If C_C is a class in Java, then SETS associates with C_C the set of all (Scala or Clojure) objects whose `java.lang.Class` contains C_C in its list of super classes. Similarly, if C_I is an interface in Java, then SETS associates with C_I the set of all objects whose `java.lang.Class` contains C_I in its list of super classes. In this sense $C_I \subset$ *Object* in SETS even though a Java programmer might insist that C_I is not a subclass of *Object*. That is to say, when we speak of subtype-ness in SETS, we are strictly referring to whether one set of objects is a subset of another set, not whether Java considers the classes as subtypes.

In order to avoid confusion, we state explicitly that we have not extensively experimented with parameterized Java types such as `List<String>`. Java type-erasure makes some such introspection impossible, but we do not know to what extent (if at all) it is possible to query such types at run-time via the Java reflection API. Furthermore, we do not know to what extent a user of Genus can confuse the system by using such parameterized types.

4.2 Genus, SETS in Clojure

The Clojure implementation of SETS is called Genus. The documentation is available publicly, at https://gitlab.lrde.epita.fr/jnewton/clojure-rte/-/blob/master/doc/genus.md. The source code is available as part of a larger project called `clojure-rte` at https://gitlab.lrde.epita.fr/jnewton/clojure-rte. Readers familiar with the Common Lisp type system will find Clojure Genus to be intuitive, and will recognize that we are, in user space, imposing a simplified version of the Common Lisp type system onto Clojure.

4.2.1 How Clojure-Genus is used. The API of Genus consists of an implicit data structure which implements the type designators, following the pattern described in Definition 3.2. In Clojure Genus, a type designator is one of the following:

- **Hosted types:** A symbol, *s* for which `(resolve s)` has type `java.lang.Class`. *E.g.*, `String` or `clojure.lang.Symbol`.
- **Universal**: `:sigma`.
- **Empty**: `:empty-set`.
- **Singletons**: A list of the form `(= x)`. *E.g.*, `(= 0)`.
- **Predicates**: A list of the form `(satisfies f)`. `f` is a symbol resolving to a function. *E.g.*, `(satisfies odd?)`.
- **Union**: A list of the form `(or ...)`. Operands are type designators. *E.g.*, `(or String (satisfies keyword?))`.
- **Intersection**: A list of the form `(and ...)`. Operands are type designators. *E.g.*, `(and Long (satisfies pos?))`.
- **Complement**: A list of the form `(not x)`. Operand is a type designator. *E.g.*, `(not (or Long String))`.

As required by Definition 3.2, two functions `typep` and `subtype?` on type designators implement the $\in$ and $\subset$ operators. The `typep` (binary) function can be used as in `(typep "hello" '(or String clojure.lang.Symbol))`, and returns `true` or `false`.

The `subtype?` function is more complicated; it accepts two arguments A and B, and returns one of `true`, `false`, or `:dont-know`. If `subtype?` proves that $A \subset B$, it returns `true`; if it proves that $A \not\subset B$, it returns `false`. Otherwise, it returns `:dont-know`.

In addition to `subtype?` two other functions are provided for reasoning about types via type designators:

- `disjoint?`, returns `true`, `false`, or `:dont-know` indicating whether two types are disjoint, *i.e.* their intersection is empty.
- `inhabited?` returns `true`, `false`, or `:dont-know` indicating whether the type is not empty.

The `disjoint?` and `inhabited?` functions are not required by the SETS specification. We have implemented them here to facilitate the implementation of the subtype predicate. In fact, given a fully functional, self-contained implementation of `subtypep?`, `disjoint?` and `inhabited?`, if needed, could be implemented in terms of `subtype?`. Two types are disjoint if their intersection is a subtype of the empty type. A type is inhabited if it is not a subtype of the empty type. Special care would need to be taken for the case that `subtype?` returns `dont-know`.

4.2.2 How Clojure-Genus is implemented. The implementation of the public API (including the functions `typep`, `subtype?`, `disjoint?`, and `inhabited?`) is around 1600 lines of Clojure code. Space does not permit a full explanation of all the code. We invite the reader to peruse the code. We will summarize some details of the functions `typep` and `disjoint?`.

The `typep` function is implemented as a multi-method, where each method implements the type membership decision for a particular type designator. Two such methods are shown here.

```
1 (defmethod typep 'not [a-value [_a-type t]]
2   (not (typep a-value t)))
3
4 (defmethod typep 'and [a-value [_a-type & others]]
5   (every? (fn [t1] (typep a-value t1)) others))
```

Whereas methods implement `typep` directly, other functions are implemented with a *shadow* function; e.g., `-subtype?`, `-disjoint`, and `-inhabited`. These are not methods in the normal Clojure sense, but rather are called according to a Common Lisp-like method-combination [35]. The method-combination calls each of the methods in turn. Each method attempts to prove sufficient

conditions or disprove necessary conditions as explained in Section 3.5 and thus return `true`, `false`, or `:dont-know`. The method-combination continues to call the methods until one is conclusive, returning `:dont-know` as a last resort. The `disjoint?` function is an exception. Its method-combination assures that `(disjoint A B)` returns the first conclusive value of either `(-disjoint? A B)` or `(-disjoint B A)`, or `:dont-know` as a last resort.

```
(defmethod -disjoint? 'or [t1 t2]
  (cond (not (gns/or? t1))
        :dont-know
        ;; sufficient
        (every? (fn [t1'] (disjoint? t1' t2 false))
                (rest t1))
        true
        ;; necessary
        (some (fn [t1'] (not (disjoint? t1' t2 true)))
              (rest t1))
        false
        :else :dont-know))

(defmethod -disjoint? :subtype [sub super]
  (cond (and (subtype? sub super false)
             (inhabited? sub false))
        false
        :else :dont-know))
```

In the `-disjoint? 'or` method above, the code asks whether the first argument called is a type designator of the form `(or ...)`. If it is not of this form, then `:dont-know` is returned. Otherwise other necessary and sufficient conditions are tested. If they are inconclusive, `:dont-know` is returned, indicating to the caller to continue with other methods of the method combination.

Two tests are made (in the 2nd and 3rd clause of the `cond`). First, a sufficient condition examines each of operand `(or ...)` to detect whether all of them designate a type which is disjoint from `t2`; if so, then the types are deemed to be disjoint. Second, a necessary condition examines each operand of `(or ...)` to detect whether at least one of them is definitely *NOT* disjoint from `t2`; if such a type is found, then we conclude that that `t1` and `t2` are not disjoint. Finally, if the tests were inconclusive we return `:dont-know`.

The `-disjoint :subtype` method shown above checks an expensive but general condition. The method attempts to detect whether the two designated types are in a subtype relation. If $A \subset B$ then the types are not disjoint, except in the case that $A = \emptyset$. Since $\emptyset$ is disjoint from every type, including itself, we must assure that the type in question is inhabited, via a call to the `inhabited?` method.

We show the implementation of this method to partially justify the reason we have included the `inhabited?` method in our protocol. During our implementation of Clojure Genus we found the existence of `inhabited?` simplifies the logic of certain subtype and disjoint decisions, even if not strictly necessary. In Common Lisp `(not (subtypep A nil))` serves a similar purpose, understanding the caveat that if Common Lisp `(subtypep A nil)` returns a second value of `nil` then `(not (subtypep A nil))` still evaluates to *true*, even though `A` may not really be inhabited.

The method-combination assures that once the method above with dispatch value `'or` returns `true` or `false`, then the method whose dispatch value is `:subtype` is averted.

4.3 Genus, SETS in Scala

The Scala Genus implementation is about 1500 lines of Scala code—roughly the same size as the Clojure version. The code is available publicly at https://gitlab.lrde.epita.fr/jnewton/regular-type-expression/-/tree/master/cl-robdd-scala/src/main/scala/genus.

In Scala Genus a type designator is implemented as an abstract class named `SimpleTypeD` (simple-type-designator), and several leaf classes which extend the abstract class. There is one leaf class per type designator, *e.g.* the case class `SAnd` defines the type designator implementing intersection, and analogously `SOr` for union. There are also two singleton objects `STop` and `SEmpty` representing the top and bottom types in the type lattice. The class `SAtomic` serves to implement the hosted types, interfacing the Genus system with the class system of the JVM. Finally, the classes `SEql` (singletons), `SCustom` (predicates), `SNot` (complements) complete the list of cases required in Definition 3.2.

The class, `SimpleTypeD`, declares methods named `typep`, `disjoint`, `inhabited`, and `subtypep`, and each subclass overrides the methods in these protocols to implement case specific logic. The method `typep` returns type `Boolean`; however, whereas the corresponding methods in Clojure return `true`/`false`/`:dont-know`, these Scala methods return `Option[Boolean]` with `Some(true)` and `Some(false)`, and `None`. That these methods return an `Option` as opposed to a `Boolean` alleviates the temptation to the programmer of treating the return value as a Boolean, as further discussed in Section 4.5.

We omit more details of the Scala implementation in this article, and invite the curious reader to download the Scala code.

4.4 Disjoint decision on JVM hosted classes.

In Section 3.1 we explained how the Clojure function `isa?` and the Scala method `isAssignableFrom` are used to determine the subtype relation for JVM classes which are the classes we are hosting in our simple, embedded type system.

We wish now to discuss how the disjoint decision is computed. Note that we are not trying to ask whether the two classes have a common superclass. The question, rather, is about the intersection of two sets of objects: all the objects of class c_1 and all the objects of class c_2. We ask whether it is provable that those sets have no common elements. A simple example is that if A and B are two distinct *final* classes. Then there is no object which is a member of both classes simultaneously. I.e., the set of objects of class A is disjoint from the set of objects of class B. The question is trickier when considering interfaces, abstract classes, and other classes.

Some readers may object to the fact that we refer to Java interfaces and Java classes both as classes. In fact, within the JVM both interfaces and classes are themselves objects whose type is `java.lang.Class`, thus we unapologetically refer to them as classes. Various *flavors* of Java classes may be distinguished using the Java reflection API which is available both from Clojure and also Scala. The Clojure interface is a bit easier so we start there. The Clojure `clojure.reflect` library provides a function

`type-reflect` which returns a data structure (a map) with a `:flags` field. We use the value of this field to distinguish between four cases: `:interface`, `:final`, `:abstract`, and `:public`. The code is shown here.

```
1 (defn class-primary-flag [t]
2   (let [r (refl/type-reflect (find-class t))]
3     (cond (contains? (:flags r) :interface) :interface
4           (contains? (:flags r) :final)     :final
5           (contains? (:flags r) :abstract)  :abstract
6           :else                             :public)
```

In Scala Genus, we have implemented the same capability using JVM methods directly. The Java library `java.lang.reflect.Modifier` is available to the Scala programmer and provides methods `Modifier.isFinal`, and `Modifier.isInterface`. These methods are sufficient as the disjoint decision treats abstract and public classes identically.

Since we can detect whether a JVM class is an interface, final, or abstract/public class, the disjoint decision follows Algorithm 1.

In Algorithm 1 we check on line 1 whether either class is a subclass of the other, including whether the two classes are equal (a potentially optimizable special case). If so, every object of one type is in the other, so they are not disjoint.

The astute reader will recognize a loophole which line 1 ignores. We are implicitly supposing the types are inhabited. If a given Java class designates a vacuous type, our assumption will be violated. An empty type is disjoint from every type, including itself. More research is needed to accommodate Java classes for which they nor any subclass thereof can be instantiated.

On line 3 we ask whether either is final, encompassing two cases: exactly one is final, or both are final. If exactly one is final, then we've already determined on line 1 that the final class does not inherit from the non-final class. So in this case they are disjoint as no object can be of both classes. Second case is if both classes are final, which by definition designate disjoint sets.

On line 5 we ask whether either is an interface, knowing that neither is final. Thus we have an interface and some other non-final class. Here we chose to return *false*, because we cannot prove that no class inherits from both of these. If both classes are interfaces, then it is possible to create another class (abstract or final) which extends both of them. If one, say I, is an interface, and the other, say C, is public or abstract, then again is it possible declare a third class which inherits from C and implements interface I.

Note that the decision at line 5 to return *false* is one which determines the semantics of our system. It might happen be true that there exists no class which includes both classes in its lineage list. If this is the case, then indeed the two designated sets are in fact disjoint. However, since new classes can be loaded at run-time, and we cannot predict the future, we chose to return *false* saying that there are cases when the two sets intersect, thus are non-disjoint. The semantics of our system are currently undefined if classes are modified at runtime or if an object is mutated in a way which changes its type. See Section 5.2 for a discussion of defining these system semantics differently.

Finally, on line 7, we are in a position that neither class is final, neither is an interface, and neither is a subclass of the other; *i.e.*, they are either abstract classes or otherwise public classes in separate hierarchies which cannot be further mixed using class inheritance. Therefore we return *true* as they designate disjoint sets.

Algorithm 1: Compute the disjoint relation between JVM classes

Input: c_1, c_2 : two JVM classes
Output: Boolean indicating whether the classes are disjoint

1 **if** $c_1 \subset c_2$ *or* $c_2 \subset c_1$ **then**
2 | **return** *false*
3 **else if** c_1 *is final or* c_2 *is final* **then**
4 | **return** *true*
5 **else if** c_1 *is interface or* c_2 *is interface* **then**
6 | **return** *false*
7 **else**
8 | **return** *true*

4.5 Perceived Limitations of CL Type System

SETS addresses some of our perceived limitations of the Common Lisp type system. In the Genus implementations of SETS we have separated the subtype predicate from the inhabited and disjoint predicates. This distinction is not strictly necessary, as also discussed in Section 4.2.1; nevertheless we found the separation simplifies some of the logic and facilitates unit testing in some cases. For some types, such as hosted types, it is straightforward to define disjointness and habitation and to define subtype in terms of those.

In Common Lisp, the return value of `(subtypep (and string A) number)` depends on whether `A` designates an inhabited type. If `A` is defined using `satisfies`, then `(subtypep (and string A) number)` is apt to return `dont-know`. Genus allows the definition of a new type to specify the behaviors of the subtype, inhabited, and disjoint predicates with respect that that new type. In portable Common Lisp, user defined types cannot be integrated into `subtypep`. There is simply no facility for doing so. In Genus, subtype is defined in terms of extensible code allowing applications defining a type to specify the behavior with respect to the new type.

Another problem (feature) of `cl:subtypep` is that it can accidentally be used as a predicate. The user may make believe that it returns `true` or `false`. But in reality such usage is prone to error as the usage conflates `dont-know` with `false`. When `(subtype A B)` returns `false`, the caller should not assume that `A` is not a subtype of `B`. Nevertheless, this mistake is exceedingly easy to make.

Scala Genus defines `subtypep` with return type `Option[Boolean]`, rather than `Boolean`. Although this may sometimes be an annoyance, it forces in the programmers to remember that `subtypep` is not a 2-way predicate. Clojure Genus, admittedly, does fall prey to this potential delusion. In Clojure, `subtype?` takes three arguments, the third of which is the value to return in the `dont-know` case, and is a gentle reminder to the programmer of the 3-way nature of the function. Nevertheless, the programmer can still accidentally use logic like: `(if (subtype? A B :dont-know) ...)`, which will follow the `true` path even if `:dont-know` is returned, because `:dont-know` is considered a *truthy* value in Clojure.

	(1) Scala default	(2) Scala inhabited	(3) Clojure default	(4) Clojure inhabited
accuracy %	66.6	57.8	49.4	52.9
accuracy DNF %	81.0	92.0	61.7	83.0
gain %	18.0	35.5	20.2	33.0
loss %	3.6	1.4	7.9	2.4

Table 1: Accuracy results of `subtypep`

5 CONCLUSION

5.1 Results

We have conducted experiments to measure the accuracy of the SETS algorithm in Clojure and Scala. In both implementation host languages the test proceeds similarly. We select two type designators, `td1` and `td2`, by constructing them at random but limiting them to a maximum depth. Next we test whether `td1` is a subtype of `td2`. At each iteration we remember whether this test returns `true`, `false`, or `dont-know`. Thereafter, we canonicalize the two type designators to DNF, `dnf1` and `dnf2`; and perform the subtype test on the new type designators. We repeat this procedure 10000 times and tally the results.

The use of randomly generated test cases without regard to likelihood is a common practice in property based testing [14].

Column 1 of Table 1 summarizes the results when run in Scala. Canonicalizing to DNF form gives an apparent 18% increase in accuracy. However, we did notice that in 3.6% of the cases, `subtypep` was able to determine the relation on the non-normalized forms, but was lost that ability after converting to DNF.

These results are highly dependent on the generator of randomized type designators. We observed that about 50% of the time, a type designator was generated for which it was impossible to determine its habitation, and that `td1` and `td2` were equivalent types 20.5% of the time. In the second test we restricted the generator to produced pairs of type designators which are inhabited and not equivalent. Results of this second test are shown in column 2.

Columns 3 and 4 show similar results for Clojure Genus. We notice that the percentages are different for Clojure as compared to Scala. The subtype decision procedure is roughly 20 to 30 percent more accurate if the type designators are canonicalized. However, there is a small set of cases (2 to 8 percent) where the subtype relation is computable before canonicalization but not after.

Recall that by *accurate* we are not referring to whether the subtype relation can be determined, but rather whether one algorithm is able to determine it while another algorithm reports *dont-know*.

To help the reader understand some reasons why the testing results differ for the two Genus implementations, we offer several comments. (1) The random type generator for the intersection and union types in the Clojure implementation, always generates binary operations, while the Scala generator generates from 1 to 5 arguments, thus the random sample of types is different. (2) The `subtype?` in Clojure diverges from the corresponding code in Scala because they have been developed at different times. An audit of the code is needed to verify that the two implementations have exactly the same sets of necessary and sufficient conditions. (3) The set of base types in the Scala Genus code base includes user defined classes, to model the fact that Scala programs may make heavy use of user defined types; whereas the Clojure Genus code base has no such test cases, to model the fact that user defined types are much rarer in Clojure programs.

5.2 Perspectives

In the next phase of our research, we will present a theory describing how symbolic finite automata [11, 49] can be used in pattern recognition of heterogeneous sequences of types belonging to SETS. In order to accept such sequences, we will consider finite automata whose transitions are labeled by SETS. This possibly infinite alphabet prevents a direct determinization process. However, the Boolean operations provided by SETS will make it possible to computate an appropriate partition that allows us to make these automata finite and deterministic. This work already exists [28] for Common Lisp, and we consider it important to generalize to a wider class of programming languages with a sound theoretical foundation.

In Section 4.3 we defined the semantics of our subtype search to allow for classes to be loaded at compile time. It would be interesting to experiment with a model which assumes a disabled class loader. Our current research involves investigating the question in the Python Genus implementation of which we can walk down the class graph, determining definitively whether two given classes have a common subclass. We have not yet investigated whether the JVM reflection API allows us to perform such a downward search for classes currently defined.

In Section 4.1 we stated that the results are undefined if the Genus user employs parameterized Java types such as `Array[Int]`. We would like to characterize to which extent, if any, such types could be used, or at least whether such types could be detected and explicit exceptions thrown rather than leaving the behavior undefined.

As experimental support for our symbolic finite automata theory, we will implement regular type expressions (RTEs) for Clojure and Scala, and eventually also for Python. RTEs will allow us to specify and efficiently recognize regular patterns in sequences, as was alluded to in Section 1.1. The implementation in Clojure is well underway and is available in its preliminary form.

Baker's [3] algorithm contains many assumptions about the Common Lisp type system, such as how to handle ranges, complex numbers, and CLOS objects. SETS is by definition, a simpler type system; nevertheless, it is currently unclear how to enhance the SETS specification to make it possible to implement Baker's algorithm. We consider it a matter of ongoing research.

REFERENCES

[1] Nada Amin. Dependent Object Types. *unknown*, page 134, 2016.

[2] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[3] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992. URL http://dblp.uni-trier.de/db/journals/lisp/lisp5.html#Baker92a.

[4] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science*, 2018. URL http://www.di.unito.it/~dezani/papers/bbdgv18.pdf. to appear.

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure, 2018.

[6] Gilad Bracha. Generics in the java programming language. July 5, 2004. URL http://www.cs.rice.edu/~cork/312/Readings/GenericsTutorial.pdf.

[7] G. Castagna and V. Lanvin. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.*, unknown(1, ICFP '17, Article 41), sep 2017.

[8] Giuseppe Castagna and Alain Frisch. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 198–199, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6. doi: 10.1145/1069774.1069793. URL http://doi.acm.org/10.1145/1069774.1069793.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981. doi: 10.1002/malq.19810270205. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19810270205.

[10] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21 (4):685–693, 1980. doi: 10.1305/ndjfl/1093883253. URL https://doi.org/10.1305/ndjfl/1093883253.

[11] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*. Springer, July 2017. URL https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/.

[12] Sébastien Doeraene. Pseudo-union types in scala.js, August 2015. URL https://www.scala-js.org/news/2015/08/31/announcing-scalajs-0.6.5/.

[13] Joshua Dunfield. Elaborating Intersection and Union Types. *SIGPLAN Not.*, 47 (9):17–28, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364534. URL http://doi.acm.org/10.1145/2398856.2364534.

[14] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL https://doi.org/10.1145/263244.263267.

[15] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):19:1–19:64, September 2008. ISSN 0004-5411. doi: 10.1145/1391289.1391293. URL http://doi.acm.org/10.1145/1391289.1391293.

[16] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.

[17] Radu Grigore. Java generics are turing complete. *CoRR*, abs/1605.05274, 2016. URL http://arxiv.org/abs/1605.05274.

[18] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[19] Rich Hickey. A history of clojure. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386321. URL https://doi.org/10.1145/3386321.

[20] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

[22] Lionel Parreaux Jim Newton, Sébastien Doeraene. Union types in scala 3, feb 2020. URL https://contributors.scala-lang.org/t/union-types-in-scala-3/4046.

[23] Douglas Katzman. Email conversation with SBCL developer about the undocumented subtypep implementation with SBCL, February 2021.

[24] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, January 2007. URL https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/.

[25] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390590X.

[26] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 165–174, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800528. URL http://doi.acm.org/10.1145/800017.800528.

[27] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL https://doi.org/10.1145/367177.367199.

[28] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne University, November 2018.

[29] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[30] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[31] Martin Odersky. Dotty Documentation, 0.10.0-bin-SNAPSHOT, August 2018. URL http://dotty.epfl.ch/docs/reference/overview.html.

[32] Martin Odersky and Matthias Zenger. Scalable component abstractions. volume 40, pages 41–57, 10 2005. doi: 10.1145/1103845.1094815.

[33] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[34] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[35] Andreas Paepcke. User-Level Language Crafting – Introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. URL http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps. Downloadable version at url.

[36] David J. Pearce. Rewriting for sound and complete union, intersection and negation types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 117–130, 2017. doi: 10.1145/3136040.3136042. URL http://doi.acm.org/10.1145/3136040.3136042.

[37] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee. ISBN 9781450341431. doi: 10.1145/2872427.2883029. URL https://doi.org/10.1145/2872427.2883029.

[38] G. Pottinger. A type assignment for the strongly normalizable lambda-terms. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

[39] John C. Reynolds. Design of the Programming Language Forsythe. Technical report, 1996.

[40] Christophe Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, pages 74–86, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89275-5.

[41] Juan Jose Garcia Ripoll. ECL 9 release notes, may 2003. URL https://mailman.common-lisp.net/pipermail/ecl-devel/2003-May/000288.html.

[42] Tiark Rompf and Nada Amin. Type Soundness for Dependent Object Types (DOT). *SIGPLAN Not.*, 51(10):624–641, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984008. URL http://doi.acm.org/10.1145/3022671.2984008.

[43] Miles Sabin. Miles sabin via twitter. URL https://twitter.com/milessabin/status/953713425124818949?lang=en.

[44] Miles Sabin. Unboxed union types in Scala via the Curry-Howard isomorphism, June 2011. URL http://milessabin.com/blog/2011/06/09/scala-union-types-curry-howard/.

[45] Guy L. Steele, Jr. *Common LISP: The Language (2nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[46] Leo Valais. SUBTYPEP: An Implementation of Baker's Algorithm. Technical report, EPITA/LRDE, July 2018. URL https://www.lrde.epita.fr/wiki/Publications/valais.18.seminar.

[47] Léo Valais, Jim Newton, and Didier Verna. Implementing baker's SUBTYPEP decision procedure. In *12th European Lisp Symposium*, Genova, Italy, April 2019.

[48] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[49] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 640–654, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16242-8.