A History of Some Confusing and Entertaining Aspects of the Common Lisp Type System

A. Nonymous[WWWW-XXXX-YYYY-ZZZZ]

My Institute, My address, My Country my@email.address

Abstract. We summarize some aspects of the Common Lisp type system including a definition of types, type specifiers, and how they interact with objects in the language. The language makes use of several concepts which were not theoretically well understood at the time of the language's inception. We give a high-level profile of some reflection capabilities including membership and type inclusion. The type inclusion question may be unanswerable in some situations, and calculable in others. We give a summary and in depth discussion of some of the obscure features of the Common Lisp type system, especially with regard to degenerative cases of function types. As Common Lisp is defined by specification rather than implementation, we do not propose any fixes for problems we observe; rather we simply investigate them. Our discussion shows some corner cases which we find curious. Finally, we attempt to present a historical perspective of why things are the way they are.

1 Introduction

In this article we move toward bridging the gap between theory of and practice relating to a lattice based type system in a dynamic language with compilation semantics. We take a look at some features of the Common Lisp type system which was designed during the 1980s, and included many ideas which theoreticians are still addressing today. We reflect on some rather surprising behavior of function types, especially in light of historical aspects of their development and implementation.

In Sec. 2, we provide a brief introduction to the Common Lisp language including history and syntax. In Sec. 3 we introduce the type system including definition and meaning of types. Particularly notable is the way the type system handles the unanswerable subtype question using 3-way logic, in Sec. 3.3. In Sec. 4 we deal specifically with function types with special attention to the fact that the specification does not explain how the compiler should compute the subtype relation with function types. In Sec. 5 we provide a short historical perspective of why certain decisions were made in the design of the type system. Finally in Secs. 6 and 7 we provide a review of some related work and provide some perspectives of the next steps of this research.

2 What is Common Lisp?

Common Lisp [3] as a programming language is about 40 years old, if we count from the publication of *Common Lisp, the Language* [54], 1st edition. The language is defined by a specification rather than by an implementation. In practice this means that users of the language can defend their point of view regarding how the language should behave by pointing to excerpts from the specification, independent of what an actual Common Lisp compiler does or fails to do. Compiler implementors are obliged to fix their implementation when errors are discovered or risk being seen as non-conformant. This language concept has advantages as well as admitted disadvantages. One advantage is portability. A compliant Common Lisp program can be ported to another Common Lisp compiler, and code can be written to work across implementations. A disadvantage is that in the year 2022, with 40 years of hindsight, we might judge that certain aspects of the specification are not as great an idea as they were thought to be in 1985. One such problem which we look at in this article is what we consider to be a problem in the way function types were defined.

The Common Lisp language includes features which were novel at the time of its inception, but which have been introduced in more *modern* languages, lexical and dynamic scoping, such as first-class lexical closures, meta-programming, operating system independent path-name management; object orientation [34] supporting multiple inheritance, multi-methods, and meta-classes [42, 35], and the feature which we address in this article, the lattice-based type system supporting union, intersection, and complement types. The type system can be used by the compiler to optimize speed, safety, and memory usage, but it can also be used interactively either during debugging or reflectively in compiled code. In 2008, Frisch *et al.* [23] provide a sound theoretical basis for using sets as a basis of reasoning about types. However, Common Lisp approached this topic pragmatically during the 1980s.

We contend that It is academically relevant to analyze the strengths and weaknesses of Common Lisp. Admittedly there are not as many Common Lisp projects in active development today when compared to more modern languages; however an active Common Lisp community exists and the language is alive in many industrial applications. Google is a intense user of Common Lisp, makes regular commits to the SBCL compiler GitHub repository [59], and publishes a Common Lisp coding Style Guide [12]. AllegroGraph [22] is an example of an industrial graph data base. Common Lisp compilers are in active development: Schafmeister *et al.* [53] are developing an implementation specializing in C++ interoperability; Strandh [57, 56] is working toward a modern Common Lisp implementation which optimizes for modern memory constraints; and Steel Bank Common Lisp (SBCL) [48], a mature and stable compiler implementation we use in this article.

The default syntax of the Common Lisp language is simple. We say, *default* syntax, because the Common Lisp application programmer is allowed to change the reader syntax of the language. The source code is represented as a list of symbols and other serializable objects such as numbers and string. Functions and

data have the same syntax, and programmers may treat functions as data, both at the source code level and at the evaluation level. *E.g.*, Common Lisp macro programming allows the user to use the full force of the language to manipulate source code text as data, creating function bodies or splice values into function bodies which are then passed to the compiler.

Example 1 (Simple Common Lisp function definition).

```
(defun factorial (n)
  (if (> n 0)
      (* n (factorial (- n 1)))
      1))
```

Example 1 shows a crash course in Common Lisp programming. The example defines a recursive implementation of the factorial function, using `defun`. Functions are called using prefix notation *e.g.* `(> n 0)` calls the function `>` with two arguments, `n` and `0`.

3 Types in Common Lisp

From the perspective of the Common Lisp programmer, the use of type declarations is optional. However, type annotations may be added to function to help document code usage, or to declare the programmers intent. *E.g.*, even though a particular function *could* be called with any type of argument, the programmer might known that it is only ever actually called with an integer. Such annotations can also help the compiler create more optimal machine code by removing unnecessary type-case analysis or other safety checks.

A detailed summary of the Common Lisp types with regard to peculiarities of function types can be found in [37]. We present here a shorter version of that exposé.

Definition 1 (type). *In Common Lisp, a type is a (possibly infinite) set of objects at a particular point of time during the execution of a program [3].*

Actually the Common Lisp specification says *A type is a (possibly infinite) set of objects;* we have augmented the definition with the phrase *at a particular point of time during the execution of a program.* We believe this relaxation of the definition is necessary because of the dynamic nature of the Common Lisp language.

In Common Lisp, types and functions may be redefined. Consequently a type specified by `(satisfies f)`, which specifies the type of all Common Lisp objects for which the predicate `f` returns `true`, may change its meaning of the function `f` is redefined. This means that if a particular object belongs to type `(satisfies f)` at one moment, the object may no longer belong to `(satisfies f)` after `f` has been redefined, even though the object itself did not change.

An object of a particular class may be victim to the `change-class` function. *E.g.*., if an object, `x`, has class `pawn`, then after `(change-class x 'queen)`, then `x`, will no longer be of class `pawn`.

Finally, during development of a program, a type definition might change. For example `Y` may be defined as the type `(or number string)`, at which point the string `12` will be of that type, but if `Y` is redefined to `(or (and number (not integer)) string)`, then `12` is no longer a member of type `Y`.

All of these situations as well as several others may cause a type to change its members while a program is running.

3.1 Some Properties of Types

Notation 31 ($\perp$ and $\top$). The symbol, $\perp$, represents the Boolean *false* value, in a Boolean algebra context; the empty type, in a type system context; or the empty set, $\emptyset$, in a set theoretical context.

The symbol, $\top$, represents the Boolean *true* value, in a Boolean algebra context; the universal type, in a type system context; or the universal set, in a set theoretical context.

As illustrated in Fig. 1, Common Lisp programmers may take many ideas about types from the intuition they already have from set algebra. Two given types may be intersecting such as the case of `unsigned-byte` and `fixnum` in the figure, $(\texttt{unsigned-byte} \cap \texttt{fixnum}) \neq \emptyset$. Types may be disjoint such as `float` and `fixnum`, $(\texttt{float} \cap \texttt{fixnum}) = \emptyset$. Types may have a subtype relation such as `fixnum` and `number`, $(\texttt{fixnum} \subseteq \texttt{number})$. Types may have more complicated relations such as $\big(\texttt{bit} \subseteq (\texttt{fixnum} \cap \texttt{unsigned-byte}) \subseteq \texttt{rational}\big)$.

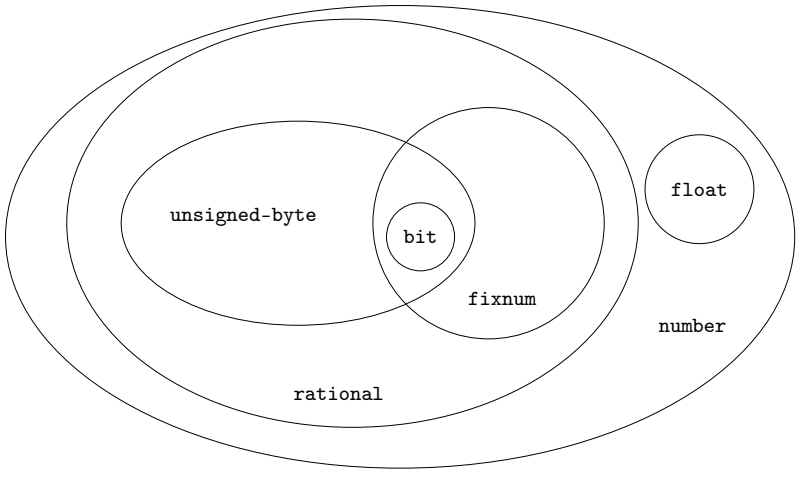

Fig. 1. Some Common Lisp types showing their intersection and subset relations

An object can belong to more than one type. Types are never explicitly represented as objects by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*.

Definition 2 (type specifier). *From the Common Lisp specification [3], a type specifier is an expression that denotes a type.*

The symbol `random-state`, the list `(integer 3 5)`, the list `(and list (not null))`, and the class named `standard-class` are type specifiers. A further discussion including more examples can be found in the Common Lisp specification [3, Sec. 4.2.3].

New type specifiers can be defined using `deftype`, `defstruct`, `defclass`, and `define-condition`. But 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`.

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 specified type. The `typep` function returns `T` or `NIL` assuming it is called with valid arguments. Of course `typep` would trigger an error if called with a second argument which does not specify a type: *e.g.* `(typep 1 2)`, as 2 does not specify a type. Another important situation where `typep` signals an error is with regard to certain function types, as explained in Sec. 4.8.

We sometimes wish to ask whether one type, T_1, is a subtype of another type T_2, for this the `subtypep` is useful: `(subtypep T1 T2)`. If we wish to know whether a type is empty, we ask whether it is a subtype of the empty type, `(subtypep T1 nil)`. If we wish to know whether the types are disjoint, we ask whether their intersection is empty, `(subtypep (and T1 T2) nil)`.

The function call `(subtypep T1 T2)` distinguishes three cases:

1. That `T1` is a subtype of `T2`, returning `T, T`,
2. That `T1` is not a subtype of `T2`, returning `NIL, T`,
3. Or that subtype relationship cannot be determined, returning `NIL, NIL`

Section 3.3 addresses situations in which the subtype relationship cannot be determined.

3.2 Meaning of Type Specifiers

There is some disagreement among experts as to how to interpret certain semantics of type specifiers in Common Lisp. The situation that the programmer may specify a type such as `(and fixnum (satisfies evenp))` is particularly problematic, because the Common Lisp specification contains a dubious, nonconforming example in the specification of `satisfies`. The problematic example in the specification says that `(and integer (satisfies evenp))` is a type specifier and denotes the set of all even integers. This claim contradicts the Common Lisp specification in at least two ways.

Bourguignon [11] explains the first violation that `evenp` is not a conforming argument of `satisfies`. The argument of `satisfies` must designate a *predicate*, which is a function (not a partial function) which returns a generalized Boolean as its first value.[1] A function which may fail to return is not a predicate. In particular `(evenp nil)` does not return, but instead signals an error.

The second reason that `(and fixnum (satisfies evenp))` is non-conforming is that the specification of the `AND` type specifier states that `(and A B)` is the intersection of types `A` and `B` and is thus the same as `(and B A)`. This presents a problem, because `(typep 1.0 '(and fixnum (satisfies evenp)))` evaluates to `NIL` while `(typep 1.0 '(and (satisfies evenp) fixnum))` signals an error.

For the purpose of this article, we assume that `(and A B)` is the same as `(and B A)`. Specifically, we interpret the `AND` and `OR` types as being commutative with respect to their operands. We would like to consider that type checking with `typep` is side-effect free, and in particular that it never signals an error. This assumption would allow programs to reorder the checks as long as we do not change the semantics of the Boolean algebra of the `AND`, `OR`, and `NOT` specifiers.

In the strict sense it is false to assume that `typep` never signals an error. The specification states that certain run-time calls to `typep` even with well-formed type specifiers must signal an error, such as if the type specifier is a list whose first element is `values` or `function`. Additionally, an evaluation of `(typep object '(satisfies F))` will signal an error if `(F object)` signals an error. One might be tempted to interpret `(typep object '(satisfies F))` as `(ignore-errors (if (F object) t nil))`, but that would be a violation of the specification which is explicit that the form `(typep x '(satisfies p))` is equivalent to `(if (p x) t nil)`.

3.3 Subtype Relation Unanswerable

As explained already the `subtypep` tries to determine whether the two specified types are in a subset relation, but there is an important caveat. The `subtypep` function is not always able to determine whether the specified types have a subtype relationship or not [4]. In such a case, `subtypep` returns `NIL` as its second return value.[2] This situation occurs most notably in the cases involving the `satisfies` type specifier. Consider Ex. 2 using the types `(satisfies evenp)` and `(satisfies oddp)`. The first problem we face is that an attempt to test type membership using such a predicate may be met with errors, such as when the argument of the `oddp` function is not an integer, as shown in Ex. 2.

Example 2 also illustrates the convention that that code typed by the user is typically represented in lower case, and return values printed by the REPL are usually represented in capital letters.

[1] Common Lisp functions may return multiple values. Function composition passes along the first value. Special forms, such as `multiple-value-bind`, are used to obtain values other than the first.

[2] Recall that Common Lisp functions return multiple values.

Example 2 (Problems with `satisfies`).

```
CL-USER> (typep 1 '(satisfies oddp))
==> T
CL-USER> (typep 0 '(satisfies oddp))
==> NIL
CL-USER> (typep "hello" '(satisfies oddp))

The value "hello" is not of type INTEGER.
   [Condition of type TYPE-ERROR]
```

Even though usage of `satisfies` with the predicate `oddp` is in violation of the Common Lisp specification (according to our interpretation explained in Sec. 3.2), such usage seems to be fairly common in real-world Common Lisp programs. For example the Google corporate Common Lisp coding Style Guide [12, Sec Pitfalls] recommends using the human readable form as shown in Ex. 3 as well as assuring the function `prime-number-p` "MUST accept objects of any type". Additionally, the Google style guide requires that the predicate be callable at compile time.

We also note here that the example in the Google corporate Style Guide violates the advice of Norvig and Pitman [39, p 13] in its usage of `when` as a two-branch expression.

Example 3 (Google Common Lisp Style Guide advice example.).

```
(deftype prime-number ()
 (satisfies prime-number-p)) ; Bad
(deftype prime-number ()
 (and integer (satisfies prime-number-p)) ; Better

(eval-when (:compile-toplevel
            :load-toplevel
            :execute)
  (defun prime-number-p (n)
    (when (integerp n) ; Better
      (let ((m (abs n)))
        (if (<= m *prime-number-cutoff*)
            (small-prime-number-p m)
            (big-prime-number-p m))))))
```

Because of the error demonstrated in Ex. 2 it becomes a little easier if we define two types `odd` and `even` using `deftype`, Ex. 4. We see very quickly that the system has difficulty reasoning about these types.

Example 4 (Definition of `odd` and `even` types).

```
CL-USER> (deftype odd ()
           '(and integer (satisfies oddp)))
==> ODD

CL-USER> (deftype even ()
```

```
            '(and integer (satisfies evenp)))
==> EVEN

CL-USER> (subtypep 'odd 'even)
==> NIL, NIL

CL-USER> (subtypep 'odd 'string)
==> NIL, NIL
```

The `subtypep` function returns `NIL` as its second value, indicating that SBCL [48] is unable to determine whether `odd` is a subtype of `even`. Similarly, SBCL is not able to determine that `odd` is not a subtype of `string`. This behavior is conforming, albeit disappointing. According to the Common Lisp specification, the `subtypep` function is permitted to return the values `NIL, NIL` (among other reasons) when at least one argument involves type specifier `satisfies` [3].

SBCL cannot determine whether two arbitrary functions might return `true` given the same argument. The human can see that the types `odd` and `even` are non-empty and disjoint, and thus neither is a subtype of the other.

Sometimes, the result of the `subtype` function can seem curious. Notice that `(subtypep 'odd 'string)` returns `NIL, NIL` indicating that it was unable to determine whether `odd` is a subtype of `string`. However, at a glance it would seem that `subtypep` should return `NIL, T` in this case, indicating that the subtype relation is provably false; *i.e.* we might be tempted to think that `(and integer (satisfies oddp))` is definitely not a subset of `string`. After all, `(and integer (satisfies oddp))` is a subset of `integer`, and `integer` which is disjoint from `string`. But there's a catch. The system does not know that `odd` and `even` are inhabited types. If a type `A` is empty, then in fact `(and integer A)` is a subtype of `string`, because $integer \cap A = \emptyset \subseteq string$ [33].

4 Function Types

In this section we explain some interesting and perhaps confusing characteristics related to Common Lisp function types.

1. Function type specifiers do not explicitly describe sets of values, but rather valid code transformations (Sec. 4.3).
2. Function type specifiers in Common Lisp seem to violate the Induced Subtype Rule (ISR) (Sec. 4.4).
3. The Common Lisp specification does not explicitly explain how to compute subset relations between function types (Sec. 4.7).
4. Common Lisp programs are forbidden from making certain run-time function subtype checks (Sec. 4.8).

4.1 Function Declarations in Common Lisp

We now make a small digression to show some examples of function declaration syntax in order to make more sense of Sec. 4 for the reader unfamiliar with Common Lisp syntax.

The type of the input variable for a Common Lisp function can be declared using the `declare` form within the function declaration as shown in Ex. 5 which declares the input variable of `function-1` to have type `float`. The return value of `function-1` is left to the compiler to infer. Information about the input and output types of a function can also be declared using `declaim` in conjunction with `ftype` as is the case for `function-2` which declares a function which accepts a `float` and returns a `string`.

Example 5 (Function type declaration).

```
(defun function-1 (x)
   (declare (type float x))
   x)
==> FUNCTION-1
(declaim (ftype (function (float) string)
                function-2))
==> (FUNCTION-2)
(defun function-2 (x)
   (if (zerop x) "hello" "world"))
==> FUNCTION-2
```

4.2 Semantics of Function Types

Many approaches to type theory assign semantics to what we sometimes call *arrow types*, denoted $A \to Y$. The symbol $A \to Y$ carries the intuition of the set of functions mapping domain A to range Y, but the specifics vary depending on the particular type-theoretical approach taken. As we address in Sec. 4.3, the meaning of arrow types in Common Lisp is different from that in other common treatments.

Pierce [44, Ch 9] defines the arrow type formally with a syntax and rewriting (evaluation) semantics, such as

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t_2 : T_1 \to T_2} \qquad [\textsc{Abstraction}] \tag{1}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \qquad [\textsc{Application}]\,. \tag{2}$$

The rewriting rules define exactly in which contexts arrow types may appear in a program, and exactly what effects they have on program evaluation. That is, rules (1) and (2) define exactly how arrow types behave in the type system of Pierce without relying on any intuition.

Castagna *et al.* [23, Sec 2.6] summarize the constraints of defining arrow types consistently. In a one to one interview, when posed the question, "What does $A \to Y$ mean?" Dr. Castagna responded, "It means whatever you want it to mean for the type system you are trying to define, as long as you are consistent." He continued by giving an example of on-going work in which he defines arrow types for a lazy language[2] in which certain terms may diverge as long as they are never evaluated.

Definition 3 (arrow type). *If A and Y denote types, we take $A \to Y$ to mean the set of all unary functions mapping every element of type A to an element of type Y.*

If we take Def. 3 to be the definition of arrow types, then we explicitly define the type to *exclude* functions which diverge or loop forever, even though it is impossible to distinguish the two programmatically. The function `strange-function` defined in Ex. 6 is not in *integer* $\to$ *integer* according to Def. 3, because there exist integers, namely 0 and 1, for which the function fails to return an integer.

Example 6 (Definition of `strange-function`).

```
(defun strange-function (x)
  (declare (type integer x))
  (case x
   ((0)
    (loop :while t)) ; loop forever
   ((1)
    (error "some error"))
   (t
    (+ 1 x))))
```

4.3 Subtypes of Function Types

It is natural to ask under which conditions there is a subtype relation between two given arrow types. Common Lisp provides two mechanisms for specifying subtypes of `function`. The first is by using the Metaobject Protocol (MOP) [42, 35, 18]. With this approach, we use the object system, CLOS [26], to define classes which inherit from the `function` class, and simultaneously from other mix-in classes to extend the behavior of function objects. When function subclasses are defined this way, Common Lisp updates the type lattice with new types corresponding to these classes, which behave as one would expect with regard to `subtypep`.

The second way the Common Lisp type system provides for describing subtypes of `function` is with a particular type specifier syntax. This syntax specifies a subtype of `function` which depends on the types of parameters and return value. An s-expression such as `(function (arg-types) return-type)` designates a subtype of `function`. Although the Common Lisp syntax allows specifying multiple arguments as well as optional arguments and multiple return values, we limit our discussion to unary functions with a single return value.

The explanation of the function type in the Common Lisp specification is vague. Rather than a clear statement of what `(function (A) Y)` means in terms of types `A` and `Y`, [3, Sec. FUNCTION] gives an explanation which is neither a necessary condition nor a sufficient condition for a function to be an element of such a type:

Claim. Every element of type `(function (A) Y)` is a function that accepts arguments of type A and returns values of type Y.

The specification further explains that if F has type `(function (A) Y)`, the consequences are undefined if F is called with an element not of type A. The specification does not indicate exactly what "a function that accepts arguments of type A" means; *i.e.* it is unclear whether this means that only elements of A are valid arguments of the function, that every element of A is valid argument of the function, or that some elements of A are valid arguments of the function.[3] However, it does clarify that if the argument is not in A, then the result is not guaranteed to be in Y.

Rather than directly defining the semantics of the type `(function (A) Y)`, the specification says:

Claim (Call-site rewriting rule).
If a function `F` is declared of type `(function (A) Y)`, then any call such as `(F x)` may be replaced with the transformed code `(the Y (F (the A x)))`.

The Common Lisp special operator `the` specifies that the value (or values) returned from the form are of the designated type[3].

Because of Claim 4.3, we infer Def. 4, which we did not find anywhere explicitly stated in the Common Lisp specification.

Definition 4 (Lisp function type). *The type `(function (A) Y)` is the largest set of functions which has the following property: if F is in the set, then every call site of the form `(F x)` can be replaced with `(the Y (F (the A x)))`, without changing the evaluation of the call site.*

An admitted weakness of Def. 4 is that it is not general enough to apply to functions called indirectly such as the following because the implementation of `mapcar` cannot be redefined and recompiled for every different type of function pass as its first argument.

```
CL-USER> (mapcar #'F (list 1 2 3 4 5))
```

The Common Lisp specification (still in [3, Sec. FUNCTION]) states that an `ftype` declaration for a function describes calls to the function, not the actual definition of the function. The `ftype` declaration is explained in the Common Lisp specification[3], as specifying that the named functions (global or local) are of the indicated function type. The exact set-theoretical properties of `calls to functions` are still unclear, and as we will discuss in Sec. 4.7, the definition was also up for debate during the Common Lisp specification process. For this reason (and others which we describe in the following sections), we omit considerations of function types from the type specifier discussions in the following chapters.

4.4 The Induced Subtype Rule (ISR)

The subtyping consequence, Corollary 1 follows from Def. 3.

[3] A restaurant may accept credit cards, but may at the same time not accept American Express.

Corollary 1. *If arrow types are defined as in Def. 3, then whenever $B \subseteq A$, we have $(A \to Y) \subseteq (B \to Y)$.*

Proof. We can see this by taking an $f \in A \to Y$, and showing that necessarily $f \in B \to Y$.

Suppose $B \subseteq A$, and take $\beta \in B$. Since all elements of B are in A, we have $\beta \in A$ so $(f\beta) \in Y$.

Section 4.6 explains that this consequence does not apply to the Common Lisp type system. This discrepancy is mentioned in the X3J13 cleanup issue, FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE [49]. The cleanup issue says that function argument type semantics are different from what users expect. One might wonder what would happen if we considered function type specifiers as behaving like other Common Lisp types specifiers with regard to set semantics. Under such an assumption, we would be able to reason about the semantics of such types using the semantics of sets. What we find, however, is that some familiar intuitions fail. In this section (Sec. 4.4) we see that a principle known as ISR (Claim 4.4) fails, and in Sec. 4.7 we see that certain intuitions regarding type intersection fail.

Claim (Induced subtype rule, ISR).
If $A \subseteq B$ and $X \subseteq Y$, then $B \to X$ is a subtype of $A \to Y$, denoted $(B \to X) \subseteq (A \to Y)$.

Claim 4.4 is not actually a principle supported in Common Lisp rather it is an assumption some users make as is further discussed in Sec. 5. The claim, which is a generalization of Corollary 1, is basically saying that function types are (should be) covariant with respect to return type and contravariant with respect to argument type. Castagna [14, Sec. 2.2] provides a proof of this claim and for a weaker version of its converse.

As Barnett [25, 5] suggested, if Common Lisp had been specified to implement ISR, the one of Common Lisp early implementation goals to support incremental compilation could have been better supported. I.e., if a function is incrementally redefined in an interactive session, then call-sites are dubious and may need to be recompiled as well. However, if the function was changed to narrow its type, *i.e.* to accept a larger type of input or produce a smaller type of output, then call-sites need not be recompiled.

Since function type specifiers are valid arguments to the `subtypep` function, we can test Claim 4.4 using SBCL version 2.2.9 as in Examples 7 and 8. In Ex. 7, we see results which agree with the ISR. We see that since $fixnum \subseteq number$, then `subtypep` considers `(function (T) fixnum)` to be a subtype of `(function (T) number)`, and `(function (T) number)` not to be a subtype of `(function (T) fixnum)`.

Example 7 (Function type specifiers with covariant return types).

```
CL-USER> (subtypep '(function (t) fixnum)
                   '(function (t) number))
```

```
==> T,T

CL-USER> (subtypep '(function (t) number)
                   '(function (t) fixnum))
==> NIL,T
```

In contrast to Ex. 7, in Ex. 8 we see that `subtypep` does not respect ISR with respect to contravariance of function arguments. The `subtype` function, at least in SBCL, believes

$$(fixnum \to \top) \subseteq (number \to \top)$$

and

$$(number \to \top) \not\subseteq (fixnum \to \top)\,.$$

Example 8 (Function type specifiers with contravariant argument types).

```
CL-USER> (subtypep '(function (number) t)
                   '(function (fixnum) t))
==> NIL,T

CL-USER> (subtypep '(function (fixnum) t)
                   '(function (number) t))
==> T,T
```

4.5 Degenerate Function Types

In this section, we take a brief look at arrow types composed of $\top$ and $\bot$. We see that Common Lisp, in particular SBCL, supports these degenerate arrow types even though they seem to violate the definition of arrow types from Def. 4. First we derive a property of types $\top \to \bot$ and $\bot \to \top$ in Corollary 2, then test the predictions in Ex. 9.

A consequence of Claim 4.4 is shown in Theorem 1, and we can test the predictions of the theorem as shown in Ex. 9.

Theorem 1. *Given types A, B, X, and Y,*

$$\big((A \cup B) \to (X \cap Y)\big) \subseteq (A \to Y) \subseteq \big((A \cap B) \to (X \cup Y)\big)\,.$$

Proof. Since $A \subseteq A \cup B$ and $Y \cap X \subseteq Y$ by ISR we can conclude that

$$((A \cup B) \to (X \cap Y)) \subseteq (A \to Y)\,.$$

Since $A \cap B \subseteq A$ and $Y \subseteq X \cup Y$, then by ISR we can conclude that

$$(A \to Y) \subseteq \big((A \cap B) \to (X \cup Y)\big)\,.$$

So together we have

$$\big((A \cup B) \to (X \cap Y)\big) \subseteq (A \to Y) \subseteq \big((A \cap B) \to (X \cup Y)\big)\,.$$

In this section we take a brief look at the degenerate cases of Theorem 1.

Corollary 2. $(\top \to \bot) \subseteq (\bot \to \top)$.

Proof. Let $B = \neg A$ and $X = \neg Y$, and apply Theorem 1.

A note about notation: the Common Lisp type specifier `(function (t) nil)` which we also denote as $\top \to \bot$, has the sense of functions whose argument comes from the universal type t and whose return value comes form the empty set. In Common Lisp a function that returns, necessarily returns a value, so if a return value comes from empty set, that means the function does not return, for example by a transfer of control such as non-local exit, or signaling a condition.

In Common Lisp, so-called *conditions* [29] generalize what are modeled as exceptions in many languages. Conditions need not be exceptional, neither indications of error. A condition may notify the calling function of an error, a warning, an information notification, an advisory event, or possibly many other things. A critical difference between a Common Lisp condition and an exception (like in Java) is that conditions can be handled without unwinding the stack, and in some cases the condition handler may take an action which causes the execution to continue from the point where the condition was signaled.

In Ex. 9 we test Corollary 2. We see that SBCL does not respect the ISR. According to SBCL $(\top \to \bot)$ is not a subtype of $(\bot \to \top)$. In fact neither is a subtype of the other.

Example 9 (Testing Corollary 2 in SBCL).

```
CL-USER> (subtypep '(function (t) nil)
                   '(function (nil) t))
==> NIL,T

CL-USER> (subtypep '(function (nil) t)
                   '(function (t) nil))
==> NIL,T
```

The correspondence of $\top \to \bot$ and $\bot \to \top$ to Common Lisp types is difficult to grasp. We see in the Ex. 10 that neither `(function (nil) t)` nor `(function (t) nil)` is empty as neither is a subtype of `nil`.

Example 10 (Degenerate function non-empty).

```
CL-USER> (subtypep '(function (nil) t) nil)
==> NIL,T

CL-USER> (subtypep '(function (nil) t) t)
==> T,T

CL-USER> (subtypep '(function (t) nil) t)
==> T,T

CL-USER> (subtypep '(function (t) nil) nil)
==> NIL,T
```

According to Def. 3, a function in $\bot \to \top$ is a function which gets is input argument from empty set. It is possible in Common Lisp to define such an *absurd* function, as shown in Ex 11, albeit with lots of warnings. However, it is impossible to call such a function correctly, because to do something we'd have to find a value in the empty set to pass as argument.

Example 11 (Functions types non-disjoint).

```
CL-USER> (defun absurd (x)
           (declare (type nil x))
           42)
; in: DEFUN ABSURD
;     (TYPE NIL X)
;
; caught WARNING:
;   The type declarations T and NIL for X conflict.
;   See also:
;     The SBCL Manual, Node "Handling of Types"

;     (SB-INT:NAMED-LAMBDA ABSURD
;         (X)
;       (DECLARE (TYPE NIL X))
;       (BLOCK ABSURD 42))
;
; caught STYLE-WARNING:
;   The variable X is defined but never used.
;
; compilation unit finished
;   caught 1 WARNING condition
;   caught 1 STYLE-WARNING condition
==> ABSURD
```

We further see in Ex. 12 that the two types, `(function (nil) t)` and `(function (t) nil)`, are non-disjoint (*i.e.* intersecting) types, at least as far as SBCL is concerned.

Example 12 (Functions types non-disjoint).

```
CL-USER> (subtypep '(and (function (nil) t)
                         (function (t) nil))
                   nil)
==> NIL,T
```

By Def. 4, `(function (nil) t)` is the set of functions F such that `(F x)` can be replaced everywhere with `(F (the nil x))`. Likewise, `(function (t) nil)`, is the set of functions G such that `(G x)` can be replaced with `(the nil (G x))`. According to the `subtypep` implementation in SBCL, those are two distinct, intersecting, non-empty sets of functions. Exactly which two sets of functions are specified by these type specifiers is not clear.

Before leaving the discussion of degenerate arrow types in Common Lisp, we look briefly at $\bot \to \bot$ and $\top \to \top$. Example 13 shows that in Common Lisp we have $\bot \to \bot \subsetneq \top \to \top$, a strict subset relation.

Example 13 (`subtypep` with degenerate arrow types).

```
CL-USER> (subtypep '(function (nil) nil)
                   '(function (t) t))
==> T,T

CL-USER> (subtypep '(function (t) t)
                   '(function (nil) nil))
==> NIL,T
```

The strict subset relation between $\bot \to \bot$ and $\top \to \top$ makes sense according to Def. 4. `(function (t) t)` is the set of unary functions which either diverge or not, *i.e.* the set of functions F for which `(F x)` can be replaced by `(the t (F (the t x)))`; while `(function (nil) nil)` contains only functions which diverge, *i.e.* `(F x)` can be replaced with `(the nil (F (the nil x)))`. It is, however curious to note that `(function (nil) nil)` does not contain all divergent unary functions, because, as Ex. 13 also shows, $\bot \to \bot \subsetneq \top \to \bot$, *i.e.* a strict subset relation.

4.6 Intuition of Function Type Intersection

The examples and results shown in Secs. 4.4 and 4.5 may come as a surprise to some readers, as they seem in violation of ISR. The theory of semantic subtyping [24] is a branch of computer science which attempts to put consistent semantics on types and subtypes to intuitively align with set theoretical semantics. As we explain in more detail in Sec. 6, at the time Common Lisp was being specified, the theory of semantic subtyping was not yet well understood. The Common Lisp specification does not explain explicitly how the `subtypep` function should behave given function type specifiers.

To illustrate the failing intuition of intersection, consider the `stringify` function:

```
(defun stringify (x)
  (format nil "~A" x))
```

The function `stringify` certainly maps all ratios to string. According to Claim 4.3 we might be tempted to think that the function `stringify` is an element of the type `(function (ratio) string)`, and since the function also maps integers to strings, we might be tempted to think that it is an element of type `(function (integer) string)`. Furthermore, since the function is in both types, it must be an element of the intersection of the two types. Such an interpretation of Claim 4.3 would be erroneous according to the call-site rewriting rule (Claim 4.3). If the function `stringify` were a member of `(function (ratio) string)` then we would be able to rewrite any call-site, even `(stringify 3.1416)`, by the call-site rewriting rule. But we know that `(stringify 3.1416)` is not

equivalent to `(the string (stringify (the ratio 3.1416)))`, as `3.1416` is not an element of the type `ratio`.

Similarly, staying with our erroneous intuition, if a function is in the intersection of the two types: `(function (ratio) string)` and `(function (integer) string)`, *i.e.*, if that function maps both ratios and integers to strings, then that function maps all `(or ratio integer)` to strings.[4] From this logic, and in keeping with the intuitions of semantic type theory, one would conclude that the intersection of the two types was simply

$$\texttt{(function ((or ratio integer)) integer)} = \texttt{(function (rational) integer)}\,.$$

One would infer the rule $(A \to Y) \cap (B \to Y) = A \cup B \to Y$. However, as explained in Sec. 4.7, the Common Lisp type system works differently in relation to function type intersection.

4.7 Calculation of Function Subtype Relation is Under-Specified

Section **System Class FUNCTION** of the Common Lisp specification does not specify how to compute the intersection of two function types of the same arity, but it does specify what rule the compiler may apply whenever a function is simultaneously declared to be in two function types. We state the rule restricted to unary function with a single return value. If the two type declarations for function `F` are in effect:

```
(function (A) X)
(function (B) Y)
```

then within the shared scope of the declarations, calls to `F` can be treated as if `F` were declared as the following:

```
(function ((and A B)) (and X Y))
```

We restate here the Common Lisp definition of function type intersection using arrow notation.

Claim (Intersection induced subtype relation). If A, B, X, and Y are types, then the following relation holds with regard to intersections:

$$(A \to X) \cap (B \to Y) \subseteq (A \cap B) \to (X \cap Y)$$

Doel [19] points out that the two are not in explicit contradiction. However, the intersection in Claim 4.7 is different from what we would guess from the ISR (Claim 4.4), which says that

$$(A \to Y) \cap (B \to Y) \subseteq (A \cup B) \to Y\,.$$

We also find no description in the Common Lisp specification for answering subtype questions about unions and complements of functions or functions of unions and complements.

[4] In Common Lisp the name given to `(or ratio integer)` is `rational`.

4.8 Run-Time Type Check of Functions Considered Harmful

At run-time a program is allowed to use `typep` to check whether an object is of type `function`, but only in the most simple form. An expression such as `(typep object 'function)` is permitted. However, the specification explicitly forces any run-time call to `typep` to signal an error, if a function type such as `(function () t)` is given as in Ex. 14.

Example 14 (Certain function types are not legal arguments of `typep`*.).*

```
CL-USER> (typep (lambda () 42) 'function)
==> T

CL-USER> (typep (lambda () 42)
                '(function () fixnum))

Function types are not a legal argument to TYPEP:
  (FUNCTION NIL (VALUES FIXNUM &REST T))
   [Condition of type SIMPLE-ERROR]
```

The specification does not justify the requirement that `typep` signal an error. One might naively think that the specification prohibits implementations for doing a correct job; *i.e.*, `(typep (lambda (x) x) '(function (t) t))` is prohibited to evaluate to `T`. We believe motivation for this requirement is indeed justified, and simply (and regrettably) undocumented in the specification. Imagine that `A`, `B`, and `X` are type specifiers and that `(subtypep 'A 'B)` returns `NIL, NIL`, indicating that the subtype relation cannot be determined. Further imagine that the function `f` has type $A \to X$. In this case it would be impossible for `(typep #'f '(function (B) X))`, as the internal call `(subtypep 'A 'B)` will have returned `NIL, NIL` (*i.e.*, don't know).

On the contrary, it is not clear from the specification what should occur with a run-time call to `subtypep` with the list form of a `function` type specifier. The specification contains what some people might interpret as contradictory information. It says, "The list form of the function type-specifier can be used only for declaration and not for discrimination." A call to `subtypep` is arguably neither declaration nor discrimination, at least not object discrimination. Even so, a run-time call to `subtypep` is not a declaration. We might reasonably conclude, based on that statement, that `subtypep` is not allowed to be called with the list form of a `function` type specifier, but that conclusion would be premature.

On the other hand, the specification for `subtypep` has a non-definitive clarification. It says "`subtypep` is permitted to return the values `NIL, NIL` (*i.e.*, don't know) *only when* at least one argument involves one of these type specifiers: `and`, `eql`, the list form of `function`, `member`, `not`, `or`, `satisfies`, or `values`." Even though *only when* does not mean *if*, we nevertheless interpret this statement to mean that a call to `subtypep` with `function` type specifiers is allowed.

The problem of these inconsistencies seems to imply that programs should not rely on the return value of `subtypep` when dealing with function types.

5 A Historical Perspective

Concerning the inconsistencies mentioned in Sec. 4.8, it would seem that at least one of the contributors to the X3J13 cleanup issue, FUNCTION-TYPE--ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE [49] shared our view, at least to some extent. We find a remark in the final paragraph of the cleanup issue, that the notion of "subtype" *does not make sense* in conjunction with the list form of function types because it is *different from most type specifiers*.

It would appear that the architects of Common Lisp were not oblivious to the question of whether ISR should apply to Common Lisp function types. As mentioned briefly above, there is an X3J13 cleanup issue named FUNCTION--TYPE-ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE [49]. This issue discusses the problem that the function argument type semantics are different from what users expect. It is mentioned that CLtL [55, pp 47-18] requires the `cons` function be type `(function (t t) cons)` and also of type `(function (float string) list)`. If this requirement is interpreted according to ISR, then the author is suggesting that `(function (float string) list)` is a subtype of `(function (t t) cons)` with covariant return value, $list \subseteq cons$; but contravariant arguments, $float \subseteq \top$ and $string \subseteq \top$.

According to the accepted Common Lisp specification (rather than that proposed in the cleanup issue), the `cons` function is not of type `(function (float string) list)`, because `(cons t nil)` is a valid call to `cons` but `(the list (cons (the float t) (the string nil)))` is not.

There is also a discussion in the X3J13 email archive [25, 5] initiated by Jeff Barnett where he suggests and explains the ISR (of course without using that name) in what he called "PROBLEM II". His argument is that if $T_1 \subseteq T_2 \subseteq T_3$, then $(T_3 \to T_1) \subseteq (T_2 \to T_2)$ and $(T_1 \to T_3) \not\subseteq (T_2 \to T_2)$. Barnett demands that the `subtypep` function "must **reverse** the order of its arguments when checking argument types—original order is used when value types are checked." His reasoning was that most Common Lisp implementations included incremental compilation. And if a function is edited and recompiled in a way which narrowed its type (new type is a subtype of previous type) the compiler need not warn about call-sites. However, if the function type was changed in any other way, call sites might or might not be invalid.

Barnett's comments were motivated [8] by experience with the CRISP [9, 7, 6] language from the early 1970s which was part of a speech understanding systems in which run time efficiency was highly important. The CRISP language was specified to be a strongly typed Lisp dialect, which defined types and in particular function types in a way compatible with what we now call ISR. The CRISP language defined a type [9, p. 49] as a collection of objects.

Unfortunately, the only follow-up response to "PROBLEM II" was by Nick Gall [25, 27], who made a dubious claim which was never challenged. He claimed "The type specifier (FUNCTION ...) is not acceptable to TYPEP (pg. 47), therefore it is not acceptable to SUBTYPEP (pg. 72)." According to an interview with Nick Gall, he clarified that page 47 and 72 are referring to CLtL [54] edition 1

(not 2). Contrary to the Gall's claim, however, we could not find anywhere in the Common Lisp specification where run-time calls to `subtypep` are prohibited.

6 Related Work

Tobin *et al.* [58] argue that during the 1990s and later, developers preferred languages such as Lisp, Ruby, Python, and Perl because the statically typed languages which existed at the time lacked the flexibility they needed. The authors present a system for migrating programs incrementally by declaring types within Racket programs which were previously implemented exploiting the flexibility of a language lacking type declarations.

Castagna *et al.* [13] argue as well that many programmers prefer the flexibility and development speed of dynamically typed languages, but that, nevertheless, compilers may infer types from the program if certain language constructs exist. The authors explore gradual typing in languages which support set-theoretical types–languages whose types support union, intersection, and complementation. In particular they argue that adding such set theoretical operations to a type system facilitates a smooth transition from dynamic typing to static typing while given even more control to the programmer.

Around the time Common Lisp was under development, other programming language researchers were experimenting with set semantics for types. We have not been able to determine how much of this research influenced Common Lisp development or vice versa. Dunfield [21] discusses works related to intersection types, but omits explicit references to the Common Lisp type system. He does mention the Forsythe [47] programming language developed in the late 1980s, which provides intersection types, but not union types. Dunfield claims that intersections were originally developed in 1981 by Coppo *et al.* [16], and in 1980 by Pottinger [45] 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.* [36].

As in Common Lisp, languages which support intersection and union types [13, 15, 43], should also be consistent with respect to subtype relations. Frisch *et al.* [24] 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 [17] 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 some influence on modern programming language extensions, at least in Java and Scala. Bettini *et al.* [10] (including Dezani, mentioned above) discuss the connection of Java λ-expressions to intersection types.

The Scala language [41] supports a type called `Either` as part of its standard library. 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 not associative

$$Either[Either[A, B], C] \neq Either[A, Either[B, C]]\,,$$

neither is it commutative

$$Either[A, B] \neq Either[B, A]\,.$$

Sabin [52] discusses user level extensions to the Scala type system to support intersection and union types. However, Sabin [51] also recommends that no one use his implementation in *real code*. Doeraene [20, 32] introduces pseudo-union types in Scala.js. Scala 3 [1, 50, 40] includes support for intersection and union types in a type lattice, but not complementary types.

In Sec. 4 we gave special attention to questions about the `(function (...) ...)` type in Common Lisp, with particular attention to the ISR. The section explains some of the historical perspectives which lead us to believe that the Common Lisp specification does not respect ISR. The specification, in our opinion, should either have an explicitly defined way to calculate the subtype relation for function types, or it should prohibit their usage in conjunction with `subtypep`. We realize our opinion may be controversial. The opinion of Bourguignon and others may be found in [38]. Finally, the Scala language [41, 46], allows programmers to decide within function and class declaration, which parameters are covariant, contravariant, or invariant.

7 Perspectives

In this article we have presented an analysis from the point of view of a user of the Common Lisp language, not as a compiler developer. A valid question might be: what could the compiler developers do to make function types semantics conforming but less confusing. We don't offer an answer to this question at present. If the analysis we offer, sparks a discussion, we consider our effort profitable.

We have done some investigation of function subtype semantics in one other Common Lisp implementation, CLISP version 2.49.92 (2018) [28]. We observed that in most of the non-trivial cases where SBCL returns confusing results about arrow types, clisp simply returns `NIL, NIL` (*i.e.*, don't know) which is of conforming behavior, but not informative.

We have omitted any discussion or historical search of tuple types. Common Lisp supports a tuple type named `cons`, which is simultaneously used to describe list types. There are X3J13 cleanup issues related to this, *e.g.*, RECURSIVE-DEFTYPE:EXPLICITLY-VAGUE.

For a better historical perspective, we would like to continue the investigation of the origins of the Common Lisp type system. We see several ideas in the Common Lisp type system which were areas of research outside the Lisp community during the time period that Common Lisp was being developed. However, we have not been able to find references between the two research communities.

For instance Common Lisp defines *type* as a set of objects; similarly Hosoya and Pierce [30, 31] simplify their model of semantic subtyping by modeling a type as a set of values of the language. Castagna and Frisch [15] claim that allowing function types to be sets rather than recursively defined structures with set semantics leads to a cardinality error where the set of objects contains its own power set.

References

1. Amin, N.: Dependent Object Types. unknown p. 134 (2016)
2. Ancona, D., Castagna, G., Petrucciani, T., Zucca, E.: Semantic subtyping for non-strict languages. In: 24th International Conference on Types for Proofs and Programs (TYPES 2018) (jun 2018)
3. Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)
4. Baker, H.G.: A Decision Procedure for Common Lisp's SUBTYPEP Predicate. Lisp and Symbolic Computation **5**(3), 157–190 (1992), `http://dblp.uni-trier.de/db/journals/lisp/lisp5.html\#Baker92a`
5. Barnett, J.: Types in CL. Computer History Museum Software Preservation Group (Dec 1987), `https://cl-su-ai.cddddr.org/msg04614.html`
6. Barnett, J.: The CRISP Programming Language System, An Historical Overview. Computer History Museum Software Preservation Group (Sep 2009), `http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/CRISP\-talk.pdf`
7. Barnett, J.: Notes on SDC CRISP for IBM 370 Computers. Computer History Museum Software Preservation Group (Jun 2010), `http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/notes/`
8. Barnett, J.: searching for Jeff Barnett. conversation on comp.lang.lisp (Aug 2018), `https://groups.google.com/d/msg/comp.lang.lisp/QLUWlAqejFA/5nvIEyFjAwAJ`
9. Barnett, J., Pintar, D.L.: CRISP: A Programming language and System (draft) (Dec 1974), `http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/TM-5444_000_00.pdf`
10. Bettini, L., Bono, V., Dezani-Ciancaglini, M., Giannini, P., Venneri, B.: Java & Lambda: a Featherweight Story. Logical Methods in Computer Science (2018), `http://www.di.unito.it/~dezani/papers/bbdgv18.pdf`, to appear
11. Bourguignon, P.J.: I'm annoyed by the specification for satisfies. Thread on comp.lang.lisp (Jan 2017), `https://groups.google.com/d/msg/comp.lang.lisp/w4bYVI7ieoA/Tn44-6coGgAJ`
12. Brown, R., Rideau, F.R.: Google Common Lisp Style Guide, Revision 1.28 (2018), `https://google.github.io/styleguide/lispguide.xml`, accessed 14 October 2018, 12h36 +0200
13. Castagna, G., Lanvin, V.: Gradual Typing with Union and Intersection Types. Proc. ACM Program. Lang. **unknown**(1, ICFP '17, Article 41) (sep 2017)
14. Castagna, G.: Covariance and Contravariance: a fresh look at an old issue. Tech. rep., CNRS (2016), `https://arxiv.org/pdf/1809.01427.pdf`
15. Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 198–199. PPDP '05, ACM, New

York, NY, USA (2005). https://doi.org/10.1145/1069774.1069793, http://doi.acm.org/10.1145/1069774.1069793
16. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional Characters of Solvable Terms. Mathematical Logic Quarterly **27**(2-6), 45–58 (1981). https://doi.org/10.1002/malq.19810270205, https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19810270205
17. Coppo, M., Dezani-Ciancaglini, M.: An extension of the basic functionality theory for the λ-calculus. Notre Dame Journal of Formal Logic **21**(4), 685–693 (1980). https://doi.org/10.1305/ndjfl/1093883253, https://doi.org/10.1305/ndjfl/1093883253
18. Costanza, P.: Closer project (2016), https://common-lisp.net/project/closer/closer-mop.html
19. (https://cs.stackexchange.com/users/93254/dan doel), D.D.: how to deduce a function subtype rule from a given function type definition. Computer Science Stack Exchange (2018), https://cs.stackexchange.com/q/97342, uRL:https://cs.stackexchange.com/q/97342 (version: 2018-09-16)
20. Doeraene, S.: Pseudo-union types in scala.js (Aug 2015), https://www.scala-js.org/news/2015/08/31/announcing-scalajs-0.6.5/
21. Dunfield, J.: Elaborating Intersection and Union Types. SIGPLAN Not. **47**(9), 17–28 (Sep 2012). https://doi.org/10.1145/2398856.2364534, http://doi.acm.org/10.1145/2398856.2364534
22. Fernandes, D., Bernardino, J.: Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In: Proceedings of the 7th International Conference on Data Science, Technology and Applications. p. 373–380. DATA 2018, SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT (2018). https://doi.org/10.5220/0006910203730380, https://doi.org/10.5220/0006910203730380
23. Frisch, A., Castagna, G., Benzaken, V.: Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM **55**(4), 1–64 (2008), extends and supersedes LICS '02 and ICALP/PPDP '05 articles
24. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 19:1–19:64 (Sep 2008). https://doi.org/10.1145/1391289.1391293, http://doi.acm.org/10.1145/1391289.1391293
25. Gabriel, R.P.: Common Lisp Email (Aug 1990)
26. Gabriel, R.P., White, J.L., Bobrow, D.G.: CLOS: integrating object-oriented and functional programming. Commun. ACM **34**(9), 29–38 (1991)
27. Gall, N.: Re: Types in CL. Computer History Museum Software Preservation Group (Dec 1987), https://cl-su-ai.cddddr.org/msg04610.html
28. Haible, B.: Gnu clisp (2010), https://clisp.sourceforge.io
29. Herda, M.p.: Implementing the Common Lisp condition system, pp. 135–219. Apress, Berkeley, CA (2020). https://doi.org/10.1007/978-1-4842-6134-7_3, https://doi.org/10.1007/978-1-4842-6134-7_3
30. Hosoya, H.: Regular Expression Types for XML. Ph.D. thesis, The University of Tokyo (Dec 2000)
31. Hosoya, H., Pierce, B.: Regular Expression Pattern Matching for XML. SIGPLAN Not. **36**(3), 67–80 (Jan 2001). https://doi.org/10.1145/373243.360209, http://doi.acm.org/10.1145/373243.360209
32. Jim Newton, Sébastien Doeraene, L.P.: Union types in scala 3 (feb 2020), https://contributors.scala-lang.org/t/union-types-in-scala-3/4046

33. Katzman, D.: Thread on SBCL devel-list sbcl-devel@lists.sourceforge.net (Dec 2015)
34. Keene, S.E.: Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS. Addison-Wesley (1989)
35. Kiczales, G.J., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA (1991)
36. MacQueen, D., Plotkin, G., Sethi, R.: An Ideal Model for Recursive Polymorphic Types. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 165–174. POPL '84, ACM, New York, NY, USA (1984). https://doi.org/10.1145/800017.800528, http://doi.acm.org/10.1145/800017.800528
37. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
38. Newton, J.: What does "argument types are not restrictive" mean? Thread on comp.lang.lisp (Aug 2018), https://groups.google.com/forum/#!topic/comp.lang.lisp/hgJNWfQOEDE
39. Norvig, P., Pitman, K.: Tutorial on Good Lisp Programming Style (aug 1993), https://www.cs.umd.edu/~nau/cmsc421/norvig-lisp-style.pdf
40. Odersky, M.: Dotty Documentation, 0.10.0-bin-SNAPSHOT (Aug 2018), http://dotty.epfl.ch/docs/reference/overview.html
41. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation, USA, 1st edn. (2008)
42. Paepcke, A.: User-Level Language Crafting – Introducing the CLOS metaobject protocol. In: Paepcke, A. (ed.) Object-Oriented Programming: The CLOS Perspective, chap. 3, pp. 65–99. MIT Press (1993), http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps, downloadable version at url
43. Pearce, D.J.: 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. pp. 117–130 (2017). https://doi.org/10.1145/3136040.3136042, http://doi.acm.org/10.1145/3136040.3136042
44. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
45. Pottinger, G.: A type assignment for the strongly normalizable lambda-terms. In: Hindley, J., Seldin, J. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 561–577. Academic Press (1980)
46. Rapoport, M., Lhoták, O.: Mutable Wadlerfest DOT. In: Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs. pp. 7:1–7:6. FTFJP'17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3103111.3104036, http://doi.acm.org/10.1145/3103111.3104036
47. Reynolds, J.C.: Design of the Programming Language Forsythe. Tech. rep., Unknown (1996)
48. Rhodes, C.: SBCL: A Sanely-Bootstrappable Common Lisp. In: Hirschfeld, R., Rose, K. (eds.) Self-Sustaining Systems. pp. 74–86. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
49. van Roggen, W.: Issue FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS Writeup (Sep 1988), http://www.lispworks.com/documentation/lw70/CLHS/Issues/iss176_w.htm
50. Rompf, T., Amin, N.: Type Soundness for Dependent Object Types (DOT). SIGPLAN Not. **51**(10), 624–641 (Oct 2016).

https://doi.org/10.1145/3022671.2984008, http://doi.acm.org/10.1145/3022671.2984008
51. Sabin, M.: Miles sabin via twitter, https://twitter.com/milessabin/status/953713425124818949?lang=en
52. Sabin, M.: Unboxed union types in Scala via the Curry-Howard isomorphism (Jun 2011), http://milessabin.com/blog/2011/06/09/scala-union-types-curry-howard/
53. Schafmeister, C.A., Wood, A.: Clasp common lisp implementation and optimization. In: Proceedings of the 11th European Lisp Symposium on European Lisp Symposium. ELS2018, European Lisp Scientific Activities Association (2018)
54. Steele, Jr., G.L.: Common LISP: The Language (1st Ed.). USA: Digital Press. With contributions by Scott E. Fahlman and Richard P (1984)
55. Steele Jr, G.L., Fahlman, S.E., Gabriel, R.P., Moon, D.A., Weinreb, D.L., Bobrow, D.G., DeMichiel, L.G., Keene, S.E., Kiczales, G., Perdue, C., et al.: Common LISP: The Language (2nd Ed.). Digital Press, Bedford, Massachusetts, Newton, MA, USA (1990)
56. Strandh, R.: Sicl: Building blocks for creators of common lisp implementations. (2013), http://metamodular.com/SICL/sicl-specification.pdf
57. Strandh, R.: Sicl: A New Common Lisp Implementation (2022), https://github.com/robert-strandh/SICL
58. Tobin-Hochstadt, S., Felleisen, M., Findler, R.B., Flatt, M., Greenman, B., Kent, A.M., St-Amour, V., Strickland, T.S., Takikawa, A.: Migratory Typing: Ten Years Later. In: SNAPL. pp. 17:1–17:17 (2017). https://doi.org/http://dx.doi.org/10.4230/LIPIcs.SNAPL.2017.17
59. various: Sbcl: General information (2022), https://github.com/sbcl/sbcl