Covariance and Contravariance: a fresh look at an old issue
(a primer in advanced type systems for learning functional programmers)

Giuseppe Castagna
CNRS
Laboratoire Preuves, Programmes et Systèmes
Université Paris Diderot – Paris 7

Abstract

Twenty years ago, in an article titled “Covariance and contravariance: conflict without a cause”, I argued that covariant and contravariant specialization of method parameters in object-oriented programming had different purposes and deduced that, not only could they, but actually they should both coexist in the same language. In this work I reexamine the result of that article in the light of recent advances in (sub)-typing theory and programming languages, taking a fresh look at this old issue.

Actually, the revamping of this problem is just an excuse for writing an essay that aims at explaining sophisticated type theoretic concepts, in simple terms and by examples, to undergraduate computer science students and/or willing functional programmers.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—object-oriented languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—type structure

General Terms Theory, Languages

Keywords Object-oriented languages, type theory, subtyping, intersection types, overloading.

1. Introduction

Twenty years ago I wrote an article titled “Covariance and contravariance: conflict without a cause” [10] where I argued that the heated debate that at the time opposed the faction of covariant overriding of methods in object-oriented languages against the congregation of the contravariant specialization had no real ground, since the two policies had different orthogonal purposes that not only could but actually should coexist in the same language. The article was, I’d dare to say, fairly successful even outside the restricted research circles. For instance for many years at the entry “Contra-/Co-variance” of FAQ of the comp.object Usenet newsgroup (question 71) the answer was just a pointer to my article (actually to the tech-rep that preceded the publication). In spite of that, I think that the message of the article did not (or at least could not) reach practitioners and thus influence actual programmers. In this work I reexamine the result of that article in the light of recent advances in (sub-)typing theory and programming languages, taking a fresh look at this old issue.

Actually, the revamping of this problem is just an excuse for writing an essay that aims at explaining sophisticated type theoretic concepts, in simple terms and by examples, to undergraduate computer science students and/or willing functional programmers.

A reader aware of the theory of semantic subtyping may be astonished that I use such a theory to target practitioners. As a matter of fact, semantic subtyping is a sophisticated theory that relies on complex set-theoretic properties that, for sure, are not accessible to most practitioners. The point is that while the underlying theory of types is out of reach of an average programmer, its types are very easy to use and understand for this programmer since, as I show in this paper, they can be explained in terms of very simple notions such as sets of values and set containment. It is like cars. Thirty years ago most cars had such a simple conception that nearly everybody with some experience and few common tools could open the trunk and fix them. Nowadays cars are so full of electronics that for many of them you must go to authorized dealers to have it repaired since this is out of reach for generic repairers. All this complexity is however hidden to the end-user, and cars today are much simpler to drive than they were thirty years ago. So it is for type systems, whose definitions are getting more and more involved but (in several cases) they are getting simpler and simpler for the programmer to use.

For what concerns the Perl 6 language, I am not a great user or supporter of it (the examples I give here count among the most complicated programs I wrote in it). Although it probably does not have the most elegant (and surely not the most streamlined) syntax I ever saw in a programming language, I chose it because it has the double advantage of having enough syntax to explain the covariance/contravariance problem and of having a fuzzy-yet-to-be-fixed type system. While to illustrate all the examples I chose to use the programming language Perl6 [30], but an important aspect of this paper is that you can read it without any preliminary knowledge of them.

Twenty years have passed, both programming languages and even more type theory have much evolved to a point that I think it is now possible to explain covariance and contravariance to practitioners, a task the first half of this article is devoted to.

To do that I will use the type theory of semantic subtyping [23], while to illustrate all the examples I chose to use the programming language Perl6 [30], but an important aspect of this paper is that you can read it without any preliminary knowledge of them.

A reader aware of the theory of semantic subtyping may be astonished that I use such a theory to target practitioners. As a matter of fact, semantic subtyping is a sophisticated theory that relies on complex set-theoretic properties that, for sure, are not accessible to most practitioners. The point is that while the underlying theory of types is out of reach of an average programmer, its types are very easy to use and understand for this programmer since, as I show in this paper, they can be explained in terms of very simple notions such as sets of values and set containment. It is like cars. Thirty years ago most cars had such a simple conception that nearly everybody with some experience and few common tools could open the trunk and fix them. Nowadays cars are so full of electronics that for many of them you must go to authorized dealers to have it repaired since this is out of reach for generic repairers. All this complexity is however hidden to the end-user, and cars today are much simpler to drive than they were thirty years ago. So it is for type systems, whose definitions are getting more and more involved but (in several cases) they are getting simpler and simpler for the programmer to use.

For what concerns the Perl 6 language, I am not a great user or supporter of it (the examples I give here count among the most complicated programs I wrote in it). Although it probably does not have the most elegant (and surely not the most streamlined) syntax I ever saw in a programming language, I chose it because it has the double advantage of having enough syntax to explain the covariance/contravariance problem and of having a fuzzy-yet-to-be-fixed type system. While to illustrate all the examples I will write you can read it without any preliminary knowledge of them.

1 Though at that time I did not succeed to convince my parents to let me repair our family car.
best candidate to present this paper (the perfect candidate would be the programming language CDuce [3, 18] whose type system is here borrowed and grafted on Perl) one of the challenges of this paper was to use a mainstream language that was not designed with types in mind—far from that—, whence the choice of Perl. I am aware that this choice will make some colleague researchers sneer and some practitioners groan: please give me the benefit of the doubt till Section 4.

**Plan of the article.** I organize the rest of the paper as if it were the documentation bundled with a TV set. When you buy a TV you want to read the instructions on how to tune channels and connect it to your Wii™, these are in the user manual. You usually skip the electrical schemes that come with it, unless you want to repair it, or to build your own TV from them. Besides, if you are curious to know why these electrical schemes show you nightly news instead of exploding and killing everybody in the room, then you probably are a researcher and you need to read few articles and books on electronics and electromagnetism for which the bundled documentation is useless.

Section 2 is my “user manual”: it is a primer on types for a Perl programmer who never used types (seriously) before. There I use (a personalized version of) Perl 6 types to explain what types are and how they are related in the “semantic subtyping” framework. Although most of the notions will be known to most of the readers, the purpose of the presentation is to demonstrate that with few easy-to-grasp key concepts, it is possible to bring programmers to the value is that few years ago I started to teach an undergraduate course on advanced programming at École Normale Supérieure de Paris. I try to summarize the lessons that can be drawn from this work and, in particular, what I learnt from writing it. The purpose of the presentation is to demonstrate that with few easy-to-grasp key concepts, it is possible to bring programmers to an advance comprehension of the power of types. Section 6 I try to summarize the lessons that can be drawn from this work and, in particular, what I learnt from writing it.

Although most of the notions will be known to most of the readers, the purpose of the presentation is to demonstrate that with few easy-to-grasp key concepts, it is possible to bring programmers to the value is that few years ago I started to teach an undergraduate course on advanced programming at École Normale Supérieure de Paris. I try to summarize the lessons that can be drawn from this work and, in particular, what I learnt from writing it. The purpose of the presentation is to demonstrate that with few easy-to-grasp key concepts, it is possible to bring programmers to an advance comprehension of the power of types. Section 6 I try to summarize the lessons that can be drawn from this work and, in particular, what I learnt from writing it.

2. **Types primer for the learning programmer**

2.1 What is a type?

If you know what a value is (i.e., any result that can be returned by an expression), then you know what a type is: it is a set of values (though not all sets of values are types). Then you also know what subtyping is since it coincides with set containment: a type is a subtype of a second type if all values of/in the former type are values of/in the latter. What a type system does is to define a typing relation that is a relation between expressions and types. This relation must have the property that an expression (i.e., it is related to) a type only if whenever it returns a value, then the value is of/in that type.

Let me give some more details. I will consider a very restricted syntax for Perl (types and actually some of these types where just proposed but never implemented) as described by the following grammar:

$$
T ::= \text{Bool} \mid \text{Int} \mid \text{Any} \mid (T, T) \mid T\to T \mid \text{not}(T) \mid T\to\to T
$$

What does each type mean? To define the meaning of a type, we define the set of values it denotes. Since the types above are defined inductively we can define their precise meaning by induction, namely:

- **Bool** denotes the set that contains just two values {true, false}
- **Int** denotes the set that contains all the numeric constants: {0, -1, 1, -2, 2, -3, ...}.
- **Any** denotes the set that contains all the values of the language.
- $$(T_1, T_2)$$ denotes the set that contains all the possible pairs $$(v_1, v_2)$$ where $$v_1$$ is a value in $$T_1$$ and $$v_2$$ a value in $$T_2$$, that is $$\{(v_1, v_2) \mid v_1 \in T_1, v_2 \in T_2\}$$.
- $$T_1\lor T_2$$ denotes the union of the sets denoted by $$T_1$$ and $$T_2$$, that is the set $$\{v \mid v \in T_1 \text{ or } v \in T_2\}$$.
- $$T_1\land T_2$$ denotes the intersection of the sets denoted by $$T_1$$ and $$T_2$$, that is the set $$\{v \mid v \in T_1 \text{ and } v \in T_2\}$$.
- $$\text{not}(T)$$ denotes the set of all the values that are not in the set denoted by $$T$$, that is $$\{v \mid v \notin T\}$$.

In particular not(Any) denotes the empty set.

If we consider a value in $$T_1$$, if they return a value (i.e., if the application does not loop), then this value is in $$T_2$$.

Of course, the last case is the most delicate one and deserves much more explanation. First of all we must define what a “functional value” is. A functional value is a (closed) expression that defines a function. In λ-calculus we would say it is a lambda abstraction. In Perl it is any expression of the form “sub (parameters) {body}”.

2 For the reader not aware of the notation that follows (called EBNF grammar), this is a standard way to define the syntax of some language (in our case, the language of types). It can be understood by reading the symbol “:::=" as “is either” and the symbol “|” as “or”. This yields the following reading: “A type $$T$$ is either $$\text{Bool}$$ or $$\text{Int}$$ or $$\text{Any}$$ or $$(T_1, T_2)$$ (where $$T_1$$ and $$T_2$$ are types) or $$T_1\land T_2$$ (where $$T_1$$ and $$T_2$$ are types) or ...”. Notice that to define, say, $$(T_1, T_2)$$ we assumed to know what the types $$T_1$$ and $$T_2$$ that compose it are: this roughly corresponds to define types by induction.

3 A function is closed if all the variables that appear in its body are either the parameters of the function or they have been defined inside the body.
value that denotes the successor function. Functions can be named (which will turn out to be quite handy in what follows) such as for\n\n\n```
sub succ("int $x") { return $x + 1 }
```

sub (which will turn out to be quite handy in what follows) such as for value that denotes the successor function. It is easy to see that succ is a value in Int---{Int} it suffices to apply the definition I gave for arrow types, that is, check that when succ is applied to a value in Int, it returns a value in Int (succ is total).

2.2 What is a subtype?

Perl 6 provides the constructor `subset` to define types such as `Int. Even` and `Odd`, respectively denoting the set of natural, even, and odd numbers. For instance the last two types can be defined as follows (where % in Perl 6 is the modulo operator):

```perl
subset Even of Int where { $%. 2 == 0 }
subset Odd of Int where { $%. 2 == 1 }
```

(e.g., the first is Perl syntax for `Even = \{ x ∈ Int | x mod 2 = 0 \}`). Both these types are subsets of Int.  Let us use these types to show that a same value may belong to different types. For instance, it is easy to see that succ is also a value in `Even→Int`: if we apply succ to a even number, it returns a value in `Int`. Since succ is a value in `Even→Int` and in `Int→Int`, then, by definition of intersection type, it is a value in `Even→Int`& `(Int→Int)`. This is true not only for succ but for all values in `Int→Int`: whenever we apply a function in `Int→Int` to an even number, if it returns a value, then this value is in Int. We say that `Int→Int` is a subtype of `Even→Int` and write it as `Int→Int <: Even→Int`. So, as I formally said at the beginning of the section, subtyping is just set containment on values.

DEFINITION 2.1 (subtype). A type `T1` is a subtype of a type `T2`, written `T1 <: T2`, if all values in `T1` are values in `T2`.

While `Int→Int<:Even→Int` holds, the converse (i.e. `Even→Int <: Int→Int`) does not. For instance the division-by-two function is a value in `Even→Int` but not in `Int→Int` (for Perl’s purists: notice that I omitted the return keyword: since in Perl 6 it is optional, in this section I will systematically omit it). Actually, since we have that `Int→Int<:Even→Int`, then stating that succ has type `Even→Int`& `(Int→Int)` does not bring any further information: we are intersecting a set with a superset of it, therefore this intersection type is equivalent to (i.e., it denotes the same set of values as) `Int→Int`. To see an example of a meaningful intersection type, the reader can easily check that the succ function is in `(Even→Odd)`& `(Odd→Even)`: succ applied to an even number returns an odd number (so it is in `Even→Odd`) and applied to an odd number it returns an even one (so it is also in `Odd→Even`) and. This brings useful information since `(Even→Odd)`& `(Odd→Even)`<: Int→Int is a strict containment, that is, there are values of the type on the right hand-side of `"<"` that do not have the type on the left hand-side of the relation (e.g., the identity function is in `Int---{Int}`, but it has neither type `Even---Odd`, nor type `Odd---Even`).

The subtyping relation above holds not only for `Even` and `Odd`, but it can be generalized to any pair of arbitrary types. Since `Int` is equivalent to `Odd`|`Even` (i.e., the two types denote the same set of values), then the last subtyping relation can be rewritten as `(Even---Odd)`& `(Odd---Even)`<: `(Odd`|`Even`)---`(Odd`|`Even)` which is just an instance of the following relation

\[
(S_1 -> T_1) & (S_2 -> T_2) <: (S_1 | S_2) -> (T_1 | T_2) \tag{1}
\]

4 In Perl variables are prefixed by the dollar sign, and `sub` is the apocope for subroutine.

5 More precisely, I should have said “they denote subsets”, but from now on I identify types with the sets of values they denote.

that holds for all types, `S_1`, `S_2`, `T_1`, and `T_2`, whatever these types are. A value in the type on the left hand-side of `(1)` is a function that if applied to an `S_1` value it returns (if any) a value in `T_1`, and if it is applied to a value in `S_2` it returns (if any) a value in `T_2`. Clearly this is also a function that when applied to an argument that is either in `S_1` or in `S_2`, it will return a value either in `T_1` or in `T_2` (from now on I will omit the “if any” parentheses). I leave as an exercise to the reader [EX1] the task to show that the inclusion is strict, that is, that there exists a value in the type on the right-hand side of `(1)` that is not in the type on left-hand side (see solutions in the appendix).

This shows how some simple reasoning on values, allows us to deduce sophisticated subtyping relations. By a similar reasoning we can introduce two key notions for this paper, those of covariance and of contravariance. These two notions come out as soon as one tries to deduce when a type `S_1→T_1` is included in another function type `S_2→T_2`. Let us first try to deduce it on a specific example: consider the function `sub double(Int $x) { 2 * x }`. This function is of type `Int→Int`. It is easy to see that `double` (but also any other function in `Int→Int`) is also a function in `Int→Int` since whenever it returns an `Int`, it therefore returns an `Int`. If `double` returns an `Int` when applied to an `Int`, then in particular `double` returns an `Int` when it is applied to, say, an `Odd` number, that is, `double` is a function in `Odd→Int`. Since this reasoning holds true for every function in `Int→Int`, then we can deduce that `Int→Int<:Odd→Int`. This relation holds not only for `Int`, `Even`, and `Odd`, but for every pair of types whose sub-components are in a similar relation. Every function value in `S_1→T_1` transforms values in `S_1` into values in `T_1`, then it is also a function that transforms values in any type `S_2` smaller than `S_1` into values that are contained in `T_1` and, thus, in any type `T_2` greater than `T_1`. Since these two conditions are also necessary for containment, then we have following rule:

\[
(S_1 -> T_1) <: (S_2 -> T_2) \iff S_2 <: S_1 \land T_1 <: T_2 \tag{2}
\]

We notice that while the “T” types have the same positions with respect to <: in both sides of equation (2), the “S” types have a different position on the right and left sides of (2). Borrowing the terminology from category theory, it is customary to say that the function type constructor "<--\st" is covariant on the codomain type—since it preserves the direction of the <: relation—and, is contravariant on the domain type—since it inverts the direction of the <: relation—.

As a final remark, note that while in general it is easy to check whether a value is in some type (just apply the rules of Section 2.1), it is usually quite harder to decide whether a type is contained into another. However while a programmer should clearly master the former problem, she should be just mildly worried by the latter. Of course she must be able to use in her programming practice some generic subtyping rules such as (2) and at least the following (pretty straightforward) ones:

\[
(S_1, T_1) <: (S_2, T_2) \iff S_1 <: S_2 \land T_1 <: T_2 \tag{3}
\]

\[
S_1 <: S_2 \land T_1 <: T_2 \implies S_1 & T_1 <: S_2 & T_2 \tag{4}
\]

\[
S_1 <: S_2 \land T_1 <: T_2 \implies S_1 | T_1 <: S_2 | T_2 \tag{5}
\]

\[
S <: T \iff \text{not}(T) <: \text{not}(S) \tag{6}
\]

However she is not required to be able to decide subtyping for every possible pair of types, since this requires a knowledge deeper

\footnote{For the advanced reader, in the presence of an empty type, the two conditions on the right-hand side of (2) are sufficient but not necessary. If we use Empty to denote not (Any)— i.e., the empty type—, then the correct rule is: (S1 -> T1) <: (S2 -> T2) if and only if (S1 <: S2) and T1 <: T2 or (S1 <: Empty). Likewise in the presence of an empty type the right-hand side condition of (3) later on are not necessary: (S1, T1) <: (S2, T2) if and only if (S1 <: Empty) or (T1 <: Empty) or (S1 <: S2) and T1 <: T2.}
than the few above rules (as an aside, notice that the rules (2)–(6) are not enough to deduce the subtyping relation in (1); technically one says that the above set of rules provides a sound but incomplete axiomatization of the subtyping relation\(^7\)). The problem of deciding subtyping between two generic types is a task for the language designer and implementer. This is the one who must implement the algorithms that not only check the containment of two generic types, but also generate informative error messages that explain the programmer the reasons why some containment she used does not hold. In Section 4 I will explain for the language designer how to do it.

### 2.3 Intersections, overloading, and dispatching

The cognoscente reader will have recognized in the intersection of arrow types such as \((S_1 \to T_1) \& (S_2 \to T_2)\) the type typical of overloaded functions: a function of this type returns results of different types according to the type of its argument. Strictly speaking, an overloaded function is a function that executes different code for arguments of different types; the semantics of the function is “overloaded” since formed of different semantics each corresponding to a different piece of code. The only function we have seen so far with an intersection type —i.e., \(\text{succ}—\) is not really “overloaded” since it always executes the same code both for arguments in \(\text{Even}\) and for arguments in \(\text{Odd}\). In this case it would be more correct to speak of behavioral type refinement since intersections of arrows do not correspond to different pieces of code (each implementing different functions all denoted by the same operator) but, instead, they provide a more precise description of the function behavior.\(^8\) However Perl 6 allows the programmer to define “real” overloaded functions by giving multiple definitions of the same function prefixed by the multi modifier:

\[
\begin{align*}
\text{multi sub } & \text{sum}(\text{Int } x, \text{Int } y) \{ \text{x } + \text{ y } \} \\
\text{multi sub } & \text{sum}(\text{Bool } x, \text{Bool } y) \{ \text{x } \&\text{ y } \}
\end{align*}
\]

Here the \(\text{sum}\) function has two different definitions: one for a pair of integer parameters (in which case it returns their \(\text{sum}\)) and one for a pair of Boolean parameters (in which case it returns its logical “and” denoted in Perl by \&). For instance, \(\text{sum}(37,5)\) returns 42 and \(\text{sum}(\text{True, False})\) returns False. Clearly the function \(\text{sum}\) above has the following type

\[
\begin{align*}
(\text{Int, Int}) \to \text{Int} & \land (\text{Bool, Boolean}) \to \text{Boolean},
\end{align*}
\]

which states that if the function is applied to a pair of integers, then it returns integer results, and that if it is applied to a pair of Boolean arguments it returns Boolean results. The actual code to be executed is chosen at run-time according to the type of the actual parameters. Although in the case above both parameters are taken into account to select the code, it is clear that checking the type of just one parameter—e.g., the first one—would have sufficed. So I could have equivalently written the \(\text{sum}\) function in a curried\(^9\) version I call \(\text{sum}C\):

\[
\begin{align*}
\text{multi sub } & \text{sumC}(\text{Int } x) \{ \text{sub } (\text{Int } y) \{ \text{x } + \text{ y } \} \}
\end{align*}
\]

\(^7\) To deduce such a kind of relations one should also use axioms such as the following ones (excerpted from [2]):

\[
\begin{align*}
(\to \&) & \land (\to \cup) & \subseteq & (\to \& \to \cup) \quad (7) \\
(\to \cup) & \lor (\to \cup) & \subseteq & (\to \cup \to \cup) \quad (8)
\end{align*}
\]

The reader may try as an exercise to prove that both the relations in (7) (8) above and their converse hold [EX2].

\(^8\) Such a usage of intersection types is known —in my opinion, somehow misleadingly—as coherent overloading; see Section 1.1.4 in [31].

\(^9\) In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument.

The syntax above makes it clear that only the type of the parameter \(x\) is used to select the code, while the second parameter is just an argument to be passed to the function returned by the selection. Perl 6 provides a very handy double-semicolon syntax \(;;\) to distinguish parameters that are used for the selection of a multi-function (those on the left of \(;;\)) from those that are just passed to the selected code (those on the right of \(;;\)). So by using the syntactic sugar \(;;\); the code above can be “equivalently” rewritten as:

\[
\begin{align*}
\text{multi sub } & \text{sumC}(\text{Int } x ; ; \text{Int } y) \{ \text{x } + \text{ y } \} \\
\text{multi sub } & \text{sumC}(\text{Bool } x ; ; \text{Bool } y) \{ \text{x } \&\text{ y } \}
\end{align*}
\]

The type of the \(\text{sum}\) function has changed since in both \(\text{multi}\) definitions (either curried or with \(;;\);) \(\text{sumC}\) has type:

\[
\begin{align*}
(\text{Int} \to (\text{Int} \to \text{Int})) & \& (\text{Bool} \to (\text{Bool} \to \text{Bool})),
\end{align*}
\]

The functions of this type are different from those of the type in (9): the type in (9) is the set of all functions that when applied to a pair of integers return an integer and when applied to a pair of Booleans they return a Boolean. The functions in the type of (10) are functions that when applied to an integer return a function from integers to integers, and when applied to a Boolean they return a function from Booleans to Booleans.

Currying requires some care when all arguments are used to select the code to execute. Imagine that I wanted to add the code to handle the cases when the two arguments are of different types and that, therefore, I added two \(\text{multi}\) definitions for \(\text{sum}\) which is then defined in the following (weird) recursive way:

\[
\begin{align*}
\text{multi sub } & \text{sum}(\text{Int } x, \text{Int } y) \{ \text{x } + \text{ y } \} \\
\text{multi sub } & \text{sum}(\text{Bool } x, \text{Bool } y) \{ \text{x } &\text{ y } \} \\
\text{multi sub } & \text{sum}(\text{Bool } x, \text{Int } y) \{ \text{sum}(\text{x}, \text{y})\} \\
\text{multi sub } & \text{sum}(\text{Int } x, \text{Bool } y) \{ \text{sum}(\text{y}, \text{x}) \}
\end{align*}
\]

As a consequence \(\text{sumC}\) has now the following type (the reader is invited to check the typing as an exercise [EX3])

\[
\begin{align*}
(\text{Int, Int}) \to \text{Int} & \land (\text{Bool, Boolean}) \to \text{Boolean} \\
& \land (\text{Bool, Bool}) \to \text{Boolean} \\
& \land (\text{Int, Bool}) \to \text{Boolean}.
\end{align*}
\]

It is clear that if we want to dispatch (i.e., perform the selection) only on the first argument, then the selection must return a nested \(\text{multi}\) function that will dispatch on the second argument. That is, we are looking for a function of the following type:

\[
\begin{align*}
(\text{Int} & \to (\text{Int} \to \text{Int}) & \& (\text{Bool} \to \text{Boolean})) \\
& \& (\text{Bool} \to (\text{Bool} \to \text{Boolean}) & \& (\text{Int} \to \text{Bool})).
\end{align*}
\]

The type above is the type of functions that when they are applied to an integer return another function of type \((\text{Int} \to \text{Int}) & \& (\text{Bool} \to \text{Boolean})) and that when they are applied to a Boolean return another function of type \((\text{Int} \to \text{Int}) \& (\text{Bool} \to \text{Boolean})\). In other terms they are functions that can be applied to a sequence of two arguments of type \text{Int} or \text{Bool}: if the first argument is an integer then the result of the double application will be of the same type as the type of the second argument; if the first argument is a Boolean then also the result will be a Boolean.\(^11\)

\(^10\) Strictly speaking this and the previous definition of \(\text{sumC}\) are not equivalent since the first application is curried and therefore it must be fed by one argument at a time (e.g., \(\text{sumC}(3)(39)\)) while in the latter it is not, so it receives all the arguments at once (e.g., \(\text{sumC}(3,39)\)) the second notation forbids partial application. Here I just focus on the types and not on the particular syntax, so I will sweep such differences under the carpet.

\(^11\) By equation (8) and the exercise in Footnote 7, the type in (12) is equivalent to:
How is a function of the type in (12) defined? In particular how can we define \texttt{sumC}—the curried version of the second definition of \texttt{sum}—which should have this type? A first attempt consists in defining two auxiliary \texttt{multi} functions \texttt{sumI} and \texttt{sumB} that perform the dispatch on the second argument of \texttt{sumC} and that are called according to whether the first argument of \texttt{sumC} is an integer or a Boolean, respectively:

\begin{verbatim}
multi sub sumI(Int $y; ; Int $x){ $x + $y }
multi sub sumB(Bool $y; ; Int $x){ $x & $y }
multi sub sumB(Bool $y; ; Bool $x){ $x & $y & $y }
multi sub sumC(Int $x; ; Int|Bool $y){ sumI($y,$x) }
multi sub sumC(Bool $x; ; Int|Bool $y){ sumB($y,$x) }
multi sub sumC(Bool $x){ sumC(Bool $x){ (Bool $y){ $x && $y } }

multi sub sumC(Bool $x){ Bool $y){ $(x & $y ) }

multi sub sumC(Bool $x){ Bool $y){ $(x && $y ) }

multi sub sumC(Bool $x){ Bool $y){ sumC($x){ $(y > 0) })
\end{verbatim}

In a nutshell, \texttt{sumC} calls \texttt{sumX($y,$x)} where \texttt{X} is either I or B according to whether the first argument of \texttt{sumC} is of type \texttt{Int} or \texttt{Bool}. When called, each \texttt{sumX} subroutine dispatches to the appropriate code according to the type of its first argument, that is, the second argument of the calling \texttt{sumC}. This technique gives us a definition of \texttt{sumC} with the expected behavior but not the expected type. The first \texttt{multi} definition of \texttt{sumC} has type \texttt{Int -->[Bool[Int]])}, notice in particular that the second argument has a union type), while the second \texttt{multi} definition of \texttt{sumC} has type \texttt{Bool-->[Int|Bool])}. Thus the whole \texttt{sumC} function has type:

\begin{verbatim}
[Int --> [Int|Bool]] --> [Int|Bool])
& [Bool --> [Int|Bool]] --> [Int|Bool]) (13)
\end{verbatim}

and this type is less precise than (i.e., it is a supertype of) the type in (12) (see also the type in Footnote 11 that is equivalent to the (12) type). For instance, the type in (12) states that when a function of this type is applied to two arguments of type \texttt{Int}, then the result will be of type \texttt{Int}, while the type in (13) states, less precisely, that the result will be of type \texttt{Int|Bool}.

The solution to define a version of \texttt{sumC} that has the type in (12) is not so difficult: instead of defining separate auxiliary functions (to which the second argument of \texttt{sumC} must be explicitly passed as an argument) I will use \texttt{nested} \texttt{multi} subroutines. Unfortunately Perl6 does not allow the programmer to write anonymous \texttt{multi} functions. But if we suppose that anonymous \texttt{multi} functions were allowed (e.g., by considering anonymous \texttt{multi} definitions in the same block to define the same anonymous function—a suggestion to the designers of Perl), then the correspondence between the following definition of \texttt{sumC} and the type in (12) should be, I think, pretty clear:

\begin{verbatim}
multi sub sumC(Int $x){
    multi sub (Int $y) { $x + $y }
    multi sub (Bool $y) { sumC($y){$x} }
}

multi sub sumC(Bool $x){
    multi sub (Bool $y) { $(x & $y ) }
    multi sub (Int $y) { sumC($x){$(y>0) })
}
\end{verbatim}

This code states that \texttt{sumC} is a \texttt{multi} subroutine formed by two definitions each of which returns as result another (anonymous) \texttt{multi} subroutine.\footnote{Although the code (14) cannot be executed in Perl6 because of the absence of anonymous \texttt{multi} subroutines, it can be faked by non-anonymous \texttt{multi} subroutines as follows:}

\begin{verbatim}
[Int --> [(Int-->[Int&[Bool])) & (Bool-->[([Int&[Bool])]])]
\end{verbatim}

prove it as an exercise [EX4].

\footnote{Although the code (14) cannot be executed in Perl6 because of the absence of anonymous \texttt{multi} subroutines, it can be faked by non-anonymous \texttt{multi} subroutines as follows:}

In a statically-typed language with subtyping, the type of an expression may decrease during the computation.

This is the reason why in the presence of subtyping it is sensible to distinguish between the \texttt{static} and \texttt{dynamic} type of an expression, the former being the type of the expression at compile time, the latter the type that this expression has after being completely evaluated.

As a simple example take the successor function I defined at the beginning of this presentation and apply it to, say, \texttt{(3+2)}:

\begin{verbatim}
(sub (Int $x){($x+1) })(3+2)
\end{verbatim}

At compile time the function in the expression above has type \texttt{Int-->Int} (i.e., the usual type of the successor function) while the argument has type \texttt{Int} (i.e., the type of the application of the plus operator to two integers); therefore the whole expression has (static) type \texttt{Int}. The first step of execution computes the argument expression to \texttt{5}; the type of the argument has thus decreased from \texttt{Int} to \texttt{Odd}. While the whole application has still type \texttt{Int} (the application of a function of type \texttt{Int-->Int} to an argument of...}
type Odd has type Int). The second step of execution computes the application and returns 6: the type of the whole expression has just decreased from Int to Eq.

Now that we have seen that in Perl 6 types may dynamically evolve, let us see an example of how to use dynamic dispatch. For instance, we can define the sum of two integers modulo 2 as follows.

```perl
multi sub mod2sum(Int $x , Int $y) { 0 }
multi sub mod2sum(Int $x , Odd $y) { ($x+1) % 2 }
```

When mod2sum is applied to a pair of integers, then the most specific definition that matches the (dynamic) types of the arguments is selected. So if the argument is a pair composed by an even and an odd number then, according to their order, either the first or the second branch is selected. If the integers are instead both even or both odd, then only the last definition of mod2sum matches the type of the argument and thus is selected. Since in general it is not possible to determine at compile time whether an integer expression will evaluate to an even or an odd number, then a selection delayed until runtime is the only sensible choice. Let us study the type of mod2sum. Obviously, the function has type (Int, Int) --> Int. However it is easy to deduce much more a precise a type for it. First, notice that the subset construction allows the programmer to define types that contain just one value. For instance the singleton type Two that contains only the value “2” is defined as:

```perl
subset Two of Int where { $_ = 2 }.
```

Let me adopt the convention to use the same syntax to denote both a value and the singleton type containing it. So I will use 2 rather than Two to denote the above singleton. More generally, for every value \( v \) of type \( \text{T} \) in the language I assume the following definition as given

```perl
subset v of \( \text{T} \) where { \$_ = \$v }.
```

With the above conventions, it is easy to deduce for mod2sum the type (Int, Int) --> 0|1, that is, the type of a function that when applied to a pair of integers returns either 0 or 1. But a slightly smarter type system could deduce a much more informative type:

```
((Even, Odd) --> 1) & ((Odd, Even) --> 1) & (Int, Int) --> 0|1)
```

First of all, notice that if in the last arrow of the intersection above we had specified just 0 as return type, then this would have been an error. Indeed mod2sum does not have (Int, Int) --> 0 type: it is not true that when mod2sum is applied to a pair of integers it always returns 0. But the type checker can be even smarter and precisely take into account the code selection policy. The code selection policy states that the code that is selected is the most specific one that is compatible with the type of the argument(s). Since the first two multi definitions of mod2sum in (15) are more specific (i.e., they have smaller input types) than the third and last multi definition, then the code in this last multi definition will be selected only for arguments whose types are not compatible with the first two multi definitions. That is, this last definition will be selected only for pairs of integers that are neither in (Odd, Even) nor in (Even, Odd). So a smarter type checker will deduce for mod2sum the following type:

```
((Even, Odd) --> 1) & ((Odd, Even) --> 1) & ((Int, Int) & not(Even, Odd) & not(Odd, Even)) --> 0)
```

This type is strictly more precise than (i.e., it is a strict subtype of) the type in (16) (as an exercise [EX5], we invite the reader to find a function value that is in the type in (16) and not in type in (17)). Actually the type is so precise that it completely defines the function it types. Indeed it is easy by some elementary set-theoretic reasoning to see that (Int, Int) & not(Even, Odd) & not(Odd, Even) is equivalent to (i.e., it denotes the same set of values as) the type (Even, Even) | (Odd, Odd). Equally easy is to see that for any type \( \text{S} \), \( \text{T}_1, \text{T}_2, \text{S} \), the type \( \text{T}_1 \times \text{T}_2 --> \text{S} \) is equivalent to \( \text{T}_1 --> \text{S} \times \text{T}_2 --> \text{S} \): since every function in the former type has both the arrow types in the intersection, and viceversa (cf. the exercise of Footnote 7). By applying these two equivalences to (17) (in particular to the last factor of the top-level intersection) we obtain that (17) is equivalent to

```
((Even, Odd) --> 1) & ((Odd, Even) --> 1) & (Odd, Odd) --> 0) & (Even, Even) --> 0)
```

from which it is easy to see that this last type is the most precise type we can deduce for mod2sum. Before proceeding, let me stop an instant to answer a possible doubt of the reader. The reader may wonder whether it is permitted to apply a function of type (18) (equivalently, of type (17)) to an argument of type (Int, Int): as a matter of fact, no arrow in (18) has (Int, Int) as domain. Of course it permitted: the domain of a function typed by an intersection of arrows is the union of the domains of all the arrows. In the specific case the union of the domains of the four arrows composing (18) is (Even, Odd) | (Odd, Even) | (Odd, Odd) | (Even, Even), that is, (Int, Int). So a function with type (18) can be applied to any argument whose type is (a subtype of) (Int, Int). The precise rule to type such an application is explained later on.

### 2.4.1 Ambiguous selection

Dynamic dispatch does not come for free. Besides the cost of performing code selection at run-time, dynamic dispatch requires a lot of care at static time as well, which is the whole point here. In particular, dynamic dispatch introduces two new problems, those of ambiguous selection and of covariant specialization. In order to illustrate them I must better describe how dynamic selection is performed at runtime. I said that always the most specific code is selected. This is the code of the multi expression whose parameter types best approximate the types of the arguments. In practice consider a function defined by a multi expressions—call them branches—, the \( i \)-th branch being defined for inputs of type \( \text{T}_i \). Apply this function to an argument that evaluates to a value of type \( \text{T} \). The possible candidates for the selection are all the branches whose input type \( \text{T}_i \) is a supertype of \( \text{T} \). Among them we choose the one defined for the least \( \text{T}_i \), that is, the most specific branch.

Of course, the system must verify—possibly at compile time—that whenever the function is applied to an argument in the function domain, then a most specific branch will always exist. Otherwise we jump into an ambiguous selection problem. For instance, imagine that in order to spare a definition I decided to write the function mod2sum in the following (silly) way:

```perl
multi sub mod2sum(Int $x , Odd $y) { ($x % 2) }
multi sub mod2sum(Int $x , ODD $y) { ($x+1) % 2 }
```

If the first argument is even, then we return the second argument modulo 2; if the second argument is odd, then we return the successor of the first argument modulo 2. While from a mathematical viewpoint this definition is correct, computationally this definition is problematic since it may lead an ambiguous dynamic selection. Let \( e \) be an expression of type Int, then mod2sum(e, 1) is well-typed (a static selection would choose the code of the second branch). But if \( e \) reduces to an even number, then both codes can be applied and there is not a branch more specific than the other:
and the return types of the branches. We have already seen that since ambiguity-freedom is needed to ensure unambiguous semantics of the program, specialization-soundness is necessary to ensure the soundness of the system. Thus even if both are formation rules for Perl 6 programs, specialization-soundness can be explained and justified purely in terms of the type system. In order to do that we need to better explain how to deduce the type of the application of a function whose type is an intersection of arrow types to some argument in its domain. To that end let us consider again the type in (20). This type is not very interesting since both arrows that constrains the input types of the various branches and ensures that the computation will never be stuck on a selection. It is important to understand that this is not a problem related to the type system, but just a problem about giving the semantics of multi-subroutine definitions. The type system has no problem of having functions of type

\[
(\text{Even}, \text{Int}) \rightarrow \{0\} \& (\text{Int}, \text{Odd}) \rightarrow \{0\}
\]

(20) which, intuitively, is the type that corresponds to the multi subroutine in (19); a function of this type can be applied to arguments of type (\text{Int}, \text{Int}) and the result will be given the type \{0\}; the definition of \text{mod2sum} given at the beginning of Section 2.4 has indeed this type. The problem only raises when there are two multi definitions with parameters (\text{Even}, \text{Int}) and (\text{Int}, \text{Odd}) since they make the run-time selection (thus, the semantics) of the multi subroutine undefined. In [10] I explained how the problem of ambiguity corresponds in object-oriented languages to the problem of method selection with multiple inheritance: I invite the interested reader to refer to that work for more information.

### 2.4.2 Overriding

In order to ensure the soundness of the type system (i.e., that all type errors will be statically detected) we must impose a second condition on the definition of multi-subroutines, which constrains the return types of the branches. We have already seen that since types evolve along the computation, so does the selection of the branches. For instance take again the original definition of \text{mod2sum} as I defined it in (15). If we apply the function to a pair of expressions of type \text{Int}, e.g., \text{mod2sum}(3+3, 3+2) — then the compiler statically deduces that the third branch will be selected. But if the pair of expressions reduces to a pair composed of, say, an even

\[
\text{mod2sum}(6, 5)
\]

... and an odd number — e.g., \text{mod2sum}(6, 5) —, then at run-time the code of the first branch will be executed. Now, different branches may have different return types. This is clearly shown by the functions \text{sum} and \text{sumC} which in the first version I gave for them are formed of two branches returning different types (\text{Int} and \text{Bool}) for the former, and \text{Int} \rightarrow \text{Int} and \text{Bool} \rightarrow \text{Bool} for the latter, as respectively stated by the types in (9) and (10)). Since the selected branch may change along the computation, so can the corresponding return type, that is, the type of the application. Type soundness can be ensured only if the type of every expression decreases with the computation. Why is it so? Imagine that a function \( f \) with input type \text{Int} is applied to an expression of type \text{Int}. Before performing the application we evaluate the argument. Therefore the value returned at runtime by this expression must be of type \text{Int} or of a smaller type, say \text{Even}, but not of a distinct type such as \text{Boo1}, since the function \( f \) expects arguments of type \text{Int}, not of type \text{Boo1}. This must hold also when the argument of \( f \) is the result of the application of a multi-subroutine, which implies that if the branch selected for the multi-subroutine changes along the computation, then the newly selected branch must have a return type smaller than or equal to the return type of the previously selected branch. Consider now a multi-function with \( n \) branches (as in Definition 2.2) whose \( i \)-th branch has input type \text{Ty} (for \( 1 \leq i \leq n \)), apply it to an argument, and statically determine that the selected branch is the \( h \)-th one. What are the branches that can potentially be selected at run-time? Since the type of the argument cannot but decrease along the computation, then the candidates for selection are all the branches whose input type is a subtype of \text{Ty}. These are the branches that specialize (I will also say override or refine) the branch \text{Ty} and, as can be evinced by the discussion above, soundness is ensured only if the return types of these branches are smaller than or equal to the return type of the \( h \)-th branch. This yields the following definition.

**Definition 2.3 (Specialization).** Let \text{Ty} and \( \text{S}_h \) respectively be the input type and the return type of the \( h \)-th branch of a multi subroutine composed of \( n \) branches. The multi-subroutine is specialization sound if and only if for all \( i, j \in [1..n] \), \( \text{Ty} \subseteq \text{S}_i \) implies \( \text{S}_j \subseteq \text{S}_i \).

In words, the return type of each branch must be smaller than or equal to the return type of all the branches it specializes.

Definition 2.3 for specialization-soundness is, as Definition 2.2 for ambiguity-freedom, a formation rule for programs in Perl 6. These rules concern more the definition of the language, than its type theory. As the type system has no problem to have a type of the form as in (20) which intersects two arrows whose domains have a common lower bound, so the type system has no problem in considering functions of type:

\[
(\text{Int} \rightarrow \text{Odd}) \& (\text{Even} \rightarrow \text{Int})
\]

(21) even if the return type of the arrow with the smaller domain is not smaller than the other return type. We will return on this type later on. For the time being notice that while ambiguity-freedom is needed to ensure unambiguous semantics of the programs, specialization-soundness is necessary to ensure the soundness of the system. Thus even if both are formation rules for Perl 6 programs, specialization-soundness can be explained and justified purely in terms of the type system. Two solutions alternative to being stuck are to chose one of the two branches either at random or according to a predetermined priority. Both solutions are compatible with the typing discipline we are describing in this work (and with the silly definition (19)) but they are seldom used: the use of some priority corresponds to using class precedence lists in object-oriented languages with multiple inheritance, while I am not aware of any language that uses random selection.

13 Two solutions alternative to being stuck are to chose one of the two branches either at random or according to a predetermined priority. Both solutions are compatible with the typing discipline we are describing in this work (and with the silly definition (19)) but they are seldom used: the use of some priority corresponds to using class precedence lists in object-oriented languages with multiple inheritance, while I am not aware of any language that uses random selection.

14 The condition of uniqueness ensures that all the \( T_h \) are (semantically) distinct.
A function of this type accepts any pair of integers that is not syntactic sugar for
if we use "\" to denote set-theoretic difference (i.e., S\T is
syntactic sugar for S\not(T) then this type is equivalent to
(\Int,\Int)\Odd,\Even. The type in (22) specifies that if
the first projection of the argument of a function of this type is even,
then the first projection of the result will be 0, and if the second
projection of the argument is odd, then the second projection of
the result will be 1. A simple example of a function with this type is
the function that maps (x,y) to (x mod 2, y mod 2).

Let us now examine all the possible cases for an application of a
function of the type in (22). According to the type of the argument
of the function, four different cases are possible:

1. The argument is of type (\Even,\Odd): both arrows must apply;
the first arrows tell us that the first projection of the result will be
0, while the second arrow tell us that the second projection will be
1; so we deduce the result type (0,1), that is the intersection
of the result types;
2. The argument is of type (Even,\Even): then only the first arrow
applies and we deduce the type (0,\Int);
3. The argument is of type (Odd,Odd): then only the second
arrow applies and we deduce the type (\Int,1);
4. The argument is of type (\Int,\Int)\Odd,\Even (i.e., we just
know that it is in the domain of the function): both arrows may
apply; since we do not know which one will be used then
we take the union of the result types, that is, (0,\Int)\Int,1).

We can generalize this typing to the application of functions of a
generic type

$$(S_1\rightarrow\text{T}_1)\&(S_2\rightarrow\text{T}_2)$$

(23)
to an argument of type \text{T}. Only four cases are possible:

1. If the argument is in $S_1\&S_2$ (i.e., if $T\cap S_1\cap S_2$), then the
application has type $T_1\&T_2$.
2. If the argument is in $S_1$ and case 1 does not apply (i.e., $T\cap S_1$
and $T\cap S_2$ is empty), then the application has type $T_1$.
3. If the argument is in $S_2$ and case 1 does not apply (i.e., $T\cap S_2$
and $T\cap S_1$ is empty), then the application has type $T_2$.
4. If the argument is in $S_1\&S_2$ and no previous case applies, then
the application has type $T_1\&T_2$.

Of course things become more complicated with functions
typed by an intersection of three or more arrows: the complete for-
malization of the typing of their applications is outside the scope
of this primer and it will be given in Section 4. But an intersection
of two arrows is all we need to explain the specialization sound-
ness rule. So let us consider the type in (23) and suppose that, say,
$S_2\subseteq S_1$. What happens when we apply a function of this type to
an argument of type $S_2$? Since $S_2\subseteq S_1$ then $S_2\subseteq S_1\&S_2$, therefore it
is the first of the four possible cases that applies: the argument is in
$S_1\&S_2$ and thus the result will be of type $T_1\&T_2$ (... which is smaller
than $T_2$).

Let us rephrase what we just discovered: if we have a function of type
$$(S_1\rightarrow\text{T}_1)\&(S_2\rightarrow\text{T}_2)$$
with $S_2\subseteq S_1$, and we apply it to an argument of the smaller type, that is $S_2$, then the result will be in $T_1\&T_2$. The fact that $T_2$ is not smaller than $T_1$ does not matter
because the typing rules tell us that the application of this function to
an argument of type $S_2$ will return a result in a type smaller than $T_1$.
Let us consider a concrete example for $T_1$, $T_2$ by going back to the
type in (21). A function is of type $(\Int\rightarrow\Odd)\&(\Even\rightarrow\Int)$
only if for arguments of type Even it returns results in Odd\&\Int,
that is in Odd: although the arrow with domain Even does not specify
a result type smaller than the one for the arrow of domain Int,
the typing rules ensure that all the results will be included in the
latter type.

In the view of what I just wrote, I can next show that Definition
2.3 is, as for the condition of ambiguity, just a formation rule
and not a problem related to the type system. It is a design choice
that ensures that the type of a well-typed multi definition is what a
programmer expects it to be. Consider a definition of the form

multi sub foo($S_1, $x) returns $T_1 \{ ... $x mod 2, $y mod 2; ...

In Perl 6 the returns keyword declares the result type of a sub-
routine. In the example above, it declares that the first definition has type $S_1\rightarrow T_1$ and the second one type $S_2\rightarrow T_2$. If each
definition is well typed, then the programmer expects the multi
definition above to have type $(S_1\rightarrow T_1)\&(S_2\rightarrow T_2)$. Suppose that
$S_2\subseteq S_1$. In this case the typing rules state that a function has type
$(S_1\rightarrow T_1)\&(S_2\rightarrow T_2)$ only if whenever it is applied to an
argument of type $S_2$ it returns a result in $T_1\&T_2$. By the semantics of
multi definitions, if foo is applied to an argument of type $S_2$, then
it executes the second definition of foo and, therefore, it returns a
result in $T_2$. Putting the two observations together we conclude that
foo has type $(S_1\rightarrow T_1)\&(S_2\rightarrow T_2)$ if and only if $T_2\subseteq T_1$ and
by a simple set-theoretic reasoning—if and only if $T_2\subseteq T_1$: ex-
actly the condition enforced by Definition 2.3.

To say it otherwise: the type system does not have any problem
in managing a type $(S_1\rightarrow T_1)\&(S_2\rightarrow T_2)$ even if $S_1\subseteq S_2$ and $T_1$
and $T_2$ are not related. However, such a type might confuse the
programmer since a function of this type applied to an argument
of type $S_1$ returns a result in $T_1\&T_2$. In order to avoid this confu-
sion the language designer can (without loss of generality) force the
programmer to specify that the return type for a $S_1$ input is (some
subtype of) $T_1\&T_2$. This can be obtained by accepting only special-
ization sound definitions and greatly simplifies the presentation of
the type discipline of the language.

This concludes the presentation of the primer on type theory.
In this section I have introduced substantial type-theoretic concepts
and explained them in terms of the set-theoretic interpretation of
types. We have seen how the set-theoretic interpretation of types al-
ows us to deduce non-trivial subtyping relations on complex types.
I have also tried to distinguish concepts that concern the seman-
tics of the language, such as the two conditions on the definition
of multi-subroutines, from the type-theoretic concepts that explain
and justify them. It is now time to put all we have learnt so far in
practice and apply it to understand and solve a controversy that has
been the core of a heated debate in the object-oriented languages
community for several year in the late eighties, early nineties: the
so-called “covariance vs. contravariance problem”.

2.5 Lessons to retain

For the busy programmer who did not have time to read this long
section, all I explained here can be summarized by the following
six simple rules:

1. Types are sets of values. In particular:
   a. the type \text{T} is the set of all functions that when applied
      to values in \text{S} return only results in \text{T};

\text{Odd}\&\Even}

\text{Even}\&\Int)
(b) union, intersection, and negation types are the sets of values obtained by applying the corresponding set operations.

2. \( S \) is a subtype of \( T \) if and only if all values in \( S \) belong also to \( T \).

3. Two types are equal if they are the same set of values.

4. A multi-subroutine is typed by the intersection of the types of each multi definition; but, for that to hold, these definitions must satisfy the property of covariant specialization (Definition 2.3).

5. When a multi-subroutine is applied to an argument, the most precise multi definition for that argument is used; but, for that to happen, the definitions that compose the multi subroutine must be free from ambiguity (Definition 2.2).

6. The definition of a subroutine in which only some arguments are used for code selection (the ‘; ; ’ notation) corresponds to defining a function whose parameters are those preceeding the ‘ ; ’ and that returns another function whose parameters are those following the ‘ ; ’.

3. The covariance and contravariance (false) problem

If I have not disgusted my average programmer reader, yet, she should have acquired enough acquaintance with types and dynamic dispatching to be ready to tackle what in the study of object-oriented languages is called the covariance vs. contravariance problem. Since this controversy took place in the object-oriented community, I therefore start explaining it by using Perl 6 objects and then reframe it in the context introduced in the previous section. I assume the reader to be familiar with the basic concepts of object-oriented programming.

3.1 Objects in Perl 6

The classic example used in the late eighties to explain the problem at issue was that of programming a window system in which the basic objects were pixels, represented by the class `Point` written in Perl 6 as:

```perl
class Point {
  has $.x is rw;
  has $.y is rw;
  method origin() {
    ($.x==0) && ($.y==0)
  }
  method move(Int $dx, Int $dy) {
    $.x += $dx;
    $.y += $dy;
    return self;
  }
};
```

Objects (or instances) of the class `Point` have two instance variables (fields, in Java parlance) \( x \) and \( y \), and two methods associated to the messages `origin` and `move`, respectively. The former takes no argument and returns whether the receiver point has null coordinates or not, the latter takes two integers, modifies the instance variables of the receiver of the message `move` (i.e., the `invocant` of the method `move`, in Perl parlance), and returns as result the receiver itself, which in Perl 6 is denoted by the keyword `self`. New instances are created by the class method `new`. Methods are invoked by sending the corresponding message to objects, by dot notation. For example,

```perl
my Point $a = Point.new(x => 23, y => 42);
$a.move(19,-19);
```

creates an instance of the class `Point` whose instance variables have values 23 and 42 and then moves it by inverting the values of the instance variables. Notice that in the definition of \( \$a \) (first line of our code snippet) the name `Point` has a double usage: while the second occurrence of `Point` denotes a `class` (to create a new instance) the first occurrence is a `type` (to declare the type of \( \$a \)).

In our set-theoretic interpretation the type `Point` denotes the set of all the objects that are instances of the `class` `Point`.

If later we want to extend our window system with colors (in late eighties black and white screens were the norm), then we define by `inheritance` a subclass `ColPoint` that adds a \( c \) field of type string (in Perl, \( \text{Str} \)) for the color and a method `iswhite`, it `inherits` the fields \( x \) and \( y \) and the method `origin` from `Point`, and, finally, it `specializes` (or overrides) the method `move` of `Point` so that white colored points are not modified.

```perl
class ColPoint is Point {
  has Str $.c is rw;
  method iswhite() {
    return ($.c=="white");
  }
  method move(Int $dx, Int $dy) {
    if not(self.iswhite()) {
      $.x += $dx;
      $.y += $dy;
    }
    return self;
  }
};
```

In many object-oriented languages, Perl 6 included, inheritance is associated with subtyping: declaring a subclass by the keyword `is` as, in the example above, has the double effect of making the new class definition inherit the code of the super-class definition and of declaring the type of the objects of the subclass to be a subtype of the type of the objects of the super-class. In our example it declares `ColPoint` to be a subtype of `Point`. Therefore every object of type/class `ColPoint` is also of type `Point` and, as such, it can be used wherever a `Point` object is expected. For instance, the following code

```perl
my Point $a = ColPoint.new(x=>2,y=>21,c=>'white');
$a.move(3,3);
```

is legal and shows that it is legitimate to assign a `ColPoint` object to a variable \( \$a \) declared of type `Point`. Notice that, even though \( \$a \) has static type `Point`, it is dynamically bound to an object of type/class `ColPoint` and, therefore, the code executed for \( \$a\).move will be the one defined in the class `ColPoint`. In particular, during the execution of this method, the message `iswhite` will be dynamically sent to (the object denoted by) \( \$a \) even though \( \$a \) has static type `Point`, for which no `iswhite` method is defined (the words “dynamic dispatch” and “dynamic binding” come from there). On the contrary, an invocation such as \( \$a\).iswhite will (or, rather, should, since current implementations of Perl 6 do not check this point) be statically rejected since \( \$a \), being of type `Point`, cannot in general answer to the message `iswhite`, and it is in general out of reach of a type system to determine at static time whether \( \$a \) will be bound to a `ColPoint` or not (although in this particular case it is pretty obvious).

3.2 Inheritance and subtyping

Since the objects of a subclass can be used where objects of the super-class are expected, then definitions by inheritance must obey some formation rules to ensure type soundness. This is sensible for overridden methods: while it is harmless to add new instance
variables or methods in a subclass, specialization of methods requires the use of a subtype. The reasons for this is that, as an object of a subclass can be used where an object of the super-class is expected, so the overriding method of the subclass can be used where the overridden method of the super-class is expected. This is clearly shown by the code snippet:

```perl
@a.move(3,3)
```

I wrote a few lines above: since $a is of (static) type Point, then the system assumes that the message move will execute the method defined in the class Point, which expects a pair of integers and returns a Point (i.e., it is a function of type (Int, Int)--&gt;Point), even though in reality it is the method in the definition of ColPoint that is used instead. Using the latter definition where the former is expected is type safe since the type of the method in ColPoint is a subtype of the type of the method in Point: rule (2) states that (Int, Int)--&gt;ColPoint &lt;: (Int, Int)--&gt;Point, and indeed $a.move(3,3) will return a color point, that is, a value of a subtype of the expected Point type.

**Contravariant overriding:** In general, to ensure type safety, when in a definition by inheritance we specialize (i.e., override) a method the new method must have a subtype of the type of the original method. By the co-contra-variant rule in (2) this means that the return type of the new method must be a subtype of the return type of the old method (covariant specialization) while its arguments must have a supertype of the type of the arguments of the old method (contravariant specialization). Since the latter is taken as characteristics of this kind of specialization, the whole rule has taken the name of contravariant overriding.

**Covariant overriding:** So far so good. Troubles start when one considers binary methods, that is methods with arguments of the same type as the type of the receiver [7]. The paradigmatic example is the equal method which, for the class Point, can be defined as follows.

```perl
class Point {
  has $.x is rw;
  has $.y is rw;
  method equal(Point $p) { ($ .x==$p.x) && ($ .y==$p.y) }
}
```

When later we introduce the ColPoint class it is natural to want to redefine the method equal so as (i) it takes arguments of type ColPoint and (ii) it compares also the colors, that is:

```perl
class ColPoint is Point {
  has Str $.c is rw;
  method equal(ColPoint $p) {
    ($ .x==$p.x) && ($ .y==$p.y) && ($ .c==$p.c)
  }
}
```

This is called covariant specialization, since in the subclass a method overrides the previous definition of the method by using parameters whose types are subtypes of the types of the corresponding parameters in the overridden method. Unfortunately, the definition above (and covariant specialization in general) is unsound as shown by the following snippet which is statically well typed but yields a type error at run-time:

```perl
my Point $a = Point.new(x=&gt;23, y=&gt;42);
$a.move();
```

Perl 6 complains by “Not enough positional parameters passed; got 1 but expected 3” since in the second line of the above code it detected the point argument $a of move, but the two integer arguments are missing. Now, if we consider a single message, such as move, then it is associated to different function definitions (the methods in Point and ColPoint for move) and the actual code to execute when a message such as move is sent is chosen according to the type of the receiver of the message, that is, according to the type of the methods’ hidden argument. In Section 2 we already saw that functions with multiple definitions are called multi subroutines, and that the arguments used to choose the code are those listed before the “;” (if it occurs) in the parameter list.

If we remove method definitions from classes and replace them by multis where the receiver parameter has become an explicit pa-

---

[16] For Perl 6 purists. Since Perl 6 performs dynamic type checking, then the run-time error actually happens at the call of the equal method rather than at its execution. Here I described the behaviour shown when types are checked only at compile time, since a sound static type checking makes dynamic type checking useless. Also notice that, for the sake of simplicity, I used “==” (rather than “eq” as required in Perl) for the string equality operator.
If we apply the same transformation to the equal method we obtain

```perl
multi sub equal(Point $self ; ; Point $p) {
    ($self.x==$p.x) && ($self.y==$p.y);
}
```

```perl
multi sub equal(ColPoint $self ; ; ColPoint $p) {
    ($self.x==$p.x) && ($self.y==$p.y) && ($self.c==$p.c);
}
```

The definitions above define a function equal that should have the following type:

```perl
equal: (Point-->(Point-->Bool))
     & (ColPoint-->(ColPoint-->Bool))
```

However, the two multi definitions yield a function that does not have this type inasmuch as they do not comply with the specialization formation rule: since ColPoint<:Point, then by Definition 2.3 we need (ColPoint-->Bool):(Point-->Bool) which by contravariance does not hold. Thus the definition is unsound. An equivalent way to see this problem is that a function equal is of the type in (24) only if when applied to a ColPoint object it returns a value in the following intersection type:

```perl
(Point--->Bool) & (ColPoint--->Bool)
```

(cf. the first of the cases listed right after equation (23) and the discussion thereby); but the multi definitions above—in particular the second multi definition of equal—do not (the second definition does not have type Point-->Bool and, a fortiori, it does not have the intersection above).

So far we did not learn anything new since we already knew that the definition above was not sound. If however in the definition of the multi subroutine equal we replace a "", " for "", " that is

```perl
multi sub equal(Point $self , Point $p) {
    ($self.x==$p.x) && ($self.y==$p.y);
}
```

```perl
multi sub equal(ColPoint $self , ColPoint $p) {
    ($self.x==$p.x) && ($self.y==$p.y) && ($self.c==$p.c);
}
```

then the definition becomes well typed, with type

```perl
equal: ((Point,Point)--->Bool)
      & ((ColPoint,ColPoint)--->Bool)
```

In particular, the specialization formation rule of Definition 2.3 is now satisfied: (ColPoint,ColPoint)<:(Point,Point), and this requires Bool<:Bool, which clearly holds. So the definition of equal is now sound: where is the trick? The consequence of replacing "", " for "", " is that the new version of equal uses the dynamic types of both arguments to choose the code to execute. So the second definition of equal is selected only if both arguments are (dynamically) instances of ColPoint. If any argument of equal is of type Point, then the first definition is executed. For instance, the method invocation $a.equal($b) I gave in Section 3.2 to show the unsoundness of covariant specialization now becomes equal($a,$b); and since the dynamic type of $b is also used to select the code to execute, then when this type is Point the first definition of the multi subroutine equal is selected.

With hindsight the solution is rather obvious: this transformation tells us that it is not possible to choose the code for binary methods, such as equal, by using the type of just one parameter (i.e., the type of the receiver); the only sound solution is to choose the code to execute based on the types of both arguments.

Finally, observe that we are not obliged to check the type of both arguments at the same time: we can first check one argument and then, if necessary (e.g., for equal if the first argument is a ColPoint), check the other. This tells us how to solve the problem in the original setting, that is where methods are defined in classes: it suffices to use multi subroutines also for methods. In Perl 6 this
is obtained by adding the modifier multi in front of a method definition. This yields a solution in which the definition of the class Point does not change:

```java
class Point {
    has $.x is rw;
    has $.y is rw;

    method equal(Point $p) {
        ($.x==$p.x) && ($.y==$p.y)
    }
}
```

while the class ColPoint now defines a “multi method” for equal:

```java
class ColPoint is Point {
    has Str $.c is rw;

    multi method equal(Point $p) {
        ($.x==$p.x) && ($.y==$p.y)
    }

    multi method equal(ColPoint $p) {
        ($.x==$p.x) && ($.y==$p.y) && ($.c==$p.c)
    }
}
```

The body of the `equal` method with a ColPoint parameter is as before. We just added an extra method in the class ColPoint to handle the case in which the argument of equal is a Point, that is, it is an argument that was statically supposed to be compared with another Point: in this we do not check the c instance variable. In words, if `equal` is sent to a Point, then the method defined in Point is executed; if `equal` is sent to a ColPoint, then the type of the argument is used to select the appropriate multi method.

While the type of `equal` defined in Point is `Point --> Bool` (as it was before), the type of `equal` in ColPoint is now the intersection type `Point --> Bool & ColPoint --> Bool`. The latter is a subtype of the former (obviously, since for all types S, T, we have S & T < S). So at the end we did not discovered nothing really new, since it all sums up to using the old classic rule for subclassing:

**Overriding rule:** the type of an overriding method must be a subtype of the type it overrides

The only novelty is that now intersection types give us the possibility to have some form of covariant specialization: in a class D subclass of a class C we can safely override a method of type C --> S by a new method of type (C --> S) & (D --> T) where the code associated to D --> T represents the covariant specialization of the old method.

Two concluding remarks on the Point-ColPoint example. First, notice that the type of `equal` in ColPoint is exactly the type we found in (25), that is, the type suggested by the intersection type theory for the overriding method. Hence, if we abstract from the “syntactic sugar” of the objects and we consider methods as multi-subroutines with implicit parameters, then the last two classes define a multi-subroutine `equal` that has the type (24) we were looking for. Second, it is worth noticing that the solution based on multi methods is modular. The addition of the class ColPoint does not require any modification of the class Point, and a class such as Point can be defined independently from whether it will be later subclassed with a covariant method specialization or not.\(^{18}\)

\(^{18}\)Interestingly, the same behaviour as the solution above can be obtained simply by adding the modifier multi to the method `equal` in Point to the definitions given in Section ref since in that case the (multi) method of Point is not overridden in ColPoint.

**Excursus.** In the preceding example the method that must be added to handle covariant specialization is explicitly defined by the programmer. However, it is possible to imagine a different solution in which this “missing” method is instead added by the compiler. All the method added by the compiler has to do is to call the overridden method (i.e., to dispatch the message to “super”). The choice of whether the method added to handle covariant specialization is to be written by the programmer or inserted by the compiler, belongs to design. The reader can refer to [6] and [5] to see how the second choice can be implemented in Java and O₂, respectively.

To summarize, we have just seen how the type theory we studied in Section 2 allows us to propose a solution to the covariant specialization of methods. Although the rule for subclassing is still the usual one — you can override methods only by methods of smaller type —, the presence of intersection types tells us that it is ok to covariantly specialize a method as long as we do it by a multi-method that dynamically handles the case in which the argument has a supertype of the expected type (i.e., an argument that was intended for the overridden method). Although the detailed explanation of how we arrived to this solution is to some degree convoluted, the solution for the final programmer amounts to retaining the Overriding Rule I stated above and to be able to apply it when complex types are involved, in particular when functions are typed by intersection types. This is the reason why it is important for a programmer to grasp the basic intuition of the subtyping relation and, in that respect, the set-theoretic interpretation of types as sets of values should turn out to be, I believe, of invaluable help.

### 3.4 Lessons to retain

As for the previous section, I summarize for the busy programmer the content of this section in two rules:

1. The type of an overriding method must be a subtype of the type of the method it overrides, whether these types are arrows or intersections of arrows.
2. As a consequence of the previous point, it is ok to covariantly specialize a method as long as we do it by a multi-method in which at least one definition can handle arguments intended for the overridden method.

### 4. Type algorithms for the language designer (i.e., the electrical blueprints)

In this section I am going to describe the algorithms and datastructures needed to implement the type system of the previous section. I will give a bare description of the algorithms and datastructures and justify them just informally. Detailed justifications and proof of formal properties such as correctness can be found in the references commented in Section 5.

```java
class Point {
    has $.x is rw;
    has $.y is rw;

    multi method equal(Point $p) {
        ($.x==$p.x) && ($.y==$p.y)
    }
}

class ColPoint is Point {
    has Str $.c is rw;

    multi method equal(ColPoint $p) {
        ($.x==$p.x) && ($.y==$p.y) && ($.c==$p.c)
    }
}
```

however this solution is less modular than the one above since it may require to modify the definition of Point when the class ColPoint is added at a later time.
I will proceed first by defining the algorithm that decides whether two types are in subtyping relation as defined in Definition 2.1. Next I will describe the data structures used to efficiently implement types and their operations. Then, I will describe the typing of the expressions, focusing on those that are more difficult to type, namely, subroutines, projections, applications, and classes. I conclude the section with the algorithms for records whose presentation is quite technical and can be skipped at first reading.

4.1 Type syntax

But first I must define the types I will be working with. These are the types defined in Section 2.1 with two slight improvements: more precise base types and recursive types.

More precisely I replace \( \text{Bool} \) and \( \text{Int} \) respectively by “tags” (ranged over by \( t \)) and “intervals”, two base types that cover a wide range of possible implementations. This yields the following grammar

\[
T ::= [i..j] | \text{Any} | (T,T) | T --> T | T \& T | \neg (T)
\]

An interval is denoted by \([i..j]\) where \( i \) and \( j \) are numeric constants or \( * \) (which denotes infinite). For instance, \([2..4]\) is the type that denotes the set \( \{2, 3, 4\} \), \([1..*]\) is the type of positive integers, while \( \text{Int} \) now becomes a shorthand for \(*..*\). A tag is a sequence of letters starting by a back-quote, such as ‘li’ or ‘title’, that denotes a user-defined value and are akin to “barewords” (actualy, barewords are deprecated in Perl). When used as a type, a tag denotes the singleton containing that tag. In particular, I can encode \( \text{Bool} \) as the union of two tag types, ‘true’ and ‘false’. The remaining types have the same interpretation as before. Tags, intervals, products and arrows are called type connectors, while unions, intersections, and negations are called type connectives. I will use \( \text{Empty} \) to denote the type \( \neg (\text{Any}) \), and \( S \setminus T \) to denote the set theoretic difference of two sets \( S \) and \( T \).

To cope with recursive types, I will consider the set of possibly infinite syntax trees generated by the grammar above (in technical jargon, it is the language “coinductively” produced by the grammar) that satisfy the following two conditions: (a) they are regular trees (i.e., each tree has finitely many distinct subtrees) and (b) on every infinite branch of a tree there are infinitely many occurrences of product or arrow type constructors (these two conditions are called “contractivity conditions”). The first restriction ensures that recursive types have a finite representation (e.g., finite sets of equations or recursive definitions). The second restriction ensures that recursion always traverses at least one type constructor, thus barring out ill-formed types such as \( T \rightarrow T \). The algorithm uses set-theoretic equivalences and decomposes the problem into simpler subproblems, namely: \( (T_1, T_2) \) and \( (S_1, S_2) \) is empty if and only if \( (T_1 \& S_1, T_2 \& S_2) \) is empty, if and only if either \( T_1 \& S_1 \) is empty or \( T_2 \& S_2 \) is empty. With that in mind the subtyping algorithm can be summarized in 4 simple steps.

Step 1: transform the subtyping problem into an emptiness decision problem. Deciding whether \( S \subseteq T \) is equivalent to deciding whether the difference of the two types is empty. So the first step of the algorithm is to transform the problem \( S \subseteq T \) into \( \text{Empty} \).

Step 2: put the type whose emptiness is to be decided in a disjunctive normal form. Our types are a propositional logic whose atoms are defined by grammar (27). A literal (ranged over by \( \ell \)) is either an atom or the negation of an atom: \( \ell ::= a | \neg (a) \). Every type is equivalent to a type in disjunctive normal form, that is, a union of intersections of literals:

\[
\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}
\]

with the convention that \( \text{Any} \) and \( \text{Empty} \) are, respectively, the empty intersection and the empty union. Therefore, the second step of the algorithm consists in transforming the type \( S \& \neg (T) \) whose emptiness is to be checked, into a disjunctive normal form.

Step 3: simplify mixed intersections. The algorithm has to decide whether a disjunctive normal form, that is, a union of intersections, is empty. A union is empty if and only if every member of the union is empty. Therefore the problem reduces to deciding emptiness of an intersection of literals: \( \bigwedge_{i \in I} \ell_i \). Notice that there are four kinds of atoms and thus of literals, one for each type constructor. Intersections that contain literals of different kinds can be straightforwardly simplified: if in the intersection there are two atoms of different constructors, say, \((T_1, T_2)\) and \(S_1 \rightarrow S_2\), then their intersection is empty and so is the whole intersection; if one of the two atoms is negated and the other is not, say, \((T_1, T_2)\) and \(\neg (S_1 \rightarrow S_2)\), then the negated atom is useless and can be removed since it contains the one that is not negated; if both atoms are negated, then the intersection can be split in two intersections of atoms of the same kind by intersecting the atoms...
with the respective top types (e.g., \( \text{not}(T_1, T_2) \) \& \( \text{not}(S_1 \rightarrow S_2) \) can be split into the union of \( \text{not}(T_1, T_2) \) \& \( \text{not}(S_1 \rightarrow S_2) \) \& \( \text{Empty} \rightarrow \text{any} \)).

Therefore the third step of the algorithm performs these simplifications so that the problem is reduced to deciding emptiness of interstions that are formed by literals of the same kind, that is one of the following cases (where \( P \) stands for “positives” and \( N \) for “negatives”):

\[
\bigwedge_{p \in P} t_p \& \bigwedge_{n \in N} \text{not}(t_n) \tag{29}
\]

\[
\bigwedge_{p \in P} [t_p \ldots t_p] \& \bigwedge_{n \in N} \text{not}([t_n \ldots t_n]) \tag{30}
\]

\[
\bigwedge_{(S_1, S_2) \in P} (S_1, S_2) \& \bigwedge_{(T_1, T_2) \in N} \text{not}((T_1, T_2)) \tag{31}
\]

\[
\bigwedge_{S_1 \rightarrow S_2} (S_1 \rightarrow S_2) \& \bigwedge_{T_1 \rightarrow T_2} \text{not}(T_1 \rightarrow T_2) \tag{32}
\]

**Step 4: solve uniform intersections and recurse.** At this point we have to decide whether every intersection generated at the previous step is empty. When the intersection is formed by atoms of base type \(-\), i.e., cases (29) and (30), then emptiness can be immediately decided: an intersection as in equation (29) is empty if and only if the same tag appears both in a positive and a negative position (i.e., there exists \( p \in P \) and \( n \in N \) such that \( t_p = t_n \)). If two distinct tags appear in positive position (i.e., there exist \( p_1, p_2 \in P \) such that \( t_{p_1} \neq t_{p_2} \)), while whether an intersection as in equation (30) is empty can be decided by simple computations on the interval bounds.

If instead the intersection is composed of products or arrows, then first we check whether we already proved that the intersection type at issue is empty (we have recursive types so we memoize intermediate results). If we did not, then we memoize the type and decompose it using its set-theoretic interpretation. Precisely, to decide emptiness of the type in (31), we decompose the problem into checking that for all \( N' \subseteq N \)

\[
\bigwedge_{(S_1, S_2) \in P} (S_1 <: \bigvee_{(T_1, T_2) \in N'} T_1) \text{ or } \bigwedge_{(S_1, S_2) \in P} (S_2 <: \bigvee_{(T_1, T_2) \in N'} T_2) \tag{33}
\]

holds, which is done by recursively proceeding to Step 1.

To decide emptiness of the type in (32), we decompose the problem into checking whether there exists \( T_1 \rightarrow \cdots \rightarrow T_2 \subseteq N \) such that \( T_1 \subseteq S_1 \rightarrow S_2 \rightarrow P \) and for all \( P' \subseteq P \) (notice that containment is strict)

\[
(T_1 <: \bigvee_{S_1 \rightarrow S_2 \in P'} S_1) \text{ or } (\bigwedge_{S_1 \rightarrow S_2 \in P'} S_2 <: T_2) \tag{34}
\]

holds, which is checked by recursively proceeding to Step 1.

**[End of the Algorithm]**

In order to complete the presentation of the algorithm above, let me show how to define the recursive functions that compute the Boolean functions defined in (33) and (34) in Step 4. I focus on the algorithm for (34) since it is the most difficult one and leave the case for products as an exercise. Given a \( T_1 \rightarrow \cdots \rightarrow T_2 \subseteq N \) I first need to check whether \( T_1 \subseteq \bigvee_{S_1 \rightarrow S_2 \in P} S_1 \) and, if so, then compute a function, say \( \Phi \) that given \( T_1, T_2 \) and \( P \) checks whether for every strict subset \( P' \) of \( P \), the formula in (34) holds. If \( P \) is empty, then \( \Phi \) must return true. Otherwise pick an element \( e \in P \) (the choice of \( e \) can be arbitrary) and let \( Q = P \setminus \{ e \} \); then \( \Phi \) does three things: (i) it solves the problem for \( P' = \{ e \} \); (ii) it recursively solves the problem for all subsets of \( Q \) (i.e., the subsets of \( P \) that do not contain \( e \)) and (iii) it recursively solves the problem for all non-empty subsets of \( Q \) to which \( e \) is added (i.e., the non-singleton subsets of \( P \) that contain \( e \)). This yields the following recursive definition of the subtyping relation for the arrow case \( S_1 \rightarrow S_2 \in P \) \( S_1 <: T_1 \rightarrow T_2 \):

\[
\begin{align*}
(T_1 <: \bigvee_{S_1 \rightarrow S_2 \in P} S_1) \text{ and } \Phi(T_1, T_2, P, \text{empty, } \text{any}) \tag{35}
\end{align*}
\]

where

\[
\begin{align*}
\Phi(T_1, T_2, P, D, C) &= \text{true} \\
\Phi(T_1, T_2, P \cup \{ S_1 \rightarrow S_2 \}) &= ((T_1 <: S_1') \text{ or } \Phi(S_1 \rightarrow S_2, P, D, C)) \\
\Phi(T_1, T_2, S_1 \rightarrow S_2) &= \text{false}
\end{align*}
\]

Notice that \( \Phi \) uses two extra parameters \( D \) and \( C \) used as accumulators respectively for the domains and codomains met in previous recursive calls. As a matter of fact, we do not need these two extra parameters since we can store them in the first two parameters, by using suitable set-theoretic operations. If we define the function \( \Phi' \) as follows:

\[
\begin{align*}
\Phi'(T_1, T_2, P, D, C) &= \Phi'(T_1, T_2, P \cup \{ S_1 \rightarrow S_2 \}) \\
\Phi'(T_1, T_2, S_1 \rightarrow S_2) &= \text{false}
\end{align*}
\]

then it is not difficult to see that for all \( T_1, T_2, P, D, C \),

\[
\Phi(T_1, T_2, P, D, C) = \Phi'(T_1 \setminus D, C \setminus T_2, P)
\]

and, therefore, we can use

\[
\begin{align*}
(T_1 <: \bigvee_{S_1 \rightarrow S_2 \in P} S_1) \text{ and } \Phi'(T_1, \text{not}(T_2), P) \tag{36}
\end{align*}
\]

instead of (35).

I invite the reader to verify that both \( \Phi \) and \( \Phi' \) compute the Boolean function described in (34) and leave the definition of a similar function for (33) as exercise [EX6]. In particular, I suggest to try to use the properties \( \bigwedge_{(S_1, S_2) \in P} (S_1, S_2) = \bigcup_{(S_1, S_2) \in P} S_1 \), \( \bigcup_{(S_1, S_2) \in P} S_2 \) and \( \bigcap_{(T_1, T_2) \in N} (S_1, S_2) \setminus \bigcup_{(T_1, T_2) \in N} (S_1, S_2) \) to define an algorithm potentially more efficient.

### 4.3 Data structures and their operations

The subtyping algorithm (as well as the typing algorithm I describe farther on) works with types in disjunctive normal form, which are transformed by applying unions, intersections, and differences. Since I want to avoid to normalize types at each step of the algorithm, then the algorithm will work with types stored in disjunctive normal form. Thus I need to find an efficient representation for disjunctive normal forms and define the operations of union, intersection and difference so that they preserve this representation. Recall that a type in disjunctive normal form can be represented as:

\[
\bigvee_{i \in \{ \bigwedge_{p \in R_i} \text{not}(a_p) \}} \bigwedge_{i \in \{ \bigwedge_{p \in R_i} \text{not}(a_p) \}} \tag{37}
\]

where \( a_i \)'s are atoms. A naive representation of (37) (such as lists of pairs of lists) does not fit our needs since it is not compact and makes it difficult to efficiently implement set theoretic operations.

**Binary decision diagrams.** A classic technique to store compactly Boolean functions is to use Binary Decision Diagrams (BDD). In the context I am studying a BDD is a labeled binary tree whose nodes are labeled by atoms and whose leaves are labeled by 0 or 1. In a BDD every path starting from the root and terminating by \( 1 \) represents the intersection of the atoms that label
the nodes of the path, each atom occurring negated or not according to whether the path went through the left or right child of the atom’s node. For instance, the BDD representing the disjunctive normal form \((a1 \land a2) \lor (a1 \land \neg a2 \land a3) \lor \neg(a1 \land \neg a3)\) is given in Figure 1 (the first and third summands of the union correspond respectively to the leftmost and rightmost paths of the BDD). Formally, BDDs are defined by the following grammar:

\[
B ::= \mathbf{0} \mid \mathbf{1} \mid a?B:B
\]

and have the following interpretation:

\[
\begin{align*}
\mathbf{0} & \quad \text{Empty} \\
\mathbf{1} & \quad \text{Any} \\
[a?B1:B2] & \quad (a \land [B1]) \lor (\neg(a) \land [B2])
\end{align*}
\]

The interpretation above maps a BDD into the disjunctive normal form it represents (of course, after having simplified the intersections with \(\text{Any}\), Empty), that is, into the union of the intersections that correspond to paths ending by a \(a\). To ensure that the atoms occurring on a path are distinct, we define a total order on the atoms and impose that on every path the order of the labels strictly increases. It is possible to implement all set-theoretic operations directly on BDD. Let \(B, B1,\) and \(B2\) denote generic BDDs, \(B = a1?C1:D1\) and \(B2 = a2?C2:D2\). Unions, intersections, and differences of BDDs are defined as follows:

\[
\begin{align*}
\text{\(1 \lor B\)} & = \text{\(1 \lor B\)} \\
\text{\(1 \land B\)} & = \text{\(B \land 1\)} \\
\text{\(0 \lor B\)} & = \text{\(B \lor 0\)} \\
\text{\(0 \land B\)} & = \text{\(B \land 0\)} \\
\text{\(B \land \mathbf{1}\)} & = \text{\(B \land 0\)} \\
\text{\(B \lor \mathbf{1}\)} & = \text{\(B \lor 0\)}
\end{align*}
\]

Notice that \(\mathbf{1} \lor B\) computes the negation of \(B\) and is obtained by exchanging the \(0\) leaves of \(B\) into \(\mathbf{1}\) and viceversa.

After having performed any of these operations, we can simplify a BDD by replacing every subtree of the form \(a?B:B\) by just \(B\).

**BDD with lazy unions.** A well-known problem of BDDs is that by repeatedly applying unions we can have an exponential blow-up of their size. To obviate this problem the CDuce compiler uses a lazy implementation for unions. These are evaluated just when they are needed, that is, when computing differences and intersections. To obtain it we represent BDDs as ternary trees, of the form \(a?B1:B0:B2\), where the middle child represents a lazy union:

\[
[a?B1:B0:B2] = (a \land [B1]) \lor (\neg(a) \land [B2])
\]

Let \(B1 = a1?C1:U1:D1\) and \(B2 = a2?C2:U2:D2\). When two atoms are merged, unions are lazily recorded in the middle child:

\[
B1 \lor B2 = \begin{cases}
    a1?C1 \lor C2:U1 \lor U2:V1 \lor D2 & \text{for } a1 = a2 \\
    a1?C1:U1 \lor U2:V1 & \text{for } a1 < a2 \\
    a2?C2:V1 \lor U2:V1 & \text{for } a1 > a2
\end{cases}
\]

The intersection \(B1 \land B2\) materializes the lazy unions when the top-level atoms are the same and trees are merged, and is defined as:

\[
B1 \land B2 = \begin{cases}
    a1?C1 \land C2:U1 \land U2 & \text{for } a1 = a2 \\
    a1?C1:U1 \land U2 & \text{for } a1 < a2 \\
    a2?C2:V1 \land U2 & \text{for } a1 > a2
\end{cases}
\]

while the difference \(B1 \setminus B2\) materializes unions when top-level atoms are distinct, and is defined as:

\[
B1 \setminus B2 = \begin{cases}
    a1?C1 \setminus C2:U1 \setminus U2 & \text{for } a1 = a2 \\
    a1?C1:U1 \setminus U2 & \text{for } a1 < a2 \\
    a2?C2:U1 \setminus U2 & \text{for } a1 > a2
\end{cases}
\]

After having performed these operations it is possible to perform two simplifications, namely, replace \((a?B1:B2)\) by \(1\), and replace \((a?B:B)\) by \(B \lor a\).

**Disjoint atoms.** The representation above does not exploit the property used by Step 3 of the subtyping algorithm, namely, that it is possible to consider disjoint normal forms in which the intersections do not mix atoms of different kinds. This means that the union given in (37) can be seen as the union of four different unions, one for each kind of atom:

\[
\begin{align*}
\mathbf{1} \lor (\mathbf{a} \land \&_{i \in I_{\text{tag}}} p_{i} \land \&_{n \in N_{\text{tag}}} \text{not}(n_{\text{tag}})) & = (38) \\
\mathbf{1} \lor (\mathbf{a} \land \&_{i \in I_{\text{ext}}} p_{i} \land \&_{n \in N_{\text{ext}}} \text{not}(n_{\text{ext}})) & = (39) \\
\mathbf{1} \lor (\mathbf{a} \land \&_{i \in I_{\text{prod}}} p_{i} \land \&_{n \in N_{\text{prod}}} \text{not}(n_{\text{prod}})) & = (40) \\
\mathbf{1} \lor (\mathbf{a} \land \&_{i \in I_{\text{arr}}} p_{i} \land \&_{n \in N_{\text{arr}}} \text{not}(n_{\text{arr}})) & = (41)
\end{align*}
\]

Instead of representing a disjunctive normal form by a unique BDD—which mixes atoms of different kinds—, it is more compact to store it in four distinct unions. A type will be represented by a record with a field for each distinct kind of atom, each field containing a disjunctive normal form of the corresponding atom, namely:

\[
\begin{align*}
\text{tags: } & \text{dfn\_tags} ; \quad // \text{stores a union as in (38)} \\
\text{ints: } & \text{dfn\_ints} ; \quad // \text{stores a union as in (39)} \\
\text{prod: } & \text{dfn\_prod} ; \quad // \text{stores a union as in (40)} \\
\text{arrw: } & \text{dfn\_arrw} ; \quad // \text{stores a union as in (41)}
\end{align*}
\]

Let Kinds denote the set \{tags, ints, prod, arrw\}. An expression \(T\) of the type above represents the following disjunctive normal form:

\[
\bigvee_{k \in \text{Kinds}} (T, k \land \&_{y_{k}} y_{k})
\]

Different fields for different kinds of atom yield a more compact representation of the types. But the real gain of such an organization is that, since atoms of different kinds do not mix, then all set-theoretic operations can be implemented component-wise. In other terms, \(S \land T\) and \(S \lor T\) can be respectively implemented as:

\[
\begin{align*}
\text{tags: } & S\text{.tags} \land T\text{.tags} ; \quad \{ \text{tags: } S\text{.tags} \land T\text{.tags} ; \quad // \text{stores a union as in (38)} \\
\text{ints: } & S\text{.ints} \land T\text{.ints} ; \quad \{ \text{ints: } S\text{.ints} \land T\text{.ints} ; \quad // \text{stores a union as in (39)} \\
\text{prod: } & S\text{.prod} \land T\text{.prod} ; \quad \{ \text{prod: } S\text{.prod} \land T\text{.prod} ; \quad // \text{stores a union as in (40)} \\
\text{arrw: } & S\text{.arrw} \land T\text{.arrw} ; \quad \{ \text{arrw: } S\text{.arrw} \land T\text{.arrw} \}
\end{align*}
\]
and similarly for the union \( S \lor T \). Therefore, not only we have smaller data structures but also the operations are “partitioned” on these smaller data structures, and thus executed much more efficiently.

To conclude the presentation of the implementation of types I still have to show how to represent the disjunctive normal forms contained in each field and how to implement recursive types.

For the fields \( \text{prod} \) and \( \text{arr} \), corresponding to product types and arrow types the use of BDDs with lazy unions to represent contained in each field and how to implement recursive types.

Checking the emptiness of the base type fields \( \text{tags} \) and \( \text{ints} \), we can use instead a specific representation. In particular, it is not difficult to prove that any union of the form (38) can be equivalently expressed either as a union of pairwise distinct atoms \((a_1 \lor \ldots \lor a_n)\) or its negation \((\neg(a_1 \lor \ldots \lor a_n))\) (I leave this proof as an exercise for the reader [EX7]). The same representation can be used also for disjunctive normal forms of intervals, which can thus be expressed as a union of disjoint intervals or its negation (if the intervals are maximal — i.e., adjacent intervals are merged— then this representation is unique). In conclusion, a disjunctive normal form of base types can be expressed as a set of atoms and a positive/negative flag indicating whether this set denotes the union of the atoms or its complement (of course, the implementation of the set-theoretic operations for these fields must be specialized for this specific representation).

Finally, to represent recursive types it suffices to allow the records representing types to be recursively defined. By construction the recursion can occur only in the types forming an atom of a BDD in the \( \text{prod} \) or \( \text{arr} \) fields. This means that the contravariance conditions of Section 4.1 are satisfied by construction.

**Emptiness (i.e., subtyping).** I end the presentation of data structures by showing how the subtyping algorithm described in Section 4.2 specializes to these data structures. As expected with these data structures the algorithm is much simpler (the representation handles all steps of normalization) and consists just of two steps. Let \( S \) and \( T \) be two types represented by a record of four fields as described above. In order to verify whether \( S \vdash T \) holds do:

**Step 1:** Compute \( S \lor T \);

**Step 2:** Check that all the fields of \( S \setminus T \) are the empty type.

Checking the emptiness of the base type fields \( \text{tags} \) and \( \text{ints} \) is immediate: they must be an empty set of atoms with a positive flag. For the fields \( \text{prod} \) and \( \text{arr} \), if the BDD that they contain is not 0 or 1, then we apply the respective decomposition rule described in **Step 4** in Section 4.2, memoize, and recurse.

As a final exercise for this part, the reader can try to define a function \( \text{norm} \) that takes as argument a type produced by the grammar given at the beginning of Section 4.1 and returns the record representing its disjunctive normal form [EX8].

### 4.4 Typing algorithms

I conclude the presentation of my electrical blueprint by specifying the algorithms for typing some expressions that are commonly found in programming languages.

#### 4.4.1 Products

Perl 6 includes list expressions and element selection. For the sake of simplicity I will just consider products, since lists can then be encoded as nested products.

In what follows I consider expressions of the form \( (e_1, e_2) \) which returns the pair formed by the values returned by \( e_1 \) and \( e_2 \), as well as the expressions \( e[0] \) and \( e[1] \) denoting respectively the first and second projection of \( e \).

The algorithm for typing pairs is straightforward: if \( e_1 \) is of type \( T_1 \) and \( e_2 \) of type \( T_2 \), then \((e_1, e_2)\) is of type \( (T_1, T_2) \).

The algorithm for typing projections, instead, is more complicated, since projections can be applied to any expression of type \( \text{Any}_{\text{prod}} \), that is, any expression whose type has a normal form as in (40). So imagine that we want to type the expressions \( e[0] \) or \( e[1] \) where \( e \) is of some type \( T \). The first thing to do is to verify that \( e \) will return a pair, that is, that \( T \vdash \text{Any}_{\text{prod}} \) holds. If it is so, then \( T \) is equivalent to the following normal form:

\[
\left( \bigvee_{i \in I} \left( S_p \lor T_p \right) \land \bigwedge_{n \in \mathbb{N}} \neg (S_n \lor T_n) \right)
\]

in which case the expression \( e[0] \) has type

\[
\left( \bigvee_{i \in I, \neg P \subseteq \mathbb{N}} \left( S_p \lor T_p \right) \land \bigwedge_{n \in \mathbb{N}} \neg (S_n \lor T_n) \right)
\]

and, likewise, the expression \( e[1] \) has type

\[
\left( \bigvee_{i \in I, \neg P \subseteq \mathbb{N}} \left( T_p \lor \neg (S_n \lor T_n) \right) \right)
\]

with the convention that an empty intersection of atoms denotes the top type of the corresponding kind.

Let me explain how to pass from (42) to (43) and (44). The idea is simple and consists to transform the union in (42) into a union of products (i.e., no intersection of products and no negated product) by using two simple observations. First, an intersection of products is equivalent to the product of the intersections:

\[
\bigwedge_{p \in P} (S_p \lor T_p) \rightleftharpoons \bigwedge_{p \in P} S_p \land \bigwedge_{p \in P} T_p
\]

Second, the intersection of a product with a negated product can be distributed inside the product: \((S_1 \lor S_2) \land \neg (T_1 \lor T_2)\) is equivalent to \((S_1 \land \neg T_1) \lor (S_2 \land \neg T_2)\). For multiple intersections such as

\[
\bigwedge_{p \in P} (S_p \lor T_p) \land \bigwedge_{n \in \mathbb{N}} \neg (S_n \lor T_n)
\]

the two transformations above yield the following equivalent type

\[
\bigwedge_{n \in \mathbb{N}} \bigvee_{p \in P} (S_p \lor T_p) \land \bigwedge_{n \in \mathbb{N}} \neg (S_n \lor T_n)
\]

it is then easy from this type to deduce the types (43) and (44) of the projections, simply by observing that the projection of a union of production is the union of the projections of each product.

#### 4.4.2 Subroutines

The technique to type subroutines should be pretty clear by now. Given a definition either of the form

\[
\text{sub } (T_1 \; \text{\$x}_1, \ldots, \; T_n \; \text{\$x}_n) \{ \; e \; \}
\]

or of the form

\[
\text{sub } (T_1 \; \text{\$x}_1, \ldots, \; T_n \; \text{\$x}_n) \; \text{returns } \; S \{ \; e \; \}
\]

it has type \((T_1, \ldots, T_n) \rightarrow S\) if under the hypothesis that \(x_1\) has type \(T_1\) and that \(x_n\) has type \(T_n\) it is possible to deduce that \(e\) has type \(S\) (the difference between the two cases is that in the former the type \(S\) is the one returned by the algorithm for \(e\), while in the latter the algorithm checks whether the type returned for \(e\) is a subtype of the type \(S\) specified by the programmer). Likewise

\[
\text{sub } (T_1 \; \text{\$x}_1, \ldots, \; T_n \lor \text{\$x}_n) \; \text{returns } \; S \{ \; e \; \}
\]

has type \((T_1, \ldots, T_n) \rightarrow (T_{n+1}, \ldots, T_{n+k}) \rightarrow S\) if under the hypotheses that \(x_i\) has type \(T_i\) (for \(i = 1, n + k\)) it is possible to deduce that \(e\) has type \(S\).

Finally, given a multi-subroutine composed of \(n\) definitions, if the \(i\)-th definition has type \(S_i \rightarrow T_i\) and all these definitions form a set that is both specialization sound and free of ambiguity (cf. Definitions 2.3 and 2.2), then the multi-subroutine has type \(\bigvee_{i=1}^n S_i \rightarrow T_i\).

Notice that I defined type-checking only for subroutines whose parameters are explicitly typed. The definition of a type system that infers also the types of subroutine parameters (technically, this is
called a “type reconstruction” system and that infers polymorphic types as done in the languages of the ML family is possible. However, its technical development is complex: it would need twice the space taken by this paper and is clearly outside the scope of our presentation. I invite the interested reader to consult the references given in Section 5 on the subject.

4.4.3 Applications

Now that we can type subroutines, it is time to type their applications. The typing of an application is the same, independently from whether the applied subroutine is multi or not. In both cases the function value is typed by an intersection of arrows, formed by just one arrow when the subroutine is not multi. Suppose we have two expressions \(e_1\) and \(e_2\) which are well typed respectively with type \(T\) and \(S\). We want to check \((a)\) whether the application \(e_1 e_2\) is well typed and, if so, \((b)\) deduce the best possible type for this application.

In order to verify \((a)\) the algorithm proceeds in two steps: first it checks that \(e_1\) is a function (i.e., it returns neither a constant nor a pair). This is done by checking that the type \(T\) of \(e_1\) is a subtype of \(\text{Any}_{\text{arrow}}\), that is \(T \subseteq \text{Empty} \rightarrow \text{Any}\). If this holds then it checks that the type \(S\) of the argument is a subtype of the domain \(\text{dom}(T)\) of the function, that is \(S \subseteq \text{dom}(T)\), where the domain is defined as follows:

\[
\text{dom}(V \{\bigcup_{i \in I} S_p \rightarrow T_p \& \bigwedge_{n \in N_i} \not= (S_n \rightarrow T_n)\}) = \bigcup_{i \in I} \bigwedge_{p \in P_i} S_p
\]

The definition of domain is given for a normal form as in (41) since this is the disjunctive normal form of any type smaller than \(\text{Any}_{\text{arrow}}\). To compute the domain only the positive arrows are used. The domain of an intersection of arrows is the union of their domains, since a function of that type accepts arguments that are compatible with at least one arrow in the intersection: whence the inner union (e.g., a function of type \(\text{Bool} \rightarrow \text{S}\&\text{Int} \rightarrow \text{T}\)) can be applied both to Boolean and to integer arguments. If a function is typed by a union of function types, then it accepts only arguments that are compatible with all arrows in the union: whence the outer intersection (e.g., an expression of type \(\{(1 \ldots 4) \rightarrow \text{S}\} \{(2 \ldots 6) \rightarrow \text{T}\}\)) can be applied only to arguments in \(\{2 \ldots 4\}\); since we cannot statically know which of the two types the function returned by the expression will have, then we must accept only arguments that are compatible with both arrows).

Once the algorithm has verified that \(S\) and \(T\) satisfy the two conditions above, then the application is well typed. It is then time to compute the type of this application.

Since \(T \subseteq \text{Any}_{\text{arrow}}\—i.e., \(T\) is a function type—then

\[
T = \bigcup_{i \in I} \bigwedge_{p \in P_i} S_p \rightarrow T_p \& \bigwedge_{n \in N_i} \not= (S_n \rightarrow T_n)\)  \hspace{1cm} (45)
\]

for some \(I, P_i\) and \(N_i\). Then the type of the application of a function of the type \(T\) defined in (45) to an argument of type \(S\) is

\[
\bigwedge_{i \in I} \left( Q \subseteq P, Q \& S \& P \rightarrow Q \& T_p \right) \left( \bigwedge_{p \in P, Q} T_p \right)\)  \hspace{1cm} (46)
\]

Let me decrypt this formula for you, but feel free to skip directly to Section 4.4.4 if you are not interested in these details. First, consider the case of (45)—and, thus, (46)—where \(I\) is a singleton. Since negated arrow types (i.e., those in \(N_i\)) do not play any role in the typing expressed by formula (46), then we can for simplicity consider that the type \(T\) of the function is

\[
\bigwedge_{p \in P} S_p \rightarrow T_p
\]

and that the type \(S\) of the argument is a subtype of the domain of \(T\), that is \(S \subseteq \bigwedge_{p \in P} S_p\).

If the argument can return a value in the domain \(S_j\) (for some \(i \in P\)) of some arrow (i.e., if \(S \& S_j\) is not empty), and this actually happens, then the result returned by the application (if any) will be in \(T_j\). If the argument can return a value in domain of two arrows say \(S_i\) and \(S_j\) for \(i, j \in P\) (i.e., \(S \& S_i \& S_j\) is not empty) then the result of the application may be a value in \(T_i \& T_j\). Of course we want to deduce the most precise type for the result type. So in this case we will consider for our result type \(T_i \& T_j\) rather than just \(T_i\) or \(T_j\). Also, it may be the case that \(T_i \& T_j\) does not cover all possible cases for the result: not only \(S\) may intersect domains of the smaller than \(S_i \& S_j\) (e.g., the intersection of three arrow domains), but also this does not cover the case in which the argument returns a result that falls outside \(S_i \& S_j\), that is, a result in \(S \setminus (S_i \cup S_j)\), whenever this set is not empty. The return type for this case must be computed separately and then added to the final result type.

So the algorithm proceeds as follows. First \((a)\) it computes all the possible intersections of the types \(S_i\) for \(i \in P\) and keeps only those whose intersection with \(S\) is not which (\(b)\) for every intersection kept that is also minimal it computes the intersection of the corresponding codomains; finally \((c)\) it takes the union of the computed intersections of codomains.

This is exactly what the formula in (46) does. First it considers all possible combinations of the domains \(S_p\) for \(p \in P\), that is, all non-empty subsets of \(P\). Then, for each of these subsets it keeps the largest subsets such that the intersections of \(S\) and the corresponding domains is not empty. To do that it takes every strict subset \(Q\) of \(P\) (strict, since we want \(P \setminus Q\) to be non-empty) such that \(S \not\subseteq \bigcup_{Q \subseteq Q} S_p\). Take now any maximal subset \(Q\) that satisfies \(S \not\subseteq \bigcup_{Q \subseteq Q} S_p\). Being maximal means that if we add to \(Q\) any index \(p\) in \(P \setminus Q\) then by adding the corresponding \(S_p\) we cover the whole \(S\). This means that all elements of \(S\) that are missing in \(\bigcup_{Q \subseteq Q} S_p\)—that is all elements in \(S \setminus \bigcup_{Q \subseteq Q} S_p\)—are in all the \(S_p\) for \(p \in P \setminus Q\). Therefore the intersection \(S \& \bigcup_{P \rightarrow Q} S_p\) is not empty, since it contains \(S \setminus \bigcup_{Q \subseteq Q} S_p\). The maximality of \(Q\) implies the minimality of the intersection of the domains in \(P \setminus Q\). So we take the intersection of the corresponding codomains, that is \(\bigwedge_{P \rightarrow Q} T_p\), and we union all these intersections (notice that the union in (46) also adds some smaller intersections which correspond to the subsets \(Q\) contained in the maximal sets: this does not matter since these intersection are already contained in the previous ones, and thus do not change the result).

To complete the decryption of our formula (46), it remains the case in which \(I\) is not a singleton: when we have a union of function types, then we can apply a function of this type only to arguments that are in the intersection of the domains of the types that form the union and, therefore, we are sure that whatever the actual type of the function will be, the result will be in every result type computed for each element of the union, and therefore in their intersection.

4.4.4 Classes

I already discussed how to type-check classes in Section 3: one can use very sophisticated type systems for classes (there is a plethora literature on the subject) but the bare minimum is that \((i)\) methods have the type they declare, \((ii)\) that overriding methods are typed by a subtype of the type of the methods they override, and \((iii)\) that the type of read and write instance variables must be the same in all subclasses.

In my discussion I considered just the very basic features of class-based object-oriented programming, since a more detailed treatment would have led me far away from the purpose this work. There is however a feature that I omitted in the discussion and that seems worth mentioning, namely the case of multiple inheritance. In many languages (Perl 6 included) it is possible to define a class
as a subclass of several distinct classes: for instance I could have defined the ColPoint class in this way:

```perl
class Point {
    has $.x is rw;
    has $.y is rw;

    method equal(Point $p) {
        ($x==$p.x)&($y==$p.y)
    }
}

class Color {
    has Str $.c is rw;

    method iswhite() {
        return ($.c=="white")
    }
}

class ColPoint is Point is Color {
    method move(Int $dx, Int $dy) {
        if not(self.iswhite()) {
            $.x += $dx;
            $.y += $dy;
        }
        return self;
    }
}
```

A problem with multiple inheritance is how to deal with methods that are defined in more than one superclass. For instance, which method should be selected for $a.equal($a) if $a is an instance of the class ColPoint and the class Color had defined a method equal : Color --> Bool? The interpretation given in Section 3.3 to classes and the definition of ambiguity give us an immediate answer to this, since therein equal would be the following multi-subroutine

```perl
multi sub equal(Point $self ;; Point $p) {
    ($self.x==$p.x)&($self.y==$p.y) }

multi sub equal(Color $self ;; Color $p) {
    $self.c==$p.c }
```

Adding the class ColPoint corresponds to a common subtype to Point and Color. As a consequence, the type Point&Color is no longer empty (it contains all the ColPoint objects) and therefore the multi-subroutine above no longer is free from ambiguity. The solution in this case is straightforward: to satisfy the ambiguity condition the class ColPoint must provide its own definition of the equal method (in Perl 6 this is what it is done with "roles"). This is however problematic when multi-subroutines are not the result of class definitions but are defined directly: defining a class by multiple inheritance from other classes means creating a new subtype common to all these classes; therefore a multiple subroutine whose parameters are typed by some of those classes and that was ambiguity-free might, because of the new class definition, break the ambiguity condition. This means that the ambiguity condition must thus be rechecked for all such multi subroutines, thus hindering the compositionality and modularity of the type system.

4.5 Records

Although records (also known as "structs" in C(++), "structures" in Visual Basic, objects in JavaScript, ...) are pervasive in programming, I did not explain how to handle them. It is time to remedy this omission by this section (which is by far the most technical section of the presentation and can be skipped at first reading). In their simplest form records are finite maps from "labels" to values which are equipped with a selection operation that returns the value associated to a label by the record. In Perl 6 labels are strings and records (i.e., "hashes" in Perl parlance) are defined by the syntax `{ ℓ₁ => e₁,...,ℓₙ => eₙ }` which denotes the record that associates the label `ℓᵢ` to (the value returned by) the expression `eᵢ`, for `i ∈ [1...n]`. For instance, if we write

```perl
my $x = {
    "foo" => 3,
    "bar" => "foo"
};
```

then the variable `$x` denotes the record that associates the label `string foo` to the integer 3 and the string bar to the string foo. When the same label is multiply defined then the rightmost association is taken. Selection in Perl 6 is denoted by `e<ℓ>` which returns the value associated to the label `ℓ` in the record returned by the expression `e`. In the example above `$x<foo>` (note the absence of the double quotes around the label) returns the integer 3 and `$x<bar>` returns the string foo. The selection of a label that is not present in the record returns `undef`, a special value that when used in any context causes a runtime error. Perl provides syntax to initialize undefined fields and to associate new values to defined ones. To model (and type) these operations, as well as the definition of records with multiple fields, I build records starting by single-field records that are merged by the record concatenation operator "+" that, in case of multiply defined labels, gives priority to the rightmost fields. For completeness I will also consider field deletion, that is, I will use the record expression `e<ℓ>` whose result is the record value returned by `e` in which the field labeled by `ℓ`, if any, is undefined. The record expressions that I consider henceforth are then defined by the following grammar:

```
  e ::= { ℓ => e } | e + e | e<ℓ> | e<ℓ> (47)
```

For instance, the record associated to the variable `$x` in the previous expression is written in this syntax `{ foo => 3 } + { bar => "foo" }`. However, for the sake of simplicity in examples I will still use the syntax `{ foo => 3, bar => "foo" }`.

In what follows I show how to type this simple form of records. However, in many programming languages records have much a richer set of operations and features. I will discuss some of them at the end of this section.

In Perl 6 records are implemented by the class `Hash` which provides a very rich set of features for them. This means that Perl 6 lacks specific types for records. So I will depart from what I did so far and instead of borrowing the syntax of Perl types, I will use my own syntax for record types. More precisely, to type fields I will use either the syntax `$ℓ : T` or (for optional fields) the syntax `$ℓ ? : T` for the latter that the same operation will return either a value of type `T` or `undef`. This syntax will be in the record types which come in two flavors: closed record types whose values are records with exactly the fields specified by the type, and open record types whose values are records with at least the fields specified by the type. I use `{ f₁,...,fₙ }` for the former and `{ f₁,...,fₙ , .. }` for the latter (where the `fᵢ`’s denote either of the two forms of field typing I described before) and use `R` to range over either kinds of record type expressions. For instance, the record `{ foo => 3, bar => "foo" }` of our example can be given several distinct types. Among them we have: `{ foo : Int , bar : Str }`, the closed record type that includes all the records with exactly two fields, one that maps the label foo to integer values and the other that maps the label bar to string values; `{ foo : Int , .. }` the open record type formed by all the records that contain at least a field `foo`
containing integers; \{foo : Int , pol ? : Bool , ..\} the open record type with two optional fields; \{foo : Int , bar : Str , pol ? : Bool\} the closed record type in which one of the two fields that are present is declared optional. Finally the record of our example has also type \{\}, that is, the type that contains only all record values \(\{\}\) is the top type for record kind, that is, Any_{rec}.

The goal of this section is to illustrate the algorithms that assign types to expressions of records, as illustrated in the example above. Deducing that \{foo⇒3 , bar⇒"foo"\} has type \{foo : Int , bar : Str\} is straightforward. Instead, to deduce that it has type \{foo : Int , ..\} or to infer the types of record selection and concatenation operations is more difficult. The former deduction is obtained by subsumption (likewise for the deduction of \{\} type), the two latter deductions need the definition of some operations on record types. Thus I need first to explain the semantics of record types, then their subtyping relation, and finally some operations on them. At the beginning of this section I introduced records, as customary, as finite maps from an infinite set of labels \(L\) to values. In what follows I use a slightly different interpretation and consider records as specific total maps on \(L\), namely, maps that are constant but on a finite subset of the domain. So in this interpretation the record \{foo⇒3 , bar⇒"foo"\} maps the label foo into the value 3, the label bar into the value "foo", and for the other labels it is the constant function that maps them into the special value \(\perp\) (i.e., the undefined value of Perl).

Formally, let \(Z\) denote some set, a function \(r : L → Z\) is quasi-constant if there exists \(z \in Z\) such that the set \(\{l ∈ L \mid r(l) ≠ z\}\) is finite; in this case I denote this set by dom(r) and the element \(z\) by def(r). I use \(L → Z\) to denote the set of quasi-constant functions from \(L\) to \(Z\) and the notation \(\{1 = z_1 , ..., n = z_n\}\) to denote the quasi-constant function \(r : L → Z\) defined by \(r(ℓ) = z_i\) for \(ℓ = 1..n\) and \(r(ℓ) = z\) for \(ℓ \in L \setminus \{1, ..., n\}\). Although this notation is not univocal (unless we require \(z_i \neq z\)), this is largely sufficient for the purposes of this section.

Let \(⊥\) be a distinguished constant. I single out two particular sets of quasi-constant functions: the set string → Types∪\{\\}, called the set of quasi-constant string functions ranged over by \(r\); and string → Values∪\{\\} the set of record values. The constant \(\perp\) represents the value of the fields of a record that are “undefined”.

To ease the presentation I use the same notation both for a constant and the singleton type that contains it: so when \(\perp\) occurs in \(L → Values∪\{\\}\) it denotes a value, while in string → Types∪\{\\} it denotes the singleton type that contains only the value \(\perp\).

Given the definitions above, it is clear that the record type expressions I defined earlier in this section are nothing but specific notations for some quasi-constant functions in string → Types∪\{\\}. More precisely, the open record type \(\{ℓ_1 : T_1 , ..., ℓ_n : T_n\}\) denotes the quasi-constant function \(\{ℓ_1 = T_1 , ..., ℓ_n = T_n\}\) while the closed record type \(\{ℓ_1 : T_1 , ..., ℓ_n : T_n\}\) denotes the quasi-constant function \(\{1 = T_1 , ..., n = T_n\}\). Similarly, the optional field notation \(...?...\) denotes the record typing in which \(ℓ\) is mapped either to \(\perp\) or to the type \(T\), that is, \(\{..., ℓ = T | 1\} \perp\ldots\). In conclusion, the open and closed record types are just syntactic to denote specific quasi-constant functions in string → Types∪\{\\}.

Subtyping. The first thing to do is to define subtyping for record types, that is, to extend the algorithm I described in Section 4.2 to handle record type expressions. The algorithm proceeds as in Section 4.2 with the difference that the simplification performed in Step 3 may yield besides the cases (29)–(32) the following case:

\[
\bigwedge_{p \in P} R_p \land \bigwedge_{n \in N} \neg\text{not}(R_n) \quad (48)
\]

where, I remind, every type \(R_i\) in this formula is a record type atom, that is, it is either of the form \(\{ℓ_1 : T_1 , ..., ℓ_n : T_n\}\) or of the form \(\{1 = T_1 , ..., n = T_n\}\). Then Step 4 simplifies the instances of this new case by using the containment property for quasi-constant functions (see Lemma 9.1 in [21]) specialized for the case in (48) (remember that the \(R_i\)'s in the formula above denote special cases of quasi-constant functions.) In particular to decide the emptiness of the type in (48) we decompose the problem into checking that for every map \(i : N → L\) the condition holds:

\[
\begin{align*}
&\exists ℓ \in L. (\bigwedge_{p \in P} R_p(ℓ) \land \bigwedge_{n \in N} \neg\text{not}(R_n)) \lor \\
&\exists n \in N. ((1(n) = _\perp) \land (\bigwedge_{p \in P} \text{def}(R_p) \land \text{def}(R_n))) \\
\end{align*}
\]

(49)

where \(L = \bigcup_{i \in P ∪ N} \text{dom}(R_i)\). The technique used to derive (36) can be adapted to this case as well (just by branching on \(L \cup \{\perp\}\) rather than a single element).

Type operators. Let \(T\) be a type and \(R_1 , R_2\) two record type atoms. The merge of \(R_1\) and \(R_2\) with respect to \(T\), noted \(T ⊠\) and used infix, is the quasi-constant typing function defined as follows:

\[
(R_1 ⊠ T R_2)(ℓ) \overset{def}{=} \begin{cases} R_1(ℓ) & \text{if } R_1(ℓ) \not\sqsubseteq \text{Empty} \\
R_2(ℓ) & \text{otherwise} \end{cases}
\]

Record types are akin to product types (a classical interpretation of records is that of products indexed over a finite domain). As products commute with intersections (cf. the discussion at the end of Section 4.4.1) so do record types and, therefore, every type containing only record values can be expressed as a union of just record type atoms. In other terms if \(T \sqsubseteq T_1\), then \(T\) is equivalent to a finite union of the form \(\bigvee_{i \in I} R_i\) (see Lemma 11 in [4] for details). So the definition of merge can be easily extended to all record types as follows:

\[
\bigvee_{i \in I} R_i ⊠ T \bigvee_{j \in J} R_j' \overset{def}{=} \bigvee_{i \in I ∪ J} (R_i ⊠ T R_j')
\]

The merge operator can be used to define concatenation and field deletion on record types. Let \(T_1 , T_2\), and \(T\) range over record types (i.e., types smaller than \(\perp\)). Define:

\[
T_1 + T_2 \overset{def}{=} T_2 ⊠ T_1
\]

\[
T \setminus ℓ \overset{def}{=} \{ℓ = \perp\} \setminus C_{c_0} ⊠ T
\]

(50)

(51)

where \(c_0\) is some constant/tag different from \(\perp\) (the semantics of the operator does not depend on the choice of \(c_0\) as long as it is different from \(\perp\)).

Notice in particular that the result of the concatenation of two record type atoms \(R_1 + R_2\) may result for each field \(ℓ\) in three different outcomes:

1. if \(R_2(ℓ)\) does not contain \(\perp\) (i.e., the field \(ℓ\) is surely defined), then we take the corresponding field of \(R_2\): \((R_1 + R_2)(ℓ) = R_2(ℓ)\)
2. if \(R_2(ℓ)\) is undefined (i.e., \(R_2(ℓ) = \perp\)), then we take the corresponding field of \(R_1\): \((R_1 + R_2)(ℓ) = R_1(ℓ)\)
3. if \(R_2(ℓ)\) may be undefined (i.e., \(R_2(ℓ) = T \sqsubseteq \perp\) for some type \(T\), then we take the union of the two corresponding fields since it can results either in \(R_1(ℓ)\) or \(R_2(ℓ)\) according to whether the record typed by \(R_2\) is undefined in \(ℓ\) or not: \((R_1 + R_2)(ℓ) = R_1(ℓ) \sqcup (R_2(ℓ) \setminus \perp)\)

This explains some weird behavior of record concatenation such as \(\{a : Int , b : Int\} + \{a? : Bool\} = \{a = Int[Bool , b : Int]\}\) since “\(a\)” may be undefined in the right hand-side record while “\(b\)”
is undefined in it, and \(\{a : \text{Int}\} + \{a\} = \{a\}\), since “\(a\)” in the right hand-side record is defined (with \(a \mapsto \text{Any}\)) and therefore has priority over the corresponding definition in the left hand-side record.

**Typing.** All it remains to do is to give the typing rules for the expressions of grammar (47). Thanks to the definitions (50) and (51) this is really straightforward:

- **record:** if \(e\) has type \(T\), then \(\{e \mapsto e\}\) has type \(\ell : T\);
- **concatenation:** if \(e_1\) has type \(T_1\), \(e_2\) has type \(T_2\), and both \(T_1\) and \(T_2\) are subtypes of \(\{\ldots\}\), then \(e_1 + e_2\) has type \(T_1 + T_2\);
- **deletion:** if \(e\) has type \(T\) and \(T\) is a subtype of \(\{\ldots\}\), then \(e^{/}\) has type \(T^{/}\);
- **selection:** if \(e\) has type \(T\) and \(T\) is a subtype of \(\{\ldots\}\), then \(T\) is equivalent to a union type of the form \(\bigvee_{i \in I} B_i\) and the type of \(e^{\ell}\) is \(\bigvee_{i \in I} B_i(\ell)\).

To conclude this section on records let me hint at the richer forms of records you may encounter in other programming languages. A first improvement you can find in several programming languages and in Perl is that the labels of a record expression may be computed by other expressions rather than by being just constants. For instance, it is possible to define:

```perl
my $y = {
    $x:<bar> => 3,
};
```

which associates to \(\$y\) the record that maps the string \(<foo>\) to the integer \(3\) (where \(\$x\) is defined as at the beginning of the section). While this extension greatly improves the versatility of records, it also greatly complicates their static typing since in general it is not possible to statically determine the label that will be returned by a given expression. It is possible to give a rough approximation by using union types when an expression is known to produce only a finite set of strings (for an example of this technique the reader can refer to Section 4.1 of [4]) but the precision of such a technique is seldom completely satisfactory.

A further improvement is to allow the label in a selection expression to be computed by an expression, too. In Perl 6 this is done by the expression \(e_1(e_2)\) which returns the value associated to the string returned by \(e_2\) in the record returned by \(e_1\). Back in the example of the beginning of this section we have that the four expressions \(\$x:<foo>, \$x:<"foo">, \$x{$x:<bar>}, \$x{$x:"bar"}\) all return the integer \(3\) (as a matter of fact, in Perl 6 \(<\rangle\) is syntactic sugar for \(e^{\langle\rangle}\)). Such an extension makes records equivalent, to all intents and purposes, to arrays indexed over a finite set of strings (rather than an initial segment of integers). These in other languages are called associative maps/arrays/lists. Once more the extension improves versatility while complicating static typing and one can try to adapt the techniques suggested for the previous extension also to this case. Finally, some languages (e.g., MongoDB) consider the fields of a record to be ordered and allow the fields of a record to be selected by their order number, as if they were an array. The techniques presented in this section can be hardly adapted to such an extension.

### 4.6 Summary for the electrical blueprint

The algorithms I presented in this part are essentially those defined by Alain Frisch in his PhD. thesis [21] and implemented in the compiler of CDuce. These algorithms are directly derived from the set-theoretic interpretation of types. This fact, not only allows us to precisely define the semantics of the types and of the subtyping relation, but also makes it possible to optimize the algorithms by applying classic set-theoretic properties. Furthermore, by using well-known and robust data structures to represent Boolean functions such as the BDDs, it was possible to modularize the implementation according to the kinds of type constructors. This makes it possible to avoid expensive normalization phases and makes it much easier to extend the system to include new type constructors.

Although I gave a pretty complete presentation of the algorithms and data-structures that efficiently implement a type system based on semantic subtyping, this is not the whole story. The actual implementation has to use hash-tables for efficient memoization, hash-consing for efficient manipulation of types, determine the best policy for node sharing in BDDs, tailor representations of basic data according to the application domain of the language (for instance in CDuce strings use a representation tailored for a lazy implementation of basic operators), and so on. However, even a naïve implementation of the algorithms of this section should display pretty decent performances.

While I described in details how to check types and subtypes, I completely omitted any discussion about what to do when these checks fail, that is, I did not discuss the art (or black magic) of producing meaningful error messages. I did not discuss it since this would lead us quite far away, but I want at least to stress that the set-theoretic interpretation of types comes quite handy in these situations. Type-checking fails only when a subtyping check does: the programmer wrote an expression of some type \(\mathcal{S}\) where an expression of type \(T\), not compatible with \(\mathcal{S}\), was expected. In order to produce a useful error message the system can compute the type \(\mathcal{S} \cap T\) and show to the programmer some default value in this type. This is an example of value that might be produced by the expression written by the programmer and make the program fail. Our experience with CDuce shows that in many occasions producing such a sample value is well worth any other explanation.

### 5. A roadmap to the theory that makes all this work

A survey on the “Types” mailing list traces the idea of interpreting types as sets of values back to Bertrand Russell and Alfred Whitehead’s Principia Mathematica. Closer to our interests it seems that the idea independently appeared in the late sixties early seventies and later back again in seminal works by Roger Hindley, Per Martin-Löf, Ed Lowry, John Reynolds, Niklaus Wirth and probably others. More recently, it was reused in the context of XML processing languages by Hosoya, Pierce, and Vuillon [27, 25, 24, 26].

The idea of using multiple definitions of methods to implement covariant specialization of binary methods was first proposed in my PhD thesis [9, 11] and the covariance and contravariance paper I wrote 20 years ago [10]. This technique was dubbed *encapsulated multi-methods* in [7] and implemented in different flavors for O₂ [5] and Java [6]. It was based on the type theory Giorgio Ghelli, Giuseppe Longo and I developed in [13, 14] which was the first formal type theory for multiple-dispatching: the conditions of *specialization soundness* (Definition 2.3) and *ambiguity freedom* (Definition 2.2) were first introduced there and are nowadays used by several multiple dispatching programming languages such as MultiJava [20], Fortress [1], Cecil [19], and Dubious [29].

In this essay I revisited those ideas in the framework of *semantic subtyping*, that is, a type theory with a full set of Boolean type connectives whose characterization is given in terms of a semantic interpretation into sets of values. The first works to use a semantic interpretation of types as sets of values in recent research in programming languages, are those by Hosoya, Pierce already cited above. Hosoya and Pierce used the semantic interpretation essentially to characterize unions of possibly recursive types, but were not able to account for higher order functions. The semantic subtyping type system is the first and, at this moment of writing, most
complete work that accounts for a complete set of Boolean connectives as well as for arrow types. It was defined by Alain Frisch, Véronique Benzenak, and myself in [22, 23]. A gentle introduction to the main concepts of semantic subtyping can be found in the article for the joint keynote talk I gave at ICALP and PPDP [12] while the most comprehensive description of the work is by far Alain Frisch’s PhD thesis [21], a remarkable piece of work that I strongly recommend if you are interested in this topic (and if you can read French).

The use of semantic subtyping to revisit the covariance vs. contravariance problem brings two important novelties with respect to the theory I used in the original co-contra-variance paper [10].
First, the use of intersection types brings a clear distinction between what belongs to the realm of the type theory and what to the design of the language, specifically, the definition of formation rules for expressions. In particular, we have seen that types, their theory, and their subtyping relation are defined independently from the particular language we apply them to. I tried to clearly stress that conditions such as those of ambiguity (Definition 2.2) and specialization (Definition 2.3) concern the definition of the expressions: they are given to ensure that expressions have a unambiguous semantics as well as definitions that match the programmer’s intuition, but they do not concern the theory of types. In [10], instead, this difference was blurred, since these conditions were given for the formation of types rather than for the formation of expressions. So, for instance, in [10] a type such as $24$ was considered ill formed while in the semantic subtyping framework this type is legitimate (it is rather the multi-subroutine declared of having such a type that is likely to be ill-formed.) A second, more technical difference is that in [10] it was not possible to compare an arrow with an intersection of arrows (arrows and intersections were considered different type constructors) and this posed problems of redundancy (what is the difference between a function and an overloaded function with just one arrow?) and modularity (it is not possible to specialize the type of a function by adding new code unless it is already overloaded; this, transposed to the context of this paper means that multi-methods can only override other multi-methods, thus hindering the solution of Section 3.3).

A big advantage of semantic subtyping is that it comes with robust and optimal algorithms that work well in practice. They are implemented in the language CDue, whose distribution is free and open-source [18] and the reader can use the interactive toplevel of that language to play and experiment with set-theoretic types. For instance, to test the subtyping relation of equation (1) for types Int and Char one can use the debug directive as follows (the hash symbol # is the prompt of the interactive toplevel while italics is used for the toplevel’s answer):
```plaintext
# debug subtype (((Int -> Char) & (Char -> Int)) ((Int | Char) -> (Int | Char)));
[DEBUG::subtype]
Char -> Int & Int -> Char <= (Char / Int) -> (Char / Int) : true
```

These algorithms, which are the ones I outlined in Section 4, are precisely defined in [23]. There the reader will find the formal proofs that the decompositions used in Step 4 are sound and complete (see in particular Section 6.2 of [23]). These two decompositions are the generalizations to type connectives of the classic subtyping rules for products and arrows as they are found in subtyping system with syntax-oriented definitions. This point can better be grasped by considering the particular cases of the intersections in (31) and (32) when both $P$ and $N$ contain exactly one type. Then checking the emptiness of (31) and (32) corresponds to checking the following two subtyping relations

$\left( S_1 , S_2 \right) \triangleleft \left( T_1 , T_2 \right)$

and we leave as exercises to the reader [EX9,EX10] to check that in these cases the two decomposition rules of Step 4 become, respectively:

$\left( S_1 : \emptyset \right) \lor \left( S_2 : \emptyset \right) \lor \left( S_1 : T_1 \land S_2 : T_2 \right)$

$\left( T_1 : \emptyset \right) \lor \left( T_2 : \emptyset \right)$

These are nothing but the classic subtyping rules specialized for the case in which types may be empty.

The theory of quasi-constant functions and its application to record types is described in Chapter 9 of of Alain Frisch’s PhD thesis [21], where the proof of soundness and completeness of the decomposition rules I gave in Section 4.5 can also be found.

The data structures I described are those used in the implementation of the language CDue and described in details in Chapter 11 of Alain Frisch’s PhD thesis [21]. The only difference is that the structures used in the compiler of CDue to implement BDDs and types, contain some extra fields, typically for storing hashes (to perform efficient comparison and hash-consing), and for pretty printing and error messages. Also the structure representing types includes more fields to account for extra kinds of atoms, namely, unicode characters, record types, and XML types.

Semantic subtyping is a general and streamlined theory that can be adapted to other settings. One of the most recent results about semantic subtyping is that it can be extended with parametric polymorphism. It is possible to add type variables to the types I presented in this paper (namely, those of the grammar in (27)) and leave to the type system the task to deduce how to instantiate them, as it is done in languages such as OCaml and Haskell. Explaining here how to do it would have lead us too far. The interested reader can refer to the article Polymorphic Functions with Set-Theoretic Types, published in two parts [16, 15], that describes how to define polymorphic functions, to type them and to implement them in an efficient way. The work on polymorphic functions is based on the extension of semantic subtyping to polymorphic types, which was defined in [17].

6. Philosophy and Lessons

I wrote this work as a challenge: to introduce sophisticated type theory to average functional programmers and to do it by using a popular programming language such as Perl that was not conceived with types in mind. The goal was to show that when a type system is well designed, it can be explained to programmers in very simple terms, even when its definition relies on complex theories that are prerogative of specialists: hopefully what programmers must retain of this system should all sum up to the 6+2 rules I gave in Sections 2.5 and 3.4. I pushed the experiment further and in Section 4 I tried also to explain to potential language designers, the main implementation techniques for these types. Once more, I aimed at demonstrating that it is not necessary to be a researcher to be able to implement this kind of stuff: so instead of explaining why, say, the decomposition rules in Step 4 of the subtyping algorithm are correct, I’d rather explained how to implement these decompositions in a very efficient way. Whether I succeeded in this challenge or not, is not up to me to say. I just hope that by reading this paper some eventual language designers will have learned a few basic notions and techniques so as not to start the design of their language from scratch.

Personally, what I learned from this work is that you should fit programming languages to types and not the other way round, insofar as a type theory should be developed pretty much independently from the language (but, of course, not from the problem) it is to be applied to. This observation is quite arguable and runs contrary to common practice according to which type theories are devel-
oped and fitted to overcome some problems in particular languages (even though it is what I have been doing for the last ten years with the semantic subtyping approach). I reached such a conclusion not because this paper adapts a type theory (semantic subtyping) to a language for which it, or any other type theory, was not conceived (Perl), but because it shows the limitations of my own work, that is, the type system (but was it really a type system?) my colleagues (Perl), but because it shows the limitations of my own work, that is, the type system for which it, or any other type theory, was not conceived (Perl).

In that system you were not allowed to have a type and I developed twenty years ago for the covariance vs. contravariance problem. In the type system (but was it really a type system?) my colleagues developed and fitted to overcome some problems in particular languages (Perl).

As long as it is clear that a function with that type applied to an argument of type will return results in it. It now becomes a problem of language design to devise function definitions that make it clear to the programmer that the functions she/he wrote have this property. A way to do that is to design the language so that whenever the type printed for and thought by the programmer is instead which, by the way, is the same type—. This is what the covariance condition does: when adding a new code for inputs to a function of type with the type force the programmer to write this code so that the return type declared by the programmer for this code is a subtype of , so that the types and are exactly the same. Likewise, a type with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In POPL ’14, 41st ACM Symposium on Principles of Programming Languages, pages 5–17, January 2014.

In conclusion, the deliberately provocative, lesson of this work is that in order to solve type-related problems, you must first conceive the types and only after you can think of how to design a language that best fits these types.

Acknowledgements

This work benefited from the reading and suggestions of many people. In particular, I want to thank the students of my course in advanced programming languages at the École Normale Supérieure de Cachan as well as the members of the language mailing list who found errors and gave several suggestions on how to improve the presentation, among whom Brent Laabs who suggested the encoding of Footnote 12, Yary Hluchan who suggested the code in Footnote 18 and Todd Hepler. Last but not least, I want to thank my colleagues Juliusz Chroboczek, Kim Nguyen, and Luca Padovani who provided useful feedback.

References


[23] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping:
A. Exercise solutions

[EX1] Problem: find a function that is in $S \setminus S_2 \rightarrow \Phi$ and not in $(S_1 \cup T_1) \& (S_2 \cup T_2)$. 

Solution: this is possible only if both $S_1 \rightarrow T_1 \neq \emptyset$, $S_2 \rightarrow T_2$ and vice versa. So let us suppose that both $S_1 \setminus S_2$, $S_1 \& \neg (S_2)$, that is the difference between $S_1$ and $S_2$ and $T_2 \setminus T_1$ are not empty types. Then, any function in $(S_1 \rightarrow T_1) \& (S_2 \rightarrow T_2)$ that maps at least one value in $S_1 \setminus S_2$ into a value in $T_2 \setminus T_1$ is a solution of the problem.

[EX2] Problem: prove the relations in Footnote 7 and their converse

Solution: Every function in $(S \rightarrow T) \& (S \rightarrow U)$ is a function that when applied to a value in $S$ it returns a result in $T$ and when applied to a value in $S$ it returns a result in $U$; this means that when such a function is applied to a value in $S$ it returns a result in $T \& U$ and, therefore, it is a function in $S \rightarrow T \& U$. Conversely, every function in $S \rightarrow T \& U$ when applied to a value in $S$ it returns a result in $T \& U$; thus, in particular, when it is applied to a value in $S$ it returns a result in $T$ and when applied to a value in $S$ it returns a result in $U$, and therefore it is $(S \rightarrow T) \& (S \rightarrow U)$. Every function in $(S \rightarrow T) \& (S \rightarrow U)$ is a function that when applied to a value in $S$ or to a value in $T$ it returns a result in $U$; therefore when it is applied to a value in $S$ it returns a result in $U$ and thus it is in $S \rightarrow T \rightarrow U$. Conversely, every function in $S \rightarrow T \rightarrow U$ when applied to a value in $S$ it returns a result in $U$, therefore, a fortiori, when and when applied to a value in $S$ it returns a result in $U$ and so does for values in $T$; therefore it is a function in $(S \rightarrow U) \& (T \rightarrow U)$.

[EX3] Problem: check that

| multi sub sum(Int $x$, Int $y$) { $x + y$ } |
| multi sub sum(Bool $x$, Bool $y$) { $x \& $y $ } |
| multi sub sum(Bool $x$, Int $y$) { sum($y$, $y$) } |
| multi sub sum(Int $x$, Bool $y$) { sum($y$, $x$) } |

has type $((\text{Int}, \text{Int}) \rightarrow \text{Int}) \& ((\text{Bool}, \text{Bool}) \rightarrow \text{Bool})$.

Solution: This is a recursive definition. So we have to prove under the hypothesis that the recursion variable $\sum$ has type (52) that the function has each type in the intersection. This means that we have to prove that if $\sum$ is applied to a pair of integers, then it returns an integer, and that if any of the two arguments is a Boolean then it returns a Boolean. This can be easily verified by checking the selection rules for multi subroutines. The delicate cases are the last two multi definitions since they contain a recursive call. Let us try to type the last definition (the other is similar), that is to deduce that if $x$ has type Int and $y$ has type Bool, then the result of sum is of type Bool. The type returned by this definition will be the type of the application $\sum(y, x)$. By hypothesis $\sum$ has the type in (52) and it is here applied to a first argument $y$ of type Boolean to a second argument $x$ of type Int. The type in (52) tells us that when a function of this type is applied to argument in $(\text{Bool}, \text{Int})$, then the function returns a result in $\text{Bool}$ (third arrow of the intersection in (52)), which is the expected result. and

[EX4] Problem: prove that the type in (11) and in Footnote 11 are equivalent.

Solution: The exercise in Footnote 7 states that $S \rightarrow T \& U$ is equivalent to $(S \rightarrow U) \& (T \rightarrow U)$ for all types $S$, $T$, and $U$. So in particular, $\text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}$ is equivalent to $(\text{Bool} \rightarrow \text{Bool}) \& (\text{Int} \rightarrow \text{Bool})$ from which the result follows.

[EX5] Problem: find a function that is in (16) and not in (17).

Solution: a simple example is the constant function 1:

| sub (Int $x$, Int $y$) { 1 } |

[EX6] Problem: Give a linear function to compute subtyping of product types.

Solution: Since an intersection of products is the product of the component-wise intersections, then without loss of generality we can consider the case in which the left hand side intersection of (31) is formed by a single product. Then we have the following recursive definition for the product case of the subtyping relation:

$$\Phi(S_1, S_2, N) \equiv \bigwedge_{(T_1, T_2) \in N} \bigwedge_{S \subset E} \Phi(S_1, S, T_1) \& \Phi(S_2, S, T_2)$$

where $\Phi(S_1, S_2, \emptyset) = \text{true}$.

The justification of the above definition and several possible optimizations for this algorithm can be found in Section 7.3.1 of [21].

[EX7] Problem: prove that a disjunctive normal form of tags can always be expressed by either $(t_1 \lor \ldots \lor t_n)$ and not $(t_1 \lor \ldots \lor t_n)$.

Solution: First of all notice that any $\sum$ can be represented as $\neg(\sum)$ the negation of an empty union of tags. Next, take any intersection $\sum_p \& \sum_q$. If $P$ is a singleton then the intersection is empty (the intersection of two distinct tags is always empty);
if there exist \( p \in P \) and \( n \in N \) such that \( t_p = t_n \), then the intersection is empty; if there exist \( p \in P \) and for all \( n \in N \) we have \( t_p \neq t_n \), then the intersection is \( t_p \) (since it is contained in all the negations of distinct tags). Therefore we are left with just two cases: either (i) \( P \) is a singleton and \( N \) is empty or (ii) \( P \) is empty: in all the other cases the intersection \( \bigwedge_{p \in P} t_p \land \bigwedge_{n \in N} \neg t_n \) reduces either to \( \text{Empty} \) or to \( \text{Any} \). In conclusion any intersection in a disjunctive normal form of tags is equivalent either to a single positive tag or to an intersection of negated tags. Now consider again the latter case, that is when the intersection is just formed by negated tags. The tags that are contained in \( \bigwedge_{p \in P} \neg t_p \) are exactly those in the co-finite union \( \bigvee_{N \in N} \neg t_n \). Now take any union formed only of such types (i.e., either a single positive tag or a co-finite union of tags). Such a union, our disjunctive normal form, is equivalent to either a finite union of tags (when the disjunctive normal forms unite only positive tags) or a co-finite union of tags (when the disjunctive normal form unites at least one co-finite union: all the positive tags in the disjunctive normal forms are removed from it and all the other co-finite unions and the cofinite unions are merged into a single one).

[EX8] Problem: define \( \text{norm} \)

Solution: I present the solution without recursive types. These can be easily included by using references, for instance. I use the notation \( \{ r \text{ with } \ell = v \} \) to denote the record in which the field \( \ell \) contains \( v \) and the remaining fields are as in the record \( r \).

```plaintext
let empty = {
  tags = 0 ;    // or \{ \}, positive
  ints = 0 ;    // or \{ \}, positive
  prod = 0 ;
  arrw = 0 ;
}

let any = {
  tags = 1 ;    // or \{ \}, negative
  ints = 1 ;    // or \{ \}, negative
  prod = 1 ;
  arrw = 1 ;
}

let norm = function
| Empty -> empty
| Any  -> any
| \[ t \]  -> \{ empty with tags = (\{t\}, positive) \}
| \[ i..j \] -> \{ empty with ints = (\{i..j\}, positive) \}
| \( S \rightarrow T \)  -> \{ empty with prod = (S,T)?1:0:0 \}
| \( S\rightarrow T \)  -> \{ empty with arrw = S\rightarrow T?1:0:0 \}
| \( S\mid T \)  -> \( \text{norm} S\lor \text{norm} T \)
| \( S\& T \)  -> \( \text{norm} S\land \text{norm} T \)
| \( \text{not}(T) \)  -> any\( \text{not}(\text{norm} T) \)
```

According to the above definition, the types that form the atoms of BDDs are not normalized. An alternative solution is to store them already normalized, that is returning \( \text{norm } S\lor \text{norm } T \)?1:0:0 instead of \( S\rightarrow T?1:0:0 \) (and similarly for arrows).

[EX9] Problem: Prove that by applying the Step 4 of the subtyping algorithm of Section 4.2 to \( S_1 \rightarrow S_2 \& \neg((T_1 \rightarrow T_2)) \) we obtain \( S_1 \rightarrow \text{Empty} \) or \( S_1 \rightarrow \text{Empty} \) or \( S_1 \rightarrow T_1 \) and \( S_2 \rightarrow T_2 \).

Solution: We have to check two cases, that is for \( N' = \emptyset \) and \( N' = N \). These yield:

\( S_1 \rightarrow \text{Empty} \) or \( S_2 \rightarrow T_2 \) and \( S_1 \rightarrow T_1 \) or \( S_2 \rightarrow \text{Empty} \).