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

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

Abstract

We summarize some aspects of the Common Lisp type system including a definition of types and type specifiers. 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, via the `subtypep` function. 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. 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.

CCS Concepts

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

ACM Reference Format:
Jim E. Newton. 2026. A History of Some Confusing and Entertaining Aspects of the Common Lisp Type System. In *Proceedings of The 13th European Lisp Symposium (ELS'20)*. ACM, New York, NY, USA, 9 pages. https://doi.org/not-yet-available

1 Types in Common Lisp

A detailed summary of the Common Lisp types with regard to peculiarities of function types can be found in [33]. We present here a shorter version of that exposé.

Definition 1.1 (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].

TODO explain why we augment the definition with *point of time*. 1. change-class, 2. redefinition of function effects (`satisfies f`), 3. type redefinition.

In Common Lisp, types and functions may be redefined. Also an object of a particular class may be victim to the `change-class` function. Both of these situations as well as several others may cause a type to change its members while a program is running.

Notation 1 (⊂)**.** We use the notation, $A \subset B$, ($A \supset B$) to indicate that A is either a strict subset (superset) of B or is equal to B. When we wish to designate a strict subset (superset) relation, we denote $A \subsetneq B$ ($A \supsetneq B$).

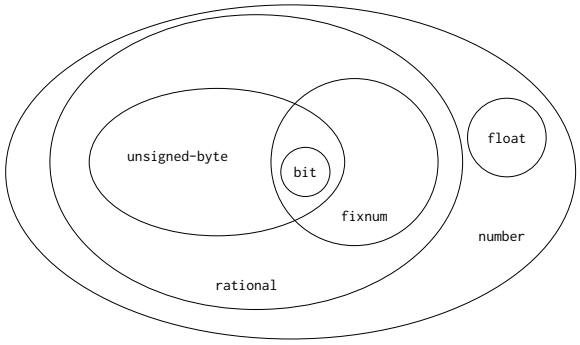

Figure 1: Some example Common Lisp types showing their intersection and subset relations

Notation 2 (⊥ and ⊤)**.** The symbol, ⊥, 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, ⊤, 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 Figure 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, (unsigned-byte ∩ fixnum) ≠ $\emptyset$. Types may be disjoint such as `float` and `fixnum`, (float ∩ fixnum) = $\emptyset$. Types may have a subtype relation such as `fixnum` and `number`, (fixnum ⊂ number). Types may have more complicated relations such as (bit ⊂ (fixnum ∩ unsigned-byte) ⊂ rational).

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 1.2 (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, Section 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

ELS'20, Zurich, Switzerland

ACM ISBN not-yet-available
https://doi.org/not-yet-available

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 given type. `typep` must always return `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 regard to certain function types, as explained in Section 4.7.

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:

> That `T1` is a subtype of `T2`, returning `T, T`
> That `T1` is not a subtype of `T2`, returning `NIL, T`,
> Or that subtype relationship cannot be determined, returning `NIL, NIL`

Section 3 discusses situations for which the subtype relationship cannot be determined.

2 Semantics 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, non-conforming 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. 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. Therefore, in our treatment of types we consider that type checking with `typep` is side-effect free, and in particular that it never signals an error. This assumption allows 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)`.

We assume, for this article, that no such problematic type specifier is used in the context of `typecase`.

3 Subtype relation Unanswerable

There is an important caveat. The `subtypep` function is not always able to determine whether the named types have a subtype relationship or not [4]. In such a case, `subtypep` returns `nil` as its second return value. This situation occurs most notably in the cases involving the `satisfies` type specifier. Consider Example 3.1 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 Example 3.1.

Example 3.1 (Problems with `satisfies`*).*

```
(typep 1 '(satisfies oddp))
==> T
(typep 0 '(satisfies oddp))
==> NIL
(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 Section 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 Example 3.2 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 advise of Norvig and Pitman [34, p 13] in its usage of `when` as a two-branch expression.

Example 3.2 (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 Example 3.1 it becomes a little easier if we define two types odd and even using `deftype`. We see very quickly that the system has difficulty reasoning about these types.

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

(deftype even ()
  '(and integer (satisfies evenp)))
==> EVEN

(subtypep 'odd 'even)
==> NIL, NIL

(subtypep 'odd 'string)
==> NIL, NIL
```

The `subtypep` function returns `nil` as its second value, indicating that SBCL[43][1] 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 `false` and `false` (among other reasons) when at least one argument involves type specifier `satisfies` [3].

SBCL cannot know 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.

Notice also that `(subtypep 'odd 'string)` returns `nil as second value` indicating that it was unable to determine whether odd is a subtype of `string`. At a glance it would seem that `(and integer (satisfies oddp))` is definitely not a subset of `string`, because `(and integer (satisfies oddp))` is a subset of `integer` which is disjoint from `string`. But there's a catch. The system does not know that odd and even are non-empty. If a type `A` is empty, then in fact `(and integer A)` is a subtype of `string` because $integer \cap A = \emptyset \subset string$ [29].

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 (Section 4.2).
(2) Function type specifiers in Common Lisp seem to violate the Induced Subtype Rule (ISR) (Section 4.3).
(3) The Common Lisp specification does not explicitly explain how to compute subset relations between function types (Section 4.6).
(4) Common Lisp programs are forbidden from making certain run-time function subtype checks (Section 4.7).

[1]SBCL is a Common Lisp compiler and is the implementation of Common Lisp used in the research for this article.

4.1 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 Section 4.2, the meaning of arrow types in Common Lisp is different from that in other common treatments.

Pierce [39, Ch 9] defines the arrow type formally with a syntax and rewriting (evaluation) semantics, such as

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

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

The meaning of these algebraic symbols is beyond the scope of this article, but suffice it to say that it defines 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.* [21, 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 4.1 (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 Definition 4.1 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 below is not in *integer* $\to$ *integer* according to Definition 4.1, because there exist integers, namely 0 and 1, for which the function fails to return an integer.

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

4.2 Subtypes of Functions

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) [18, 30, 37]. With this approach, we use the object system, Clos, 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, Section 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 1. 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.[2] 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 2 (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 2, we infer Definition 4.2, which we did not find anywhere explicitly stated in the Common Lisp specification.

Definition 4.2 (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 Definition 4.2 is that it is not general enough to apply to functions called indirectly such as:

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

The Common Lisp specification (still in [3, Section 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 Section 4.6, 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.

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

4.3 The Induced Subtype Rule (ISR)

The subtyping consequence, Corollary 4.3 follows from Definition 4.1.

Corollary 4.3. *If arrow types are defined as in Definition 4.1, then whenever $B \subset A$, we have $(A \to Y) \subset (B \to Y)$.*

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

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

Section 4.5 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 [50]. 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 (Section 4.3) we see that a principle known as ISR (Claim 3) fails, and in Section 4.6 we see that certain intuitions regarding type intersection fail.

Claim 3 (Induced subtype rule, ISR)**.**
If $A \subset B$ and $X \subset Y$, then $B \to X$ is a subtype of $A \to Y$, denoted $(B \to X) \subset (A \to Y)$.

Claim 3 is not actually a principle supported in Common Lisp rather it is an assumption some users make as is further discussed in Section 4.8. The claim, which is a generalization of Corollary 4.3, is basically saying that function types are (should be) covariant with respect to return type and contravariant with respect to argument type. Castagna [14, Section 2.2] provides a proof of this claim and for a weaker version of its converse.

Since function type specifiers are valid arguments to the `subtypep` function, we can test Claim 3 using SBCL 1.4.10 as in Examples 4.4 and 4.5. In Example 4.4, we see results which agree with the ISR. We see that since $fixnum \subset number$, 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 4.4 (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 Example 4.4, in Example 4.5 we see that subtypep does not respect ISR with respect to contravariance of function arguments. The subtype function, at least in SBCL, believes $(fixnum \to \top) \subset (number \to \top)$ and $(number \to \top) \not\subset (fixnum \to \top)$.

Example 4.5 (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
```

4.4 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 thought they seem to violate the definition of arrow types from Definition 4.2. First we derive a property of types $\top \to \bot$ and $\bot \to \top$ in Corollary 4.7, then test the predictions in Example 4.8.

A consequence of Claim 3 is shown in Theorem 4.6, and we can test the predictions of the theorem as shown in Example 4.8.

Theorem 4.6. *Given types A, B, X, and Y,*

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

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

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

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

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

So together we have

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

□

In this section we take a brief look at the degenerate cases of Theorem 4.6.

Corollary 4.7. $(\top \to \bot) \subset (\bot \to \top)$.

Proof. Let $B = \neg A$ and $X = \neg Y$, and apply Theorem 4.6. □

In Example 4.8 we test Corollary 4.7. 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 4.8 (Testing Corollary 4.7 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 below that neither `(function (nil) t)` nor `(function (t) nil)` is empty as neither is a subtype of `nil`.

```
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
```

We further see in the next example that the two types, `(function (nil) t)` and `(function (t) nil)`, are non-disjoint (*i.e.* intersecting) types.

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

By Definition 4.2, `(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 4.9 shows that in Common Lisp we have $\bot \to \bot \subsetneq \top \to \top$, a strict subset relation.

*Example 4.9 (*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 Definition 4.2. `(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 Example 4.9 also shows, $\bot \to \bot \subsetneq \top \to \bot$, *i.e.* a strict subset relation.

4.5 Intuition of function type intersection

The examples and results shown in Sections 4.3 and 4.4 may come as a surprise to some readers, as they seem in violation of ISR. The theory of semantic subtyping [22] 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 Section 5, 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 1 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 1 would be erroneous according to the call-site rewriting rule (Claim 2). 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.[3] 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 (function ((or ratio integer)) integer) = (function (rational) integer). One would infer the rule $(A \to Y) \cap (B \to Y) = A \cup B \to Y$. However, as explained in Section 4.6, the Common Lisp type system works differently in relation to function type intersection.

4.6 Calculation of function subtype relation is underspecified

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

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

Doel [27] points out that the two are not in explicit contradiction. However, the intersection in Claim 4 is different from what we would guess from the ISR (Claim 3), which says that

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

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.

4.7 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.

```
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 regretably) 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.

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

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

`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 false and false *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.

4.8 A historical perspective

Concerning the inconsistencies mentioned in Section 4.7, it would seem that at least one of the contributors to the X3J13 cleanup issue, FUNCTION-TYPE-ARGUMENT-TYPE--SEMANTICS:RESTRICTIVE [50] 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 [50]. This issue discusses the problem that the function argument type semantics are different from what users expect. It is mentioned that CLtL [48, 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, $list \subset cons$; but contravariant arguments, $float \subset \top$ and $string \subset \top$.

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 [5, 23] 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 $T_1 \subset T_2 \subset T_3$, then $(T_3 \rightarrow T_1) \subset (T_2 \rightarrow T_2)$ and $(T_1 \rightarrow T_3) \not\subset (T_2 \rightarrow T_2)$. 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 [6, 7, 9] 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 [23, 24], 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 [47] 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.

5 Related work

Tobin *et al.* [49] 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 [20] discusses works related to intersection types, but omits explicit references to the Common Lisp type system. He does mention the Forsythe [42] 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 [40] 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.* [31].

As in Common Lisp, languages which support intersection and union types [13, 15, 38], should also be consistent with respect to subtype relations. Frisch *et al.* [22] 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 [36] 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 [46] discusses user level extensions to the Scala type system to support intersection and union types. However, Sabin [45] also recommends that no one use his implementation in *real code*. Doeraene [19, 28] introduces pseudo-union types in Scala.js. Scala 3 [1, 35, 44] includes support for intersection and union types in a type lattice, but not complementary types.

In Section 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 [32]. Finally, the Scala language [36, 41], allows programmers to decide within function and class declaration, which parameters are covariant, contravariant, or invariant.

6 Perspectives

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 [25, 26] simplify their model of semantic subtyping by modeling a type as a set of values of the language.

References

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

[2] Davide Ancona, Giuseppe Castagna, Tommaso Petrucciani, and Elena Zucca. 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] 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.

[5] Jeff Barnett. Types in CL. Computer History Museum Software Preservation Group, December 1987. URL https://cl-su-ai.cddddr.org/msg04614.html.

[6] Jeff Barnett. The CRISP Programming Language System, An Historical Overview. Computer History Museum Software Preservation Group, September 2009. URL http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/CRISP\protect\discretionary{\char\hyphenchar\font}{}{}talk.pdf.

[7] Jeff Barnett. Notes on SDC CRISP for IBM 370 Computers. Computer History Museum Software Preservation Group, June 2010. URL http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/notes/.

[8] Jeff Barnett. searching for Jeff Barnett. conversation on comp.lang.lisp, August 2018. URL https://groups.google.com/d/msg/comp.lang.lisp/QLUWlAqejFA/5nvIEyFjAwAJ.

[9] Jeff Barnett and D. L. Pintar. CRISP: A Programming language and System (draft), December 1974. URL http://www.softwarepreservation.org/projects/LISP/crisp_ibm370_sdc/TM-5444_000_00.pdf.

[10] 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.

[11] Pascal J. Bourguignon. I'm annoyed by the specification for satisfies. Thread on comp.lang.lisp, January 2017. URL https://groups.google.com/d/msg/comp.lang.lisp/w4bYVI7ieoA/Tn44-6coGgAJ.

[12] Robert Brown and Francoise-Rene Rideau. Google Common Lisp Style Guide, Revision 1.28, 2018. URL https://google.github.io/styleguide/lispguide.xml. accessed 14 October 2018, 12h36 +0200.

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

[14] Giuseppe Castagna. Covariance and Contravariance: a fresh look at an old issue. Technical report, CNRS, 2016. URL https://arxiv.org/pdf/1809.01427.pdf.

[15] 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.

[16] 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.

[17] 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.

[18] Pascal Costanza. Closer project, 2016. URL https://common-lisp.net/project/closer/closer-mop.html.

[19] 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/.

[20] 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.

[21] A. Frisch, G. Castagna, and V. Benzaken. 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.

[22] 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.

[23] Richard P. Gabriel. Common Lisp Email, August 1990.

[24] Nick Gall. Re: Types in CL. Computer History Museum Software Preservation Group, December 1987. URL https://cl-su-ai.cddddr.org/msg04610.html.

[25] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, December 2000.

[26] Haruo Hosoya and Benjamin Pierce. Regular Expression Pattern Matching for XML. *SIGPLAN Not.*, 36(3):67–80, January 2001. ISSN 0362-1340. doi: 10.1145/373243.360209. URL http://doi.acm.org/10.1145/373243.360209.

[27] Dan Doel (https://cs.stackexchange.com/users/93254/dan doel). how to deduce a function subtype rule from a given function type definition. Computer Science Stack Exchange, 2018. URL https://cs.stackexchange.com/q/97342. URL:https://cs.stackexchange.com/q/97342 (version: 2018-09-16).

[28] 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.

[29] Douglas Katzman. Thread on SBCL devel-list sbcl-devel@lists.sourceforge.net, December 2015.

[30] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[31] 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.

[32] Jim Newton. What does "argument types are not restrictive" mean? Thread on comp.lang.lisp, August 2018. URL https://groups.google.com/forum/#!topic/comp.lang.lisp/hgJNWfQOEDE.

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

[34] Peter Norvig and Kent Pitman. Tutorial on Good Lisp Programming Style. aug 1993. URL https://www.cs.umd.edu/~nau/cmsc421/norvig-lisp-style.pdf.

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

[36] 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.

[37] 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.

[38] 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.

[39] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

[40] 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.

[41] Marianna Rapoport and Ondřej Lhoták. Mutable Wadlerfest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs*, FTFJP'17, pages 7:1–7:6, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5098-3. doi: 10.1145/3103111.3104036. URL http://doi.acm.org/10.1145/3103111.3104036.

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

[43] 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.

[44] 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.

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

[46] 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/.

[47] Guy L. Steele, Jr. *Common LISP: The Language (1st Ed.)*. USA: Digital Press. With contributions by Scott E. Fahlman and Richard P, 1984. ISBN 0-932376-41-X.

[48] Guy L Steele Jr, Scott E Fahlman, Richard P Gabriel, David A Moon, Daniel L Weinreb, Daniel G Bobrow, Linda G DeMichiel, Sonya E Keene, Gregor Kiczales, Crispin Perdue, et al. *Common LISP: The Language (2nd Ed.)*. Digital Press, Bedford, Massachusetts, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[49] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In *SNAPL*, pages 17:1–17:17, 2017. doi: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2017.17.

[50] Walter van Roggen. Issue FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS Writeup, September 1988. URL http://www.lispworks.com/documentation/lw70/CLHS/Issues/iss176_w.htm.