COVARIANCE AND CONTRAVARIANCE: A FRESH LOOK AT AN OLD ISSUE
(A PRIMER IN ADVANCED TYPE SYSTEMS FOR LEARNING FUNCTIONAL PROGRAMMERS)

GIUSEPPE CASTAGNA

CNRS, Institut de Recherche en Informatique Fondamentale, Université Paris Diderot – Paris 7, Paris, France

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 they could, 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.

Finally, I took advantage of this opportunity to describe some undocumented advanced techniques of type-systems implementation that are known only to few insiders that dug in the code of some compilers: therefore, even expert language designers and implementers may find this work worth of reading.

1. INTRODUCTION

Twenty years ago I wrote an article titled "Covariance and contravariance: conflict without a cause" [13] 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 programming. Probably all that may have got to (some) programmers was that there was some theoretical paper that explained what each of covariance and contravariance was good for (with the associated reaction: "... so what?"). One reason for that, I think, is that at the time both type theory and programming languages were not developed enough to well explain the issue. I won't explain here again the whole article but the point is that in order to expose my argumentation I had to invent some constructions that did not look close to anything present in programming languages at the time: in particular I had to use weird "overloaded types" (these were sets of function types with

Key words and phrases: Object-oriented languages, type theory, subtyping, intersection types, overloading, semantic subtyping.

Preprint. First version 2013. This version: November 2018
Latest revision available at `http://www.irif.fr/~gc`

two eerie formation conditions), write functions by using a strange "&"-infix notation, and even in its most simple forms functions had to distinguish two kinds of parameters by separating them by a vertical bar. I am sure that alone any of these oddity was enough to put off any serious programmer.

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* [26], while to illustrate all the examples I chose to use the programming language *Perl 6* [34], 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. Forty 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.[1] 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 forty 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. So while all the expressions I will write can be run on any of the several Perl 6 implementations currently being developed, I will keep of Perl types just their syntax (and with several liberties), and give of them and of their subtyping relation my very personal interpretation. Although Perl 6 is not the best candidate to present this paper (the perfect candidate would be the programming language $\mathbb{C}$Duce [4, 21] 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 6. I am aware that this choice will make some collegue 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

[1] Though at that time I did not succeed to convince my parents to let me repair our family car (well, I was twelve).

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 sophisticated reasoning about programs and types. Section 3 applies the notions introduced in the primer to the covariance vs. contravariance issue. In that section I will present object-oriented programming in Perl 6, show why one can consider objects and classes as syntactic sugar to define some "multi subroutines", explain the issue of covariance and contravariance and their use in terms of these multi subroutines. Section 4 is my "electrical blueprint". It aims at language designers and implementers and explains how the semantic subtyping can be efficiently implemented, by describing the algorithms and data structures to be used to check type inclusion and perform type assignment. Section 5 is the one that you never find in a TV documentation and it gives a roadmap to the references that allow a researcher to understand the principles underlying semantic subtyping and why the algorithms of the preceding sections work (if you do not believe me and you need pointers to the formal proofs, then this is the section you need to look at). In Section 6 I try to summarize the lessons that can be drawn from this work and, in particular, what I learnt from writing it. The primer section contains several exercises whose solutions are given in appendix. All the article long I use footnotes to complete the main text with practical observations, missing definitions, and technical considerations for the advanced reader; the last kind of footnotes are marked by an asterisk and can be safely skipped.

This article has several possible keys of reading; an obvious one is that this paper can be taken as a proposal for a statically safe type-system for the functional core of Perl 6 and a specification of the algorithms to implement it. However, the actual motivation to write this article came from two distinct sources. The first motivation is that semantic subtyping explains the covariance/contravariance issue of [13] in much a cleaner way and without resorting to a somehow "ad hoc" formalism (as the $\lambda\&$-calculus of [13] was). So for a long time I have been wanting to reframe my old work in terms of semantic subtyping. But the motivation that spurred me to start writing it, is that few years ago I started to teach an undergraduate course on advanced programming at École Normale Supérieure de Cachan. Formerly I had been teaching issues about covariance and contravariance only at master level, and I then realized that while the problem is still relevant, the original explanation was too high-level for undergraduate students (i.e., it still contained too many λ's). The real target reader of this work is, thus, the undergraduate student of an advanced programming course, and this explains why all examples of this paper are written in a popular language and it does not contain a single "λ", theorem, or inference rule (just two formal definitions). So the not so hidden challenge tackled by this work is to explain sophisticated type theoretic concepts to a willing functional programmer. These two motivations also explain why I consider this work as both a *theoretical* and an *educational* pearl, in the sense of the ICFP and POPL conferences call for papers.

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[2]). 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

[2]* Formally, this can be seen by considering cardinalities: types and values are inductively defined and, therefore, they are recursively enumerable sets, while the powerset of values is not. More prosaically, the set of all the functions that terminate when applied to, say, 42 is hardly a type, since it is not possible to decide whether a generic function has this type or not.

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 is given (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:[3]

$$\texttt{T} ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid \texttt{(T , T)} \mid \texttt{T|T} \mid \texttt{T\&T} \mid \texttt{not(T)} \mid \texttt{T-->T}$$

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 $\{\texttt{true}, \texttt{false}\}$

`Int`: denotes the set that contains all the numeric constants: $\{\texttt{0, -1, 1, -2, 2, -3,...}\}$.

`Any`: denotes the set that contains all the values of the language.

$(\texttt{T}_1 , \texttt{T}_2)$: denotes the set that contains all the possible pairs (v_1, v_2) where v_1 is a value in $\texttt{T}_1$ and v_2 a value in $\texttt{T}_2$, that is $\{(v_1, v_2) \mid v_1 \in \texttt{T}_1 , v_2 \in \texttt{T}_2\}$.

$\texttt{T}_1 \mid \texttt{T}_2$: denotes the *union* of the sets denoted by $\texttt{T}_1$ and $\texttt{T}_2$, that is the set $\{v \mid v \in \texttt{T}_1 \text{ or } v \in \texttt{T}_2\}$

$\texttt{T}_1 \& \texttt{T}_2$: denotes the *intersection* of the sets denoted by $\texttt{T}_1$ and $\texttt{T}_2$, that is the set $\{v \mid v \in \texttt{T}_1 \text{ and } v \in \texttt{T}_2\}$.

`not(T)`: denotes the set of all the values that are not in the set denoted by `T`, that is $\{v \mid v \notin \texttt{T}\}$. So in particular `not(Any)` denotes the empty set.

$\texttt{T}_1\texttt{-->}\texttt{T}_2$: is the set of all function values that when applied to a value in $\texttt{T}_1$, if they return a value (i.e., if the application does not loop), then this value is in $\texttt{T}_2$.

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[4]) expression that defines a function. In λ-calculus we would say it is a lambda-abstraction.[5] In Perl 6 it is any expression of the form "**`sub`** `(`*parameters*`){`*body*`}`". So for instance **`sub`**`(Int $x){ return $x + 1 }` is a Perl 6 value that denotes the successor function.[6] Functions can be named (which will turn out to be quite handy in what follows) such as for **`sub`**`succ(Int $x){ return $x + 1 }` which gives the name `succ` to 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).[7]

---

[3] 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* `Bool` *or* `Int` *or* `Any` *or* $(\texttt{T}_1 , \texttt{T}_2)$ (where $\texttt{T}_1$ and $\texttt{T}_2$ are types) *or* $\texttt{T}_1 \mid \texttt{T}_2$ (where $\texttt{T}_1$ and $\texttt{T}_2$ are types) *or* ..." . Notice that to define, say, $(\texttt{T}_1 , \texttt{T}_2)$ we assumed to know what the types $\texttt{T}_1$ and $\texttt{T}_2$ that compose it are: this roughly corresponds to define types *by induction*.

[4] 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

[5] Lambda-abstractions have been recently integrated in languages such as `C#` and Java, where they are called "lambda-expressions".

[6] In Perl variables are prefixed by the dollar sign, and `sub` is the apocope for `sub`routine.

[7] * Notice that the definition of the set of values denoted by an arrow type does not depned on the inputs on which a function diverges. As a consequence, the function that diverges on all inputs belongs to every arrow type. This is the reason why the set denoted by an arrow type (or by any intersection of arrow types) is never empty: it always contains the function that diverges on all arguments.

2.2. **What is a subtype?** Perl 6 provides the constructor `subset` to define types such as `Nat`, `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 `%` is in Perl 6 the modulo operator):

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

(e.g., the first is Perl syntax for $Even = \{x \in \texttt{Int} \mid x \bmod 2 = 0\}$). Both these types are subsets of `Int`.[8] 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 both 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 informally said at the beginning of the section, subtyping is just set containment on values.

Definition 2.1 subtype**.** *A type T_1 is a* subtype *of a type T_2, written T_1 `<:` T_2, if all values in T_1 are values in T_2.*

While the subtyping relation `Int-->Int<:Even-->Int` holds, the converse (ie, `Even-->Int <:Int-->Int`) does not. For instance the division-by-two function

```
sub (Int $x){ $x / 2 }
```

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

$$(\texttt{S}_1\texttt{-->}\texttt{T}_1)\,\&\,(\texttt{S}_2\texttt{-->}\texttt{T}_2) \;\texttt{<:}\; (\texttt{S}_1\,\texttt{|}\,\texttt{S}_2)\texttt{-->}(\texttt{T}_1\,\texttt{|}\,\texttt{T}_2) \tag{2.1}$$

that holds for all types, $\texttt{S}_1$, $\texttt{S}_2$, $\texttt{T}_1$, and $\texttt{T}_2$, whatever these types are. A value in the type on the left hand-side of (2.1) is a function that if applied to an $\texttt{S}_1$ value it returns (if any) a value in $\texttt{T}_1$, and if it is applied to a value in $\texttt{S}_2$ it returns (if any) a value in $\texttt{T}_2$. Clearly, this is also a function that when applied to an argument that is either in $\texttt{S}_1$ or in $\texttt{S}_2$, it will return a value either in $\texttt{T}_1$ or in $\texttt{T}_2$ (from now on I will omit the "if any" parentheses). I leave as an exercise to the reader [EX1] the task to

[8] More precisely, I should have said "they denote subsets", but from now on I identify types with the sets of values they denote.

show that the inclusion is strict, that is, that there exists a value in the type on the right-hand side of (2.1) that is not in the type on left-hand side (see exercise 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 $\mathtt{S_1}\mathtt{-->}\mathtt{T_1}$ is included in another function type $\mathtt{S_2}\mathtt{-->}\mathtt{T_2}$. Let us first try to deduce it on a specific example: consider the function `sub double(Int $x){ x+x }`. This function is of type `Int-->Even`. It is easy to see that `double` (but also any other function in `Int-->Even`) is also a function in `Int-->Int`, since whenever it returns an `Even`, 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-->Even`, then we can deduce that `Int-->Even<:Odd-->Int`. This relation holds not only for `Int`, `Even`, and `Odd`, by for every pair of types whose sub-components are in a similar relation. Every function value in $\mathtt{S_1}\mathtt{-->}\mathtt{T_1}$ transforms values in $\mathtt{S_1}$ into values in T_1, then it is also a function that transforms values in any type $\mathtt{S_2}$ smaller than $\mathtt{S_1}$ into values that are contained in $\mathtt{T_1}$ and, thus, in any type $\mathtt{T_2}$ greater than $\mathtt{T_1}$. Since these two conditions are also necessary for containment, then we have following rule[9]:

$$(\mathtt{S_1}\mathtt{-->}\mathtt{T_1})\mathtt{<:}(\mathtt{S_2}\mathtt{-->}\mathtt{T_2}) \iff \mathtt{S_2}\mathtt{<:}\mathtt{S_1} \text{ and } \mathtt{T_1}\mathtt{<:}\mathtt{T_2} \tag{2.2}$$

We notice that while the "T" types have the same positions with respect to `<:` in both sides of equation (2.2), the "S" types have a different position on the right and left sides of (2.2). Borrowing the terminology from category theory, it is customary to say that the function type constructor "`-->`" 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 (2.2) and at least the following (pretty straightforward) ones:

$$(\mathtt{S_1}\mathtt{,}\mathtt{T_1})\mathtt{<:}(\mathtt{S_2}\mathtt{,}\mathtt{T_2}) \iff \mathtt{S_1}\mathtt{<:}\mathtt{S_2} \text{ and } \mathtt{T_1}\mathtt{<:}\mathtt{T_2} \tag{2.3}$$

$$\mathtt{S_1}\mathtt{<:}\mathtt{S_2} \text{ and } \mathtt{T_1}\mathtt{<:}\mathtt{T_2} \implies \mathtt{S_1}\mathtt{\&}\mathtt{T_1}\mathtt{<:}\mathtt{S_2}\mathtt{\&}\mathtt{T_2} \tag{2.4}$$

$$\mathtt{S_1}\mathtt{<:}\mathtt{S_2} \text{ and } \mathtt{T_1}\mathtt{<:}\mathtt{T_2} \implies \mathtt{S_1}\mathtt{|}\mathtt{T_1}\mathtt{<:}\mathtt{S_2}\mathtt{|}\mathtt{T_2} \tag{2.5}$$

$$\mathtt{S}\mathtt{<:}\mathtt{T} \iff \mathtt{not(T)}\mathtt{<:}\mathtt{not(S)} \tag{2.6}$$

However she is not required to be able to decide subtyping for every possible pair of types, since this requires a knowledge deeper than the few above rules (as an aside, notice that the rules (2.2)–(2.6) are not enough to deduce the subtyping relation in (2.1): technically one says that the above set of rules provides a sound but incomplete axiomatization of the subtyping relation[10]). The problem of

[9]* For the advanced reader, in the presence of an empty type, the two conditions on the right-hand side of (2.2) are sufficient but not necessary. If we use `Empty` to denote `not(Any)`—i.e., the empty type—, then the correct rule is: $(\mathtt{S_1}\mathtt{-->}\mathtt{T_1})\mathtt{<:}(\mathtt{S_2}\mathtt{-->}\mathtt{T_2})$ if and only if ($\mathtt{S_2}\mathtt{<:}\mathtt{S_1}$ and $\mathtt{T_1}\mathtt{<:}\mathtt{T_2}$) *or* ($\mathtt{S_2}\mathtt{<:}\mathtt{Empty}$). Likewise in the presence of an empty type the right-hand side condition of (2.3) later on are not necessary: $(\mathtt{S_1}\mathtt{,}\mathtt{T_1})\mathtt{<:}(\mathtt{S_2}\mathtt{,}\mathtt{T_2})$ if and only if ($\mathtt{S_1}\mathtt{<:}\mathtt{Empty}$) *or* ($\mathtt{T_1}\mathtt{<:}\mathtt{Empty}$) *or* ($\mathtt{S_1}\mathtt{<:}\mathtt{S_2}$ *and* $\mathtt{T_1}\mathtt{<:}\mathtt{T_2}$).

[10]* To deduce such a kind of relations one should also use axioms such as the following ones (excerpted from [3])

$$(\mathtt{S}\mathtt{-->}\mathtt{T})\mathtt{\&}(\mathtt{S}\mathtt{-->}\mathtt{U}) \mathtt{<:} \mathtt{S}\mathtt{-->}\mathtt{T}\mathtt{\&}\mathtt{U} \tag{2.7}$$

$$(\mathtt{S}\mathtt{-->}\mathtt{U})\mathtt{\&}(\mathtt{T}\mathtt{-->}\mathtt{U}) \mathtt{<:} \mathtt{S}\mathtt{|}\mathtt{T}\mathtt{-->}\mathtt{U} \tag{2.8}$$

The reader may try as an exercise to prove that both the relations in (2.7) (2.8) above and their converse hold [EX2].

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 $(\mathtt{S_1}\texttt{-->}\mathtt{T_1})\,\&\,(\mathtt{S_2}\texttt{-->}\mathtt{T_2})$ 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., `succ`— is not really "overloaded" since it always executes the same code both for arguments in `Even` and for arguments in `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.[11] However Perl 6 allows the programmer to define "real" overloaded functions by giving multiple definitions of the same function prefixed by the `multi` modifier:

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

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

$$\texttt{((Int , Int)-->Int)}\;\&\;\texttt{((Bool , Bool)-->Bool)}, \tag{2.9}$$

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 sum function in a *curried*[12] version I call `sumC`:

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

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

[11]* Such a usage of intersection types is known —in my opinion, somehow misleadingly— as *coherent overloading*: see Section 1.1.4 in [35]).

[12]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.

selected code (those on the right of "`;;`"). So by using the syntactic sugar "`;;`", the code above can be "equivalently" rewritten as:[13]

```
multi sub sumC(Int $x ;; Int $y) { $x + $y }
multi sub sumC(Bool $x ;; Bool $y) { $x && $y }
```

The type of the `sum` function has changed since in both `multi` definitions (either curried or with"`;;`") `sumC` has type:

$$(\texttt{Int}\texttt{-->}(\texttt{Int}\texttt{-->}\texttt{Int}))\ \&\ (\texttt{Bool}\texttt{-->}(\texttt{Bool}\texttt{-->}\texttt{Bool})). \tag{2.10}$$

The functions of this type are different from those of the type in (2.9): the type in (2.9) 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 (2.10) 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 `multi` definitions for `sum` which is then defined in the following (weird) recursive way:

```
multi sub sum(Int $x, Int $y) { $x + $y }
multi sub sum(Bool $x, Bool $y) { $x && $y }
multi sub sum(Bool $x, Int $y) { sum($x , $y>0) }
multi sub sum(Int $x, Bool $y) { sum($y , $x) }
```

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

$$\begin{array}{ll} & (\texttt{(Int , Int)}\texttt{-->}\texttt{Int}) \\ \& & (\texttt{(Bool , Bool)}\texttt{-->}\texttt{Bool}) \\ \& & (\texttt{(Bool , Int)}\texttt{-->}\texttt{Bool}) \\ \& & (\texttt{(Int , Bool)}\texttt{-->}\texttt{Bool}). \end{array} \tag{2.11}$$

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 multi function that will dispatch on the second argument. That is, we are looking for a function of the following type:

$$\begin{array}{r} (\texttt{Int}\ \texttt{-->}\ ((\texttt{Int}\texttt{-->}\texttt{Int})\ \&\ (\texttt{Bool}\texttt{-->}\texttt{Bool}))) \\ \&\ (\texttt{Bool}\texttt{-->}((\texttt{Bool}\texttt{-->}\texttt{Bool})\ \&\ (\texttt{Int}\texttt{-->}\texttt{Bool}))) \end{array} \tag{2.12}$$

The type above is the type of functions that when they are applied to an integer return another function of type `(Int-->Int) & (Bool-->Bool)` and that when they are applied to a Boolean return another function of type `(Bool-->Bool) & (Int-->Bool)`. In other terms they are functions that can be applied to a sequence of two arguments of type `Int` or `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.[14]

[13]* Strictly speaking this and the previous definition of `sumC` are not equivalent since the first application is curried and therefore it must be fed by one argument at a time (e.g., `sumC(3)(39)`) while in the latter it is not, so it receives all the arguments at once (e.g., `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.

[14]By equation (2.8) and the exercise in Footnote 10, the type in (2.12) is equivalent to:

$$(\texttt{Int}\ \texttt{-->}\ ((\texttt{Int}\texttt{-->}\texttt{Int})\ \&\ (\texttt{Bool}\texttt{-->}\texttt{Bool})))\ \&\ (\texttt{Bool}\texttt{-->}((\texttt{Bool}\texttt{|}\texttt{Int})\texttt{-->}\texttt{Bool}));$$

prove it as an exercise [EX4].

How is a function of the type in (2.12) defined? In particular how can we define `sumC` —the curried version of the second definition of `sum`— which should have this type? A first attempt consists in defining two auxiliary multi functions `sumI` and `sumB` that perform the dispatch on the second argument of `sumC` and that are called according to whether the first argument of `sumC` is an integer or a Boolean, respectively:

```
multi sub sumI(Int $y ;; Int $x){ $x + $y }
multi sub sumI(Bool $y ;; Int $x) { sumC($y,$x)}

multi sub sumB(Bool $y ;; Bool $x) { $x && $y }
multi sub sumB(Int $y ;; Bool $x) { sumC($x,$y>0) }

multi sub sumC(Int $x ;; Int|Bool $y) { sumI($y,$x) }
multi sub sumC(Bool $x ;; Int|Bool $y) { sumB($y,$x) }
```

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

```
  (Int --> (Int|Bool)-->(Int|Bool))
& (Bool-->(Int|Bool)-->Bool)
```
(2.13)

and this type is less precise than (i.e., it is a supertype of) the type in (2.12) (see also the type in Footnote 14 that is equivalent to the (2.12) type). For instance, the type in (2.12) states that when a function of this type is applied to two arguments of type `Int`, then the result will be of type `Int`, while the type in (2.13) states, less precisely, that the result will be of type `Int|Bool`.

The solution to define a version of `sumC` that has the type in (2.12) is not so difficult: instead of defining separate auxiliary functions (to which the second argument of `sumC` must be explicitly passed as an argument) I will use nested `multi` subroutines. Unfortunately Perl 6 does not allow the programmer to write anonymous `multi` functions. But if we suppose that anonymous `multi` functions were allowed (e.g., by considering anonymous `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 `sumC` and the type in (2.12) should be, I think, pretty clear:

```
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)}
      }
```
(2.14)

This code states that `sumC` is a multi-subroutine formed by two definitions each of which returns as result another (anonymous) multi-subroutine.[15]

[15] Although the code (2.14) cannot be executed in Perl 6 because of the absence of anonymous multi-subroutines, it can be faked by non-anonymous multi-subroutines as follows:

I have not done anything new here. As it will be clear later on, to define the code in (2.14) I just applied the "double dispatching" technique proposed by Ingalls in 1986 at the first OOPSLA conference [31], though I doubt that anybody at the time saw it as a special case of currying in the presence of intersection types: as a matter of fact, it is not so straightforward to see that the type in (2.12) is the curried version of the type in (2.11) (the code in (2.14) is actually a "proof" —in the sense of [9]— that currying is the relation between the type in (2.12) and the one in (2.11)).

2.4. **Formation rules for multi-subroutines.** In order to conclude the discussion about multi-subroutines I have to give some more details about how their "dynamic dispatch" is performed. When I first introduced multi-subroutines I said that the code to execute was chosen at run-time according to the type of the actual parameter: this is called *dynamic dispatch*, since the argument of the function is *dispatched* to the appropriate code *dynamically*, that is, at run-time. However, in all the examples I showed so far delaying the choice of the code until run-time does not seem a smart choice since in all of them it is possible to choose the code at compile time according to whether the arguments are Booleans or Integers. In other terms, the examples I gave are not conceptually different from operator overloading as found in Algol 68 where it is resolved at static time (i.e., during the compilation). Of course, making the dispatch at run-time is computationally much more expensive than making it at compile time. Therefore, dynamic dispatch is useful and justified only if the selection of the code may give at run time a result different from the one that would be obtained at compile time, that is to say, if the type of function arguments may evolve during the computation. In statically-typed languages[16] this happens in the presence of a subtyping relation: without subtyping an expression in a statically-typed language must have the same type the whole computation long; with a subtyping relation, instead, the type of an expression may change during the computation, notably decrease:

> *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 *static* and *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, `(3+2)`:

```
(sub(Int $x){$x+1})(3+2)
```

At compile time the function in the expression above has type `Int-->Int` (i.e., the usual type of the successor function) while the argument has type `Int` (i.e., the type of the application of the

```
multi sub sumC(Int $x){
           multi sub foo(Int $y) { $x + $y }
           multi sub foo(Bool $y) { sumC($y)($x) }
           &foo
         }
multi sub sumC(Bool $x){
           multi sub foo(Bool $y) { $x && $y }
           multi sub foo(Int $y) {sumC($x)($y>0)}
           &foo
         }
```

where `&foo` is Perl's notation to return the subroutine named `foo`.

[16] In a typed programming language, types can be checked either at compile-time or a run-time. Languages that check types at compile time are called statically-typed langages, while those that check types at run-time are said to be dynamically typed. A combination of static and dynamic type-checking is also possible. In this work I focus on how to define static type-checking for Perl, which currently is dynamically typed.

plus operator to two integers); therefore the whole expression has (static) type `Int`. The first step of execution computes the argument expression to `5`; the type of the argument has thus decreased from `Int` to `Odd`, while the whole application has still type `Int` (the application of a function of type `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 `Even`.

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.

```
multi sub mod2sum(Even $x , Odd $y) { 1 }
multi sub mod2sum(Odd $x , Even $y) { 1 }
multi sub mod2sum(Int $x , Int $y) { 0 }
```
(2.15)

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:

```
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 type. More generally, for every value v of type `T` in the language I assume the following definition as given

```
subset v of 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)
```
(2.16)

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 (2.15) 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)
```
(2.17)

This type is strictly more precise than (i.e., it is a strict subtype of) the type in (2.16) (as an exercise [EX5], we invite the reader to find a function value that is in the type in (2.16) and not in type in (2.17)). 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 $\mathtt{T}_1$, $\mathtt{T}_2$, S, the type $\mathtt{T}_1|\mathtt{T}_2\mathtt{-->S}$ is equivalent to $(\mathtt{T}_1\mathtt{-->S})\&(\mathtt{T}_2\mathtt{-->S})$, since every function in the former type has both the arrow types in the intersection, and viceversa (*cf.* the exercise of Footnote 10). By applying these two equivalences to (2.17) (in particular to the last factor of the top-level intersection) we obtain that (2.17) is equivalent to

```
   ((Even , Odd)-->1)
&  ((Odd , Even)-->1)
&  ((Odd , Odd)-->0)
&  ((Even , Even)-->0)
```
(2.18)

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 (2.18) (equivalently, of type (2.17)) to an argument of type `(Int , Int)`: as a matter of fact, no arrow in (2.18) has `(Int , Int)` as domain. Of course it is 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 (2.18) is `(Even,Odd)|(Odd,Even)|(Odd,Odd)|(Even,Even)`, that is, `(Int , Int)`. So a function with type (2.18) 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 n `multi` expressions—call them *branches*—, the i-th branch being defined for inputs of type $\mathtt{T}_i$. Apply this function to an argument that evaluates to a value of type T. The possible candidates for the selection are all the branches whose input type $\mathtt{T}_i$ is a supertype of T. Among them we choose the one defined for the least $\mathtt{T}_i$, 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 bump 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:

```
multi sub mod2sum(Even $x , Int $y){ $y % 2 }
multi sub mod2sum(Int $x , Odd $y){ ($x+1) % 2 }
```
(2.19)

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:

according to the current Perl semantics the execution is stuck.[17] Clearly such a problem can and must be statically detected (though, current implementations of Perl 6 detect it only at run-time). This can be done quite easily as explained hereafter. Consider again our silly definition (2.19) of `mod2sum` which is composed by two branches (i.e., pieces of code) one for inputs of type `(Even , Int)`, the other for inputs of type `(Int , Odd)`. Consider the intersection of these two types, `(Even , Int) & (Int , Odd)`, which is equal to `(Even , Odd)`. The function `mod2sum` can be applied to values in this intersection, but there does not exist a most specific branch to handle them. In order to avoid selection ambiguity, we have to perform systematically such a check on the intersections of the input types, as stated by the following definition:

Definition 2.2 Ambiguity**.** *Let* $\mathtt{T}_h$ *denote the input type of the h-th branch of a multi-subroutine composed of n branches. The multi-subroutine is* free from ambiguity *if and only if for all i,* $j \in [1..n]$ *and* $i \neq j$*, either* $\mathtt{T}_i\&\mathtt{T}_j$ *is empty (i.e.,* $\mathtt{T}_i\&\mathtt{T}_j\mathtt{<:not(Any)}$*), or the set* $\{\mathtt{T}_h \mid \mathtt{T}_i\&\mathtt{T}_j\mathtt{<:}\mathtt{T}_h, h \in [1..n]\}$ *has a unique least element.*[18]

In words, although we must test that for all possible types of the arguments, there always exists a most specific branch, in practice it suffices to test such existence for the types that are a non-empty intersection of the input types of two distinct branches. A compiler must reject any multi-subroutine that is not free of ambiguity.

Ambiguity freedom is a formation condition for the definition of multi-subroutines 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

```
( (Even,Int) --> 0|1 ) & ( (Int,Odd) --> 0|1 )        (2.20)
```

which, intuitively, is the type that corresponds to the multi subroutine in (2.19): a function of this type can be applied to arguments of type `(Int , Int)` and the result will be given the type `0|1`; the definition of `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 `(Even , Int)` and `(Int , Odd)` since they make the run-time selection (thus, the semantics) of the multi-subroutine undefined. In [13] 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 `mod2sum` as I defined it in (2.15). If we apply the function to a pair of expressions of type `Int`, —e.g., `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 and an

[17]* Two solutions alternative to being stuck are to choose 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 (2.19)) but they are seldom used: the use of some priority corresponds to using *class precedence lists* in object-oriented languages with multiple inheritance (e.g., this is what is used in Common Lisp Object System), while I am not aware of any language that uses random selection.

[18]The condition of uniqueness ensures that all the $\mathtt{T}_h$ are (semantically) distinct.

odd number —e.g., `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 `sum` and `sumC` which in the first version I gave for them are formed of two branches returning different types (`Int` and `Bool` for the former, and `Int-->Int` and `Bool-->Bool` for the latter, as respectively stated by the types in (2.9) and (2.10)). 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 `Int` is applied to an expression of type `Int`. Before performing the application we evaluate the argument. Therefore the value returned at runtime by this expression must be of type `Int` or of a smaller type, say `Even`, but not of a distinct type such as `Bool`, since the function `f` expects arguments of type `Int`, not of type `Bool`. 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 $\mathtt{T}_i$ (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 $\mathtt{T}_h$. These are the branches that *specialize* (I will also say *override* or *refine*) the branch $\mathtt{T}_h$ 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* $\mathtt{T}_h$ *and* $\mathtt{S}_h$ *respectively denote 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,* $j \in [1..n]$, $\mathtt{T}_i\mathtt{<:}\mathtt{T}_j$ *implies* $\mathtt{S}_i\mathtt{<:}\mathtt{S}_j$.

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 (2.20) which intersects two arrows whose domains have a common lower bound, so the type system has no problem in considering functions of type:

$$(\mathtt{Int-->Odd})\ \mathtt{\&}\ (\mathtt{Even-->Int}) \tag{2.21}$$

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. 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 (2.20). This type is not very interesting since both arrows that compose it return the same type: now we know (see Footnote 10) that $(\mathtt{S}_1\mathtt{-->T})\mathtt{\&}(\mathtt{S}_2\mathtt{-->T})$ is equivalent to $\mathtt{S}_1\mathtt{|}\mathtt{S}_2\mathtt{-->T}$ and therefore (2.20) is equivalent to a single arrow without intersections (i.e., `((Even,Int)|(Int,Odd))-->(0|1)`). In order to make the type more interesting let us modify it by using two distinct output types whose

intersection and union are non trivial. For instance, let us consider the following type:

$$\texttt{((Even,Int)-->(0,Int))\&((Int,Odd)-->(Int,1))} \tag{2.22}$$

A function of this type accepts any pair of integers that is not in `(Odd,Even)`: the domain of the function is the union of the domains of the arrows, that is `(Even,Int)|(Int,Odd)`; 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 (2.22) 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 $(x \,\mathbf{mod}\, 2, y \,\mathbf{mod}\, 2)$.

Let us now examine all the possible cases for an application of a function of the type in (2.22). 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

$$\texttt{(S1-->T1)\&(S2-->T2)} \tag{2.23}$$

to an argument of type `T`. Only four cases are possible:[19]

(1) If the argument is in `S1&S2` (i.e., if `T<:S1&S2`), then the application has type `T1&T2`.
(2) If the argument is in `S1` and case 1 does not apply (i.e., `T<:S1` and `T\S2` is not empty), then the application has type `T1`.
(3) If the argument is in `S2` and case 1 does not apply (i.e., `T<:S2` and `T\S1` is not empty), then the application has type `T2`.
(4) If the argument is in `S1|S2` and no previous case applies, then the application has type `T1|T2`.

Of course things become more complicated with functions typed by an intersection of three or more arrows: the complete formalization 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 soundness rule. So let us consider the type in (2.23) and suppose that, say, `S2<:S1`. What happens when we apply a function of this type to an argument of type `S2`? Since `S2<:S1` then `S2<:S1&S2`, therefore it is the first of the four possible cases that applies: the argument is in `S1&S2` and thus the result will be of type `T1&T2` (... which is smaller than `T2`).

Let us rephrase what we just discovered: if we have a function of type `(S1-->T1)&(S2-->T2)` with `S2<:S1`, and we apply it to an argument of the smaller type, that is `S2`, then the result will be in `T1&T2`. The fact that `T2` is not smaller than `T1` does not matter because the typing rules tell us that the application of this function to an argument of type `S2` will return a result in a type smaller than `T1`. Let us consider a concrete example for $\texttt{S}_i$, $\texttt{T}_i$ by going back to the type in (2.21). A function

[19] There is a mildly interesting fifth case when the type `T` of the argument is empty, that is when the argument is statically know to diverge.

is of type `(Int-->Odd) & (Even-->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(S1 $x) returns T1 { ... }
multi sub foo(S2 $x) returns T2 { ... }
```

In Perl 6 the `returns` keyword declares the result type of a subroutine. In the example above, it declares that the first definition has type $S_1 \texttt{-->} T_1$ and the second one type $S_2 \texttt{-->} T_2$. If each definition is well typed, then the programmer expects the multi definition above to have type $(S_1 \texttt{-->} T_1)\texttt{\&}(S_2 \texttt{-->} T_2)$. Suppose that $S_2 \texttt{<:} S_1$. In this case the typing rules state that a function has type $(S_1 \texttt{-->} T_1)\texttt{\&}(S_2 \texttt{-->} T_2)$ only if whenever it is applied to an argument of type S_2 it returns a result in $T_1\texttt{\&}T_2$. 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 $(S_1 \texttt{-->} T_1)\texttt{\&}(S_2 \texttt{-->} T_2)$ if and only if $T_2 \texttt{<:} T_1\texttt{\&}T_2$ and —by a simple set-theoretic reasonment— if and only if $T_2 \texttt{<:} T_1$: exactly the condition enforced by Definition 2.3.

To say it otherwise: the type system does not have any problem in managing a type of the form $(S_1 \texttt{-->} T_1)\texttt{\&}(S_2 \texttt{-->} T_2)$ even if $S_1 \texttt{<:} S_2$ 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 $T_1\texttt{\&}T_2$. In order to avoid this confusion 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) $T_1\texttt{\&}T_2$. This can be obtained by accepting only specialization 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 allows us to deduce non-trivial subtyping relations on complex types. I have also tried to distinguish concepts that concern the semantics 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 `S-->T` is the set of all functions that when applied to values in `S` return only results in `T`;
   (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:

```
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,

```
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: in Perl, `my` is the keyword to define a variable in the current block) 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, `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.

```
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

```
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 `$a` has static type `Point`, for which no `iswhite` method is defined (the wordings "*dynamic dispatch*" and "*dynamic binding*" come from there). On the contrary, an invocation such as `$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 `$a.move(3,3)` I wrote a few lines above: since `$a` is of (static) type `Point`, then the type 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)-->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 (2.2) states that `(Int , Int)-->ColPoint <: (Int , Int)-->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 (2.2) 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 [8]. The paradigmatic example is the `equal` method which, for the class `Point`, can be defined as follows.

```
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:

```
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:

```
my Point $a = ColPoint.new(x=>2,y=>21,c=>"white");
my Point $b = Point.new(x=>2,y=>21);
$a.equal($b);
```

The code above passes static type-checking: in the first line a `ColPoint` object is used where a `Point` instance is expected—which is legitimate—; in the second line we simply create a new object; while in the last line we send the message `equal` to the object `$a` with argument `$b`: since `$a` is (statically) a `Point` object, then it is authorized to send to it the message `equal` with a `Point` argument. However, at run-time `$a` is bound to a color point and by, dynamic binding, the method in the definition of `ColPoint` is used. This tries to access the `c` instance variable of the argument `$b` thus yielding a run-time error: the static type system is unsound.[20]

To have soundness and use color points where points are expected the type of the parameter of the `equal` method in `ColPoint` must be either `Point` or a supertype of it.

Despite this problem, covariant overriding had (has?) its strenuous defenders who advocated that they wanted both to use color points where points were expected and to define an `equal` method that compared color points with other color points and not just with points. The various solutions proposed (perform a dynamic check for method arguments —as in O_2—, adopt covariant specialization despite its unsoundness —as in Eiffel— or simply do not care about parameter types in overriding methods— as in Perl 6) were all incompatible with static soundness of the type system (or, as for LOOM [10], they gave up subtyping relation between `Point` and `ColPoint`). Thus contravariant overriding and covariant specialization looked as mutually exclusive typing disciplines for inheritance, and the dispute about the merits and demerits of each of them drifted towards a religious war. In what follows I will use the type theory we learned in the primer contained in Section 2 to argue that covariance and contravariance are not opposing views, but distinct concepts that can coexists in the same object-oriented language. But first I have to transpose the whole discussion in the setting of Section 2 where objects were absent. This is quite easy since if we abstract from some details (e.g., encapsulation, implementation, access), then the object-oriented part of Perl 6 is all syntactic sugar.

3.3. **It is all syntactic sugar.** Consider a class definition in Perl 6. It is composed of two parts: the part that describes the class's objects (i.e., the fields that compose them) and the part that describes the operations on the class objects (i.e., the methods). Methods are nothing but functions associated to a name (the message) and with an implicit parameter denoted by `self`. Thus a method such as the one defined for `move` in class `Point` is a function with 3 parameters, one of type `Point` (denoted by `self`) and two of type `Int`. As a matter of fact, if we write:

```
my Point $a = Point.new(x => 23, y => 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 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.

[20] * 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 cheking 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 remove method definitions from classes and replace them by multis where the receiver parameter has become an explicit parameter, then we cannot observe any difference from an operational point of view. In other terms we can rewrite the first two class definitions of Section 3.1 as follows and obtain something that is observationally equivalent

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

multi sub origin(Point $self) {
   ($self.x==0)&&($self.y==0)
};

multi sub move(Point $self ;; Int $dx, Int $dy) {
   $self.x += $dx;
   $self.y += $dy;
   return $self;
};

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

multi sub iswhite(ColPoint $self) {
   return ($self.c=="white");
};

multi sub move(ColPoint $self ;; Int $dx, Int $dy) {
   if not(iswhite($self)) {
      $self.x += $dx;
      $self.y += $dy;
   }
   return $self;
};
```

Notice that class definitions now contain only instance variable declarations. In practice, classes have become record types (i.e., respectively *hashes* and *structures* in Perl and C) whose subtyping relation is explicitly defined.[21] Method definitions are external to class definitions and have become multi subroutines enriched with an extra parameter `$self` whose type is used to select the code to execute and, as such, it is separated from the other parameters by "`;;`". Finally, method invocation has become function application. This is shown by the body of the method `move` for `ColPoint` where the method invocation `self.iswhite()` has been transformed into `iswhite($self)`.

The transformation we just defined tells us that from a point of view of types the two class definitions at the beginning of Section 3.1 are equivalent to defining two (record) types `Point` and `ColPoint` (the latter subtype of the former) and three multi-subroutines respectively of type:

[21]* Technically, in this case one speaks of *name* subtyping, insofar as we *declared* `ColPoint` to be a subtype of `Point`.

```
origin:  Point-->Bool
iswhite: ColPoint-->Bool
move:   (Point-->((Int , Int)-->Point)) & (ColPoint-->((Int , Int)-->ColPoint))
```

The formation rules of multi-subroutines are satisfied, specifically for `move` where `ColPoint<:Point` requires by Definition 2.3 that `((Int , Int)-->ColPoint)<:((Int , Int)-->Point)`. Since the latter holds, then the above definitions —thus, the class definitions at the beginning of Section 3.1— are type sound.

If we apply the same transformation to the `equal` method we obtain

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

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:

$$\begin{array}{l}\texttt{equal:}\quad (\texttt{Point-->}(\texttt{Point-->Bool}))\\ \qquad\texttt{\&}\ (\texttt{ColPoint-->}(\texttt{ColPoint-->Bool}))\end{array} \tag{3.1}$$

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 (3.1) only if when applied to a `ColPoint` object it returns a value in the following intersection type:

$$(\texttt{Point-->Bool})\texttt{\&}(\texttt{ColPoint-->Bool}) \tag{3.2}$$

(*cf.* the first of the cases listed right after equation (2.23) 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

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

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

$$\begin{array}{l}\texttt{equal:}\quad (\texttt{(Point , Point)-->Bool})\\ \qquad\texttt{\&}\ (\texttt{(ColPoint , ColPoint)-->Bool})\end{array} \tag{3.3}$$

In particular, the specialization formation rule of Definition 2.3 is now satisfied: since we have `(ColPoint , ColPoint)<:(Point , Point)`, then 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:

```
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`:

```
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 case 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 of the method 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 (3.2), 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 (3.1) 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.[22]

> 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 [7] and [6] to see how the second choice can be implemented in Java and O_2, 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 great help.

---

[22]* 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 3.2, since in that case the (multi) method of `Point` is not overridden in `ColPoint`:

```
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.

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 data-structures needed to implement the type system of the previous section. I will give a bare description of the algorithms and data-structures 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.

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 `Bool` and `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

$$\texttt{T} ::= t \mid \texttt{[}i\texttt{..}j\texttt{]} \mid \texttt{Any} \mid \texttt{(T , T)} \mid \texttt{T-->T} \mid \texttt{T|T} \mid \texttt{T\&T} \mid \texttt{not(T)}$$

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 `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 Perl's "barewords" (actually, barewords are deprecated in Perl). When used as a type, a tag denotes the singleton containing that tag. In particular, I can encode `Bool` as the union of two tag types, `‘true|‘false`. The remaining types have the same interpretation as before. Tags, intervals, products and arrows are called *type constructors*, while unions, intersections, and negations are called *type connectives*. I will use `Empty` to denote the type `not(Any)`, and $S \setminus T$ to denote the set theoretic difference of two sets S and T (i.e., $S \cap \neg T$).

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 $\texttt{T} = \texttt{T|T}$ (which does not carry any information

about the set of values it denotes) or $\mathtt{T} = \mathtt{not(T)}$ (which cannot represent any set). A consequence of the second restriction is also that we can express only finite unions and intersections.

The reader disconcerted by infinite trees can consider, instead, the following inductive definition of types, that contains an explicit term for recursive definitions and that is equivalent to the previous definition stated in terms of infinite trees:

$$\textbf{Atoms}\quad a ::= t \mid \mathtt{[}i\,..\,j\mathtt{]} \mid \mathtt{(T\,,T)} \mid \mathtt{T\text{-}\text{-}>T} \tag{4.1}$$

$$\textbf{Types}\quad \mathtt{T} ::= a \mid X \mid \mathtt{rec}\,X\mathtt{=}a \mid \mathtt{T|T} \mid \mathtt{T\&T} \mid \mathtt{not(T)} \mid \mathtt{Any} \tag{4.2}$$

Although this last definition is probably easier to understand, I will mainly work with the one with infinite trees (since it yields simpler formulations of the algorithms and is closer to actual implementation) and will reserve the recursive types notation for the examples. Nevertheless the last definition is important because it clearly separates type constructors (i.e., the meta-operators that construct atoms) from type connectives and shows that there are four different kinds of constructors: tags, intervals, products, and arrows. An important property that I will use in what follows is that for each of these four kinds it is possible to define a top type that contains all and only the types of the same kind, namely.

(1) $\mathtt{(Any\,,Any)}$ is the greatest product type. It contains all the pairs of values. I use $\mathtt{Any}_{\mathtt{prod}}$ to denote it.
(2) $\mathtt{Empty\text{-}\text{-}>Any}$ is the greatest function type. It contains all the functions. I use $\mathtt{Any}_{\mathtt{arrw}}$ to denote it.
(3) $\mathtt{[*..*]}$ (i.e., $\mathtt{Int}$) is the largest interval. It contains all the integers. I use $\mathtt{Any}_{\mathtt{ints}}$ to denote it.
(4) $\mathtt{not(Any}_{\mathtt{prod}}\mathtt{|Any}_{\mathtt{arrw}}\mathtt{|Any}_{\mathtt{ints}}\mathtt{)}$ contains all the tag values. I use $\mathtt{Any}_{\mathtt{tags}}$ to denote it.

A final consideration before describing the subtyping algorithm. Types are possibly infinite trees that denote possibly infinite sets of values, but the theory presented here accounts only for *finite* values: there is not such a value as, say, an infinite list. So while the type $\mathtt{rec}\,X\mathtt{=`nil|(Int\,,}\,X\mathtt{)}$ is the type of the finite lists of integers, the type $\mathtt{rec}\,X\mathtt{=(Int\,,}\,X\mathtt{)}$ is the empty type.

4.2. **Subtyping algorithm.** The key property the subtyping algorithm is based on, is that types are sets. Since subtyping is set-containment, then the algorithm uses classic set-theoretic transformations to simplify the problem. For instance, to prove that $\mathtt{(T_1\,,T_2)\&(S_1\,,S_2)}$ is empty (i.e., $\mathtt{(T_1\,,T_2)\&(S_1\,,S_2)<:Empty}$), the algorithm uses set-theoretic equivalences and decomposes the problem into simpler subproblems, namely: $\mathtt{(T_1\,,T_2)\&(S_1\,,S_2)}$ is empty if and only if $\mathtt{(T_1\&S_1\,,T_2\&S_2)}$ is empty, if and only if either $\mathtt{T_1\&S_1}$ is empty or $\mathtt{T_2\&S_2}$ 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 $\mathtt{S<:T}$ holds 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 $\mathtt{S<:T}$ into $\mathtt{S\&not(T)<:Empty}$.

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 (4.1). A *literal* (ranged over by ℓ) is either an atom or the negation of an atom: $\ell ::= a \mid \mathtt{not(}a\mathtt{)}$. 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 `Any` and `Empty` are, respectively, the empty intersection and the empty union. Therefore, the second step of the algorithm consists in transforming the type `S&not(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, $\texttt{(}\mathtt{T}_1\texttt{,}\mathtt{T}_2\texttt{)}$ and $\mathtt{S}_1\texttt{-->}\mathtt{S}_2$, 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, $\texttt{(}\mathtt{T}_1\texttt{,}\mathtt{T}_2\texttt{)}$ and $\texttt{not(}\mathtt{S}_1\texttt{-->}\mathtt{S}_2\texttt{)}$, 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., $\texttt{not((}\mathtt{T}_1\texttt{,}\mathtt{T}_2\texttt{))\&not(}\mathtt{S}_1\texttt{-->}\mathtt{S}_2\texttt{)}$ can be split into the union of $\texttt{not((}\mathtt{T}_1\texttt{,}\mathtt{T}_2\texttt{))\&(Any,Any)}$ and $\texttt{not(}\mathtt{S}_1\texttt{-->}\mathtt{S}_2\texttt{)\&(Empty-->Any)}$).

Therefore, the third step of the algorithm performs these simplifications so that the problem is reduced to deciding emptiness of intersections 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"):

$$\bigwedge_{p\in P} t_p \;\&\; \bigwedge_{n\in N} \texttt{not(}t_n\texttt{)} \tag{4.3}$$

$$\bigwedge_{p\in P} \texttt{[}i_p\texttt{..}j_p\texttt{]} \;\&\; \bigwedge_{n\in N} \texttt{not([}i_n\texttt{..}j_n\texttt{])} \tag{4.4}$$

$$\bigwedge_{(\mathtt{S}_1\texttt{,}\mathtt{S}_2)\in P} \texttt{(}\mathtt{S}_1\texttt{,}\mathtt{S}_2\texttt{)} \;\&\; \bigwedge_{(\mathtt{T}_1\texttt{,}\mathtt{T}_2)\in N} \texttt{not((}\mathtt{T}_1\texttt{,}\mathtt{T}_2\texttt{))} \tag{4.5}$$

$$\bigwedge_{\mathtt{S}_1\texttt{-->}\mathtt{S}_2\in P} \texttt{(}\mathtt{S}_1\texttt{-->}\mathtt{S}_2\texttt{)} \;\&\; \bigwedge_{\mathtt{T}_1\texttt{-->}\mathtt{T}_2\in N} \texttt{not(}\mathtt{T}_1\texttt{-->}\mathtt{T}_2\texttt{)} \tag{4.6}$$

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 (4.3) and (4.4)—, then emptiness can be immediately decided: an intersection as in equation (4.3) 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 $t_p = t_n$) or two distinct tags appear in positive position (i.e., there exists $p_1, p_2 \in P$ such that $t_{p_1} \neq t_{p_2}$), while whether an intersection as in equation (4.4) 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 (4.5), we decompose the problem into checking that for all $N' \subseteq N$

$$\left(\bigwedge_{(\mathtt{S}_1\texttt{,}\mathtt{S}_2)\in P} \mathtt{S}_1 \;\texttt{<:}\; \bigvee_{(\mathtt{T}_1\texttt{,}\mathtt{T}_2)\in N'} \mathtt{T}_1\right) \text{ or } \left(\bigwedge_{(\mathtt{S}_1\texttt{,}\mathtt{S}_2)\in P} \mathtt{S}_2 \;\texttt{<:}\; \bigvee_{(\mathtt{T}_1\texttt{,}\mathtt{T}_2)\in N\setminus N'} \mathtt{T}_2\right) \tag{4.7}$$

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

To decide emptiness of the type in (4.6), we decompose the problem into checking whether there exists $\mathtt{T}_1\mathtt{-->}\mathtt{T}_2 \in N$ such that $\mathtt{T}_1 \leq \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_1$ and for all $P' \subsetneq P$ (notice that containment is strict)

$$\left(\mathtt{T}_1 \quad \mathtt{<:} \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P'} \mathtt{S}_1\right) \text{ or } \left(\bigwedge_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P\setminus P'} \mathtt{S}_2 \mathtt{<:} \quad \mathtt{T}_2\right) \tag{4.8}$$

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 (4.7) and (4.8) in Step 4. I focus on the algorithm for (4.8) since it is the most difficult one and leave the case for products as an exercise. Given a $\mathtt{T}_1\mathtt{-->}\mathtt{T}_2 \in N$ I first need to check whether $\mathtt{T}_1 \leq \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_1$ and, if so, then compute a function, say, Φ that given $\mathtt{T}_1$, $\mathtt{T}_2$ and P checks whether for every strict subset P' of P, the formula in (4.8) holds. The definition of Φ is not straightforward, so let me introduce it gradually.

Let me first slightly simplify the problem by relaxing the strictness of the containment: I will define a function Φ that checks (4.8) for every (possibly non-strict) subset P' of P, that is, for P too. Indeed, notice that the requirement of strictness in $P' \subsetneq P$ is just an optimization to avoid a useless check: we already know that for $P' = P$ the formula (4.8) is true (because $\mathtt{T}_1 \leq \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_1$ holds). Therefore strictness is not needed for the correctness the algorithm. I leave the implementation of the strictness optimization for Φ as an exercise.

The basic idea to program the function Φ is the following: pick an element e in P (the choice of e can be arbitrary) and let Q be $P \setminus \{e\}$; then Φ does two things: (i) it recursively solves the problem for all subsets of Q (i.e., the subsets of P that do not contain e) and (ii) it recursively solves the problem for all subsets of Q to which e is added (i.e., all the subsets of P that contain e). To avoid to repeat calculations in different recursive calls, Φ keeps track of three sets of elements of P: the set of elements that are still to be selected to form some subset P' of P (denoted by P°); the set of elements that were already selected to be in P' (denoted by P^+); and the set of elements that were already selected not to be in P' (denoted by P^-). A generic call for Φ will then be of the form $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P^\circ\ ,\ P^+\ ,\ P^-)$ such that $P = P^\circ \cup P^+ \cup P^-$; such a call will pick an element $\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ$ in P° and (i) recursively solve the problem for all subsets of $Q = P^\circ \setminus \{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}$ by adding the element to P^- and thus recording that the element must not be considered, that is $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ Q\ ,\ P^+\ ,\ P^- \cup \{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\})$ and (ii) recursively solve the problem for all subsets of Q to which $\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2$ is added, which is done by adding the element to P^+, that is $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ Q\ ,\ P^+ \cup \{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}\ ,\ P^-)$. In other terms, the recursive calls of $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P^\circ\ ,\ P^+\ ,\ P^-)$ will solve the problem for all subsets of $P^+ \cup P^\circ$ that contain (at least) P^+, that is, these recursive calls will check equation (4.8) for all subsets P' of P such that $P^+ \subseteq P' \subseteq P^+ \cup P^\circ$. When P° is empty, and there no longer is an element to pick to make the recursive calls, then P^+ is the P' at issue and P^- is $P \setminus P'$. In that case formula (4.8) tells us that we have to check that $\mathtt{T}_1$ is smaller than the union of the domains of all the arrows in P^+ and that $\mathtt{T}_2$ is larger than the intersection of the codomains of the arrows in P^-. This leads to the last observation we need before giving the actual definition of Φ: we do not need to keep the whole P^+ and P^- sets; we just need the union of the domains for P^+ and the intersection of the codomains for P^-. This yields the following recursive definition of the subtyping relation for the arrow case $\bigwedge_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}(\mathtt{S}_1\mathtt{-->}\mathtt{S}_2)\ \mathtt{<:}\ \mathtt{T}_1\mathtt{-->}\mathtt{T}_2$:

$$(\mathtt{T}_1 \mathtt{<:} \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P} \mathtt{S}_1) \text{ and } \Phi(\mathtt{T}_1, \mathtt{T}_2, P, \mathtt{Empty}, \mathtt{Any}) \tag{4.9}$$

where

$$\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = (\mathtt{T}_1\mathtt{<:D}) \text{ or } (\mathtt{C<:}\mathtt{T}_2)$$

$$\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\cup\{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = \\ \Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\ ,\ \mathtt{D}\ ,\ \mathtt{C\&}\mathtt{S}_2^\circ) \text{ and } \Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\ ,\ \mathtt{D}\,|\,\mathtt{S}_1^\circ\ ,\ \mathtt{C})$$

The definition Φ above is not very interesting: all it does is just to explore the whole space of the subsets of P. But it becomes interesting when we try get rid of the two extra parameters C and D. Notice that C and D are two accumulators respectively for the intersection of the Codomains and the union of the Domains of the arrows singled out in the 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 Φ' as follows:

$$\Phi'(\mathtt{Empty}\ ,\ \mathtt{T}_2\ ,\ \varnothing) = \mathtt{true}$$

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{Empty}\ ,\ \varnothing) = \mathtt{true}$$

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing) = \mathtt{false} \qquad \text{otherwise}$$

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\cup\{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}) = \Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\mathtt{\&}\mathtt{S}_2^\circ\ ,\ P) \text{ and } \Phi'(\mathtt{T}_1\setminus\mathtt{S}_1^\circ\ ,\ \mathtt{T}_2\ ,\ P)$$

then it is not difficult to see that for all $\mathtt{T}_1$, $\mathtt{T}_2$, P, C, and D, $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = \Phi'(\mathtt{T}_1\setminus\mathtt{D}\ ,\ \mathtt{C}\setminus\mathtt{T}_2, P)$ and that, therefore, we can use

$$(\mathtt{T}_1\ \mathtt{<:}\ \bigvee_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_1) \text{ and } \Phi'(\mathtt{T}_1,\mathtt{not}(\mathtt{T}_2),P) \tag{4.10}$$

instead of (4.9).

I invite the reader to verify that both Φ and Φ' compute the Boolean function described in (4.8) and leave the definition of a similar function for (4.7) as exercise [EX6]. In particular, I suggest to try to use the properties $\bigwedge_{(\mathtt{S}_1,\mathtt{S}_2)\in P}(\mathtt{S}_1\ ,\ \mathtt{S}_2) = (\bigwedge_{(\mathtt{S}_1,\mathtt{S}_2)\in P}\mathtt{S}_1\ ,\ \bigwedge_{(\mathtt{S}_1,\mathtt{S}_2)\in P}\mathtt{S}_2)$ and $(\mathtt{S}_1\ ,\ \mathtt{S}_2)\setminus(\mathtt{T}_1\ ,\ \mathtt{T}_2) = (\mathtt{S}_1\setminus\mathtt{T}_1\ ,\ \mathtt{S}_2)\,|\,(\mathtt{S}_1\ ,\ \mathtt{S}_2\setminus\mathtt{T}_2)$ to define an algorithm potentially more efficient. I also leave as an exercise [EX7] the modification of Φ and Φ' to implement the optimization corresponding to the strict containment $P'\subsetneq P$ when checking (4.8).

$\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P)$ is pretty efficient when the subtyping relation holds, since there is no choice: all strict subsets of P must be tested. However, it does not perform any test before emptying the parameter P by a sequence of repeated recursive calls. As a possible optimization, it may be interesting to perform some checks earlier, when P is yet not empty, thus possibly avoiding some recursive calls when the subtyping relation does not hold. A way to do so for a generic call $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P^\circ\ ,\ P^+\ ,\ P^-)$ is first to check formula (4.8) for $P' = P^\circ\cup P^+$ and, only if it succeeds, then perform the two recursive calls. This corresponds to modifying the definition of Φ as follows:

$$\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = (\mathtt{T}_1\mathtt{<:D}) \text{ or } (\mathtt{C<:}\mathtt{T}_2)$$

$$\begin{aligned}\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\cup\{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = \\ &((\mathtt{T}_1\mathtt{<:}\mathtt{S}_1^\circ\,|\,\mathtt{D}) \text{ or } (\textstyle\bigwedge_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_2\mathtt{\&C}\ \mathtt{<:}\ \mathtt{T}_2)) \qquad (*)\\ &\text{and } \Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\ ,\ \mathtt{D}\ ,\ \mathtt{C\&}\mathtt{S}_2^\circ)\\ &\text{and } \Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P\ ,\ \mathtt{D}\,|\,\mathtt{S}_1^\circ\ ,\ \mathtt{C})\end{aligned}$$

which yields the following modification for Φ':

$$\Phi'(\mathtt{Empty}\ ,\ \mathtt{T}_2\ ,\ \varnothing) = \mathtt{true}$$

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{Empty}\ ,\ \varnothing) = \mathtt{true}$$

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing) = \mathtt{false} \qquad \text{otherwise}$$

$$\begin{array}{l}\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P \cup \{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}) = \\ \quad ((\mathtt{T}_1\mathtt{<:}\mathtt{S}_1^\circ)\ \mathsf{or}\ (\textstyle\bigwedge_{\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\in P}\mathtt{S}_2\mathtt{<:}\ \mathtt{not}(\mathtt{T}_2))) \qquad (*)\\ \quad \mathsf{and}\ \Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\mathtt{\&}\mathtt{S}_2^\circ\ ,\ P)\\ \quad \mathsf{and}\ \Phi'(\mathtt{T}_1 \setminus \mathtt{S}_1^\circ\ ,\ \mathtt{T}_2\ ,\ P)\end{array}$$

The extra checks performed in the $(*)$-marked lines are interesting in the case they fail, since this allows us to stop the computation of Φ and Φ' even when the parameter P is not empty, thus potentially saving several recursive calls. The price to pay is that when the subtyping relation holds, the functions will perform n useless checks (since they will be done twice) where n is the cardinality of P. But in the case of Φ' these useless checks have a very low cost, since they amount to checking whether $\mathtt{T}_1$ of $\mathtt{T}_2$ are empty types. As we show in the next section, checking the emptiness of a type just amounts at checking the content of four fields in a record; this must be contrasted with the fact that every recursive call requires to perform either an intersection or a difference of types, which are much costlier operations.

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:

$$\bigvee_{i\in I}\Big(\bigwedge_{p\in P_i} a_p\ \mathtt{\&}\ \bigwedge_{n\in N_i}\mathtt{not}(a_n)\Big) \tag{4.11}$$

where a_i's are atoms. A naive representation of (4.11) (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 $\mathbb{0}$ or $\mathbb{1}$. In a BDD every path starting from the root and terminating by $\mathbb{1}$ 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 $(a_1\mathtt{\&}a_2)\,\mathtt{|}\,(a_1\mathtt{\&not(}a_2\mathtt{)\&}a_3)\,\mathtt{|}\,(\mathtt{not(}a_1\mathtt{)\&not(}a_3\mathtt{)})$ 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, BDD are defined by the following grammar:

$$B \quad ::= \quad \mathbb{0} \mid \mathbb{1} \mid a\mathtt{?}B\mathtt{:}B$$

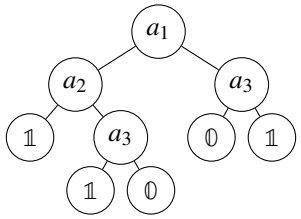

Figure 1: BDD for $(a_1 \wedge a_2) \vee (a_1 \wedge \neg a_2 \wedge a_3) \vee (\neg a_1 \wedge \neg a_3)$

and have the following interpretation:

$$\begin{array}{rcl} [\![\mathbb{0}]\!] & = & \texttt{Empty} \\ [\![\mathbb{1}]\!] & = & \texttt{Any} \\ [\![a?B_1:B_2]\!] & = & (a\texttt{\&}[\![B_1]\!])\texttt{|}(\texttt{not(}a\texttt{)\&}[\![B_2]\!]) \end{array}$$

The interpretation above maps a BDD into the disjunctive normal form it represents (of course, after having simplified the intersections with `Any` and `Empty`), that is, into the union of the intersections that correspond to paths ending by a $\mathbb{1}$. 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, B_1, and B_2 denote generic BDDs, $B_1 = a_1?C_1:D_1$ and $B_2 = a_2?C_2:D_2$. Unions, intersections, and differences of BDDs are defined as follows:

$$\begin{array}{ll} \mathbb{1} \vee B = B \vee \mathbb{1} = \mathbb{1} & \mathbb{1} \wedge B = B \wedge \mathbb{1} = B \\ \mathbb{0} \vee B = B \vee \mathbb{0} = B & \mathbb{0} \wedge B = B \wedge \mathbb{0} = \mathbb{0} \\ B \setminus \mathbb{1} = \mathbb{0} \setminus B = \mathbb{0} & B \setminus \mathbb{0} = B \end{array}$$

$$\mathbb{1} \setminus a?B_1:B_2 \;=\; a?\mathbb{1} \setminus B_1 : \mathbb{1} \setminus B_2$$

$$B_1 \vee B_2 \;=\; \begin{cases} a_1?C_1 \vee C_2 : D_1 \vee D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \vee B_2 : D_1 \vee B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \vee C_2 : B_1 \vee D_2 & \text{for } a_1 > a_2 \end{cases}$$

$$B_1 \wedge B_2 \;=\; \begin{cases} a_1?C_1 \wedge C_2 : D_1 \wedge D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \wedge B_2 : D_1 \wedge B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \wedge C_2 : B_1 \wedge D_2 & \text{for } a_1 > a_2 \end{cases}$$

$$B_1 \setminus B_2 \;=\; \begin{cases} a_1?C_1 \setminus C_2 : D_1 \setminus D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \setminus B_2 : D_1 \setminus B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \setminus C_2 : B_1 \setminus D_2 & \text{for } a_1 > a_2 \end{cases}$$

Notice that $\mathbb{1} \setminus B$ computes the negation of B and is obtained by exchanging the $\mathbb{0}$ leaves of B into $\mathbb{1}$ 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 $a?B_1:B_0:B_2$, where the middle child represents a lazy union:

$$[\![a?B_1:B_0:B_2]\!] \quad = \quad (a\&[\![B_1]\!]) \,|\, [\![B_0]\!] \,|\, (\mathtt{not}(a)\&[\![B_2]\!])$$

Let $B_1 = a_1?C_1:U_1:D_1$ and $B_2 = a_2?C_2:U_2:D_2$. When two atoms are merged, unions are lazily recorded in the middle child:

$$B_1 \vee B_2 \quad = \quad \begin{cases} a_1?C_1 \vee C_2 : U_1 \vee U_2 : D_1 \vee D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 : U_1 \vee B_2 : D_1 & \text{for } a_1 < a_2 \\ a_2?C_2 : B_1 \vee U_2 : D_2 & \text{for } a_1 > a_2 \end{cases}$$

The intersection $B_1 \wedge B_2$ materializes the lazy unions when the top-level atoms are the same and trees are merged, and is defined as:

$$\begin{cases} a_1?(C_1 \vee U_1) \wedge (C_2 \vee U_2) : \mathbb{0} : (D_1 \vee U_1) \wedge (D_2 \vee U_2) & \text{for } a_1 = a_2 \\ a_1?C_1 \wedge B_2 : U_1 \wedge B_2 : D_1 \wedge B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \wedge C_2 : B_1 \wedge U_2 : B_1 \wedge D_2 & \text{for } a_1 > a_2 \end{cases}$$

while the difference $B_1 \setminus B_2$ materializes unions when top-level atoms are distinct, and is defined as:

$$\begin{cases} a_1?C_1 \setminus C_2 : U_1 \setminus U_2 : D_1 \setminus D_2 & \text{for } a_1 = a_2 \\ a_1?(C_1 \vee U_1) \setminus B_2 : \mathbb{0} : (D_1 \vee U_1) \setminus B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \setminus (C_2 \vee U_2) : \mathbb{0} : B_1 \setminus (D_2 \vee U_2) & \text{for } a_1 > a_2 \end{cases}$$

After having performed these operations it is possible to perform two simplifications, namely, replace $(a?B_1:\mathbb{1}:B_2)$ by $\mathbb{1}$, and replace $(a?B:C:B)$ by $B \vee C$.

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 (4.11) can be seen as the union of four different unions, one for each kind of atom:

$$\bigvee_{i\in I_{\mathtt{tags}}} \Big(\bigwedge_{p\in P_i} t_p \;\&\; \bigwedge_{n\in N_i} \mathtt{not}(t_n)\Big) \quad \vee \tag{4.12}$$

$$\bigvee_{i\in I_{\mathtt{ints}}} \Big(\bigwedge_{p\in P_i} [h_p..k_p] \;\&\; \bigwedge_{n\in N_i} \mathtt{not}([h_n..k_n])\Big) \quad \vee \tag{4.13}$$

$$\bigvee_{i\in I_{\mathtt{prod}}} \Big(\bigwedge_{p\in P_i} (\mathtt{S}_p\,,\mathtt{T}_p) \;\&\; \bigwedge_{n\in N_i} \mathtt{not}((\mathtt{S}_n\,,\mathtt{T}_n))\Big) \quad \vee \tag{4.14}$$

$$\bigvee_{i\in I_{\mathtt{arrw}}} \Big(\bigwedge_{p\in P_i} \mathtt{S}_p\mathtt{-->}\mathtt{T}_p \;\&\; \bigwedge_{n\in N_i} \mathtt{not}(\mathtt{S}_n\mathtt{-->}\mathtt{T}_n)\Big) \tag{4.15}$$

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 of form of the corresponding atom, namely:

```
{ tags: dnf_tags ;        // stores a union as in (4.12)
  ints: dnf_ints ;        // stores a union as in (4.13)
  prod: dnf_prod ;        // stores a union as in (4.14)
  arrw: dnf_arrw          // stores a union as in (4.15)
}
```

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

$$\bigvee_{k\in\texttt{Kinds}} (\texttt{T}.k \wedge \texttt{Any}_k)$$

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`∧`T`, `S`∨`T`, and `S` \ `T` can be respectively implemented as:

```
{ tags = S.tags∧T.tags;   { tags = S.tags∨T.tags;   { tags = S.tags\T.tags;
  ints = S.ints∧T.ints;     ints = S.ints∨T.ints;     ints = S.ints\T.ints;
  prod = S.prod∧T.prod;     prod = S.prod∨T.prod;     prod = S.prod\T.prod;
  arrw = S.arrw∧T.arrw      arrw = S.arrw∨T.arrw      arrw = S.arrw\T.arrw
}                         }                         }
```

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 `prod` and `arrw`, corresponding to product types and arrow types the use of BDDs with lazy unions to represent the unions in (4.14) and (4.15) is the obvious choice. For base types, `tags` and `ints`, we can use instead a specific representation. In particular, it is not difficult to prove that any union of the form (4.12) can be equivalently expressed either as a union of pairwise distinct atoms $(a_1 \vee \ldots \vee a_n)$ or its negation $\neg(a_1 \vee \ldots \vee a_n)$ (I leave this proof as an exercise for the reader [EX8]). 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 `prod` or `arrw` fields. This means that the contractivity 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<:T` holds do:

Step 1:: Compute `S\T`;
Step 2:: Check that all the fields of `S\T` are the empty type.

Checking the emptiness of the base type fields `tags` and `ints` is immediate: they must be an empty set of atoms with a positive flag. For the fields `prod` and `arrw`, if the BDD that they contain is not $\mathbb{0}$ or $\mathbb{1}$, 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 `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 [EX9].

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 $e\texttt{[0]}$ and $e\texttt{[1]}$ denoting respectively the first and second projection of e.

The algorithm for typing pairs is straightforward: if e_1 is of type $\texttt{T}_1$ and e_2 of type $\texttt{T}_2$, then (e_1,e_2) is of type $(\texttt{T}_1\,,\texttt{T}_2)$.

The algorithm for typing projections, instead, is more complicated, since projections can be applied to any expression of type $\texttt{Any}_{\texttt{prod}}$, that is, any expression whose type has a normal form as in (4.14). So imagine that we want to type the expressions $e\texttt{[0]}$ or $e\texttt{[1]}$ where e is of some type $\texttt{T}$. The first thing to do is to verify that e will return a pair, that is, that $\texttt{T<:Any}_{\texttt{prod}}$ holds. If it is so, then $\texttt{T}$ is equivalent to the following normal form:

$$\bigvee_{i\in I}\Big(\bigwedge_{p\in P_i}(\texttt{S}_p\,,T_p)\;\&\;\bigwedge_{n\in N_i}\texttt{not}((\texttt{S}_n\,,\texttt{T}_n))\Big) \tag{4.16}$$

in which case the type of the expression $e\texttt{[0]}$ can be approximated by:

$$\bigvee_{i\in I}\bigvee_{N'\subseteq N_i}\Big(\bigwedge_{p\in P_i}\texttt{S}_p\;\&\;\bigwedge_{n\in N'}\texttt{not}(\texttt{S}_n)\Big) \tag{4.17}$$

and, likewise, the type of the expression $e\texttt{[1]}$ can be approximated by:

$$\bigvee_{i\in I}\bigvee_{N'\subseteq N_i}\Big(\bigwedge_{p\in P_i}\texttt{T}_p\;\&\;\bigwedge_{n\in N'}\texttt{not}(\texttt{T}_n)\Big) \tag{4.18}$$

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

Let me explain how to pass from (4.16) to (4.17) and (4.18). The idea is simple and consists to transform the union in (4.16) 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: $\bigwedge_{p\in P}(\texttt{S}_p\,,T_p)$ is equivalent to $(\bigwedge_{p\in P}\texttt{S}_p\,,\bigwedge_{p\in P}T_p)$. Second, the intersection of a product with a negated product can be distributed inside the product: $(\texttt{S}_1\,,\texttt{S}_2)\texttt{\&not}((\texttt{T}_1\,,\texttt{T}_2))$ is equivalent to $(\texttt{S}_1\texttt{\&not}(\texttt{T}_1)\,,\texttt{S}_2)\,|\,(\texttt{S}_1\,,\texttt{S}_2\texttt{\&not}(\texttt{T}_2))$. For multiple intersections such as

$$\bigwedge_{p\in P}(\texttt{S}_p\,,T_p)\;\&\;\bigwedge_{n\in N}\texttt{not}((\texttt{S}_n\,,\texttt{T}_n))$$

the two transformations above yield the following equivalent type

$$\bigvee_{N'\subseteq N}\Big(\bigwedge_{p\in P}\texttt{S}_p\;\&\;\bigwedge_{n\in N'}\texttt{not}(\texttt{S}_n)\,,\;\bigwedge_{p\in P}\texttt{T}_p\;\&\;\bigwedge_{n\in N\setminus N'}\texttt{not}(\texttt{T}_n)\Big) \tag{4.19}$$

it is then easy from this type to deduce the types (4.17) and (4.18) of the projections, simply by observing that the projection of a union of products is the union of the projections of each product.

Finally, I said that the types in (4.17) and (4.18) are approximations. Indeed in some cases a better type can be defined for the projections. Consider again the products forming the union in (4.19). If for some N' either $\bigwedge_{p\in P}\mathtt{S}_p\ \&\bigwedge_{n\in N'}\mathtt{not(S}_n\mathtt{)}$ is empty or $\bigwedge_{p\in P}\mathtt{T}_p\ \&\bigwedge_{n\in N\setminus N'}\mathtt{not(T}_n\mathtt{)}$ is empty, then so is their product. Therefore this N' can be eliminated from the union in (4.19) and, thus, also from the unions in (4.17) and (4.18). So the way to proceed to type projections is to transform the type of e into the form of (4.19), eliminate the empty products and take the union of the remaining projects. Formally, if T is equivalent to the type in (4.16) then the type of e`[0]` is

$$\bigvee_{i\in I}\ \bigvee_{N'\in\mathsf{N}_1}\ (\bigwedge_{p\in P_i}\mathtt{S}_p\ \&\bigwedge_{n\in N'}\mathtt{not(S}_n\mathtt{)}) \tag{4.20}$$

where

$$\mathsf{N}_1=\{N'\mid N'\subseteq N \text{ and } \bigwedge_{p\in P}\mathtt{T}_p\not<:\bigvee_{n\in N\setminus N'}\mathtt{T}_n\}$$

and, likewise the type of e`[1]` is:

$$\bigvee_{i\in I}\ \bigvee_{N'\in\mathsf{N}_2}\ (\bigwedge_{p\in P_i}\mathtt{T}_p\ \&\bigwedge_{n\in N'}\mathtt{not(T}_n\mathtt{)}) \tag{4.21}$$

where

$$\mathsf{N}_2=\{N'\mid N'\subseteq N \text{ and } \bigwedge_{p\in P}\mathtt{S}_p\not<:\bigvee_{n\in N'}\mathtt{S}_n\}$$

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

`sub (`$\mathtt{T}_1$ `$\mathtt{\$x}_1$`,..., `$\mathtt{T}_n$ `$\mathtt{\$x}_n$`) { `$e$ `}`

or of the form

`sub (`$\mathtt{T}_1$ `$\mathtt{\$x}_1$`,..., `$\mathtt{T}_n$ `$\mathtt{\$x}_n$`) returns S { `$e$ `}`

it has type `(`$\mathtt{T}_1$`,..., `$\mathtt{T}_n$`)-->S` if under the hypothesis that $\mathtt{\$x}_1$ has type $\mathtt{T}_1$ and that $\mathtt{\$x}_n$ has type $\mathtt{T}_n$ 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

`sub (`$\mathtt{T}_1$ `$\mathtt{\$x}_1$`,...,`$\mathtt{T}_n$ `$\mathtt{\$x}_n$ `;;` $\mathtt{T}_{n+1}$ `$\mathtt{\$x}_{n+1}$`,...,`$\mathtt{T}_{n+k}$`) { `$e$ `}`

has type `(`$\mathtt{T}_1$`,..., `$\mathtt{T}_n$`)-->(`$\mathtt{T}_{n+1}$`,...,`$\mathtt{T}_{n+k}$`)-->S` if under the hypotheses that $\mathtt{\$x}_i$ has type $\mathtt{T}_i$ (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 $\mathtt{S}_i\mathtt{-->T}_i$ 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 $\bigwedge_{i=1}^{n}\mathtt{S}_i\mathtt{-->T}_i$.

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 $\mathtt{e}_1$ and $\mathtt{e}_2$ which are well typed respectively with type `T` and `S`. We want to check (a) whether the application $\mathtt{e}_1\mathtt{e}_2$ 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 $\mathtt{e}_1$ is a function (i.e., it returns neither a constant nor a pair). This is done by checking that the type `T` of $\mathtt{e}_1$ is a subtype of $\mathtt{Any}_{\mathtt{arrw}}$, that is `T<:Empty-->Any`. If this holds then it checks that the type `S` of the argument is a subtype of the domain $\mathsf{dom}(\mathtt{T})$ of the function, that is $\mathtt{S}\mathtt{<:}\mathsf{dom}(\mathtt{T})$, where the domain is defined as follows:

$$\mathsf{dom}\Big(\bigvee_{i\in I}\Big(\bigwedge_{p\in P_i}\mathtt{S}_p\mathtt{-->}\mathtt{T}_p\ \mathtt{\&}\bigwedge_{n\in N_i}\mathtt{not(}\mathtt{S}_n\mathtt{-->}\mathtt{T}_n\mathtt{)}\Big)\Big)\stackrel{\text{def}}{=}\bigwedge_{i\in I}\bigvee_{p\in P_i}\mathtt{S}_p$$

The definition of domain is given for a normal form as in (4.15) since this is the disjunctive normal form of any type smaller than $\mathtt{Any}_{\mathtt{arrw}}$. 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 `(Bool-->S)&(Int-->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..4]-->S)|([2..6]-->T)` can be applied only to arguments in `[2..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 $\mathtt{T}\mathtt{<:}\mathtt{Any}_{\mathtt{arrw}}$ —i.e., `T` is a function type— then

$$\mathtt{T}=\bigvee_{i\in I}\Big(\bigwedge_{p\in P_i}\mathtt{S}_p\mathtt{-->}\mathtt{T}_p\ \mathtt{\&}\bigwedge_{n\in N_i}\mathtt{not(}\mathtt{S}_n\mathtt{-->}\mathtt{T}_n\mathtt{)}\Big)\tag{4.22}$$

for some I, P_i and N_i. Then the type of the application of a function of the type `T` defined in (4.22) to an argument of type `S` is

$$\bigvee_{i\in I}\left(\bigvee_{Q\subsetneq P_i\text{ s.t. }\mathtt{S}\not\leq\bigvee_{q\in Q}\mathtt{S}_q}\left(\bigwedge_{p\in P_i\setminus Q}\mathtt{T}_p\right)\right)\tag{4.23}$$

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 (4.22) —and, thus, (4.23)— 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 (4.23), then we can for simplicity consider that the type `T` of the function is

$$\bigwedge_{p\in P}\mathtt{S}_p\mathtt{-->}\mathtt{T}_p$$

and that the type `S` of the argument is a subtype of the domain of `T`, that is $\mathtt{S}\mathtt{<:}\bigvee_{p\in P}\mathtt{S}_p$.

If the argument can return a value in the domain $\mathtt{S}_i$ (for some $i\in P$) of some arrow (i.e., if $\mathtt{S}\mathtt{\&}\mathtt{S}_i$ is not empty), and this actually happens, then the result returned by the application (if any) will be in $\mathtt{T}_i$. If the argument can return a value in domain of *two* arrows say $\mathtt{S}_i$ and $\mathtt{S}_j$ for $i,j\in P$ (i.e., $\mathtt{S}\mathtt{\&}\mathtt{S}_i\mathtt{\&}\mathtt{S}_j$ is not empty), then in the result returned by application (if any) in that case will be a value in $\mathtt{T}_i\mathtt{\&}\mathtt{T}_j$. Of course we want to deduce the most precise type for the result type. So in this case we will deduce as a type for our result $\mathtt{T}_i\mathtt{\&}\mathtt{T}_j$ rather than just $\mathtt{T}_i$ or $\mathtt{T}_j$. Also, it may be the case that $\mathtt{T}_i\mathtt{\&}\mathtt{T}_j$ does not

cover all possible cases for the result: not only S may intersect intersections of the domains smaller than $\mathtt{S}_i\&\mathtt{S}_j$ (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 $\mathtt{S}_i\&\mathtt{S}_j$, that is, a result in $\mathtt{S} \setminus (\mathtt{S}_i \mid \mathtt{S}_j)$, 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 $\mathtt{S}_i$ for $i \in P$ and keeps only those whose intersection with S is not empty; then (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 (4.23) does. First it considers all possible combinations of the domains $\mathtt{S}_p$ 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 $S \not\leq \bigvee_{q\in Q} \mathtt{S}_q$. Now take any *maximal* subset Q that satisfies $S \not\leq \bigvee_{q\in Q} \mathtt{S}_q$. Being maximal means that if we add to Q any index p in $P \setminus Q$ then by adding the corresponding $\mathtt{S}_p$ we cover the whole S. This means that all elements of S that are missing in $\bigvee_{q\in Q} \mathtt{S}_q$ —that is all elements in $S \setminus \bigvee_{q\in Q} \mathtt{S}_q$— are in *all* the $\mathtt{S}_p$ for $p \in P \setminus Q$. Therefore the intersection $\mathtt{S} \wedge \bigwedge_{p\in P\setminus Q} \mathtt{S}_p$ is not empty, since it contains $S \setminus \bigvee_{q\in Q} \mathtt{S}_q$. 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 $\bigwedge_{p\in P\setminus Q} \mathtt{T}_p$, and we union all these intersections (notice that the union in (4.23) also adds some smaller intersections which corresponds 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 (4.23), 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 argument will be a subtype of the domain of the actual type. However we do not know which result type to pick among the possible ones: since it may be any of them, then we take all of them, that is their union. This explains the outermost union in (4.23). For instance, if we have a function whose static type is a union of two arrows:

```
( (Even,Int) --> 0 ) |( (Int,Odd) --> 1 )
```
(4.24)

then it may dynamically be a function of either of the two types. Thus the static typing must cover all possible cases: such a function must be applied to arguments that are compatible with both arrows—i.e., arguments that are in the intersection of the two domains: `(Even,Odd)`—; since the result of this application can be in either of the result types, then the application of such a function to an argument in the intersection of the domains will be typed by their union, that is `0|1`.

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 plethoric 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:

```
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

```
  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 adding 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 some 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 to 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 `{` ℓ_1 `=>` e_1`,...,`ℓ_n `=>` e_n `}` which denotes the record that associates the label ℓ_i to (the value returned by) the expression e_i, for $i \in [1..n]$. For instance, if we write

```
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 $e\backslash\ell$ 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:

$$e \quad ::= \quad \{\, \ell \texttt{=>} e \,\} \quad | \quad e\texttt{+}e \quad | \quad e\backslash\ell \quad | \quad e\texttt{<}\ell\texttt{>} \tag{4.25}$$

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 in this article, 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`. The former means that selecting the label ℓ will return a value of type `T`, the latter that the same operation will return either a value of type `T` or `undef`. This syntax will be used 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 $\{f_1,...,f_n\}$ for the former and $\{f_1,...,f_n$ `, ..}` for the latter (where the f_i'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 and all record values (`{..}` is the top type for record kind, that is, $\mathtt{Any}_{\mathtt{recd}}$).

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 the operations of record selection and concatenation 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 $\mathcal{L}$ to values. In what follows I use a slightly different interpretation and consider records as specific total maps on $\mathcal{L}$, 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 all the other labels it is the constant function that maps them into the special value $\bot$ (i.e., the `undef` value of Perl).

Formally, let Z denote some set, a function $r : \mathcal{L} \to Z$ is *quasi-constant* if there exists $z \in Z$ such that the set $\{\ell{\in}\mathcal{L} \mid r(\ell) \neq z\}$ is finite; in this case I denote this set by $\mathsf{dom}(r)$ and the element z by $\mathsf{def}(r)$ (i.e., the default value). I use $\mathcal{L} \to\!\!\!\!\circ\ Z$ to denote the set of quasi-constant functions from $\mathcal{L}$ to Z and the notation $\{\ell_1 = z_1, \ldots, \ell_n = z_n, \_ = z\}$ to denote the quasi-constant function $r : \mathcal{L} \to\!\!\!\!\circ\ Z$ defined by $r(\ell_i) = z_i$ for $i = 1..n$ and $r(\ell) = z$ for $\ell \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}$. Although this notation is not univocal (unless we require $z_i \neq z$), this is largely sufficient for the purposes of this section.

Let $\bot$ be a distinguished constant. I single out two particular sets of quasi-constant functions: the set $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Types} \cup \{\bot\}$, called the set of *quasi-constant typing functions* ranged over by r; and $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Values} \cup \{\bot\}$ the set of record values. The constant $\bot$ 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 $\bot$ occurs in $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Values} \cup \{\bot\}$ it denotes a value, while in $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Types} \cup \{\bot\}$ it denotes the singleton type that contains only the value $\bot$.

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 $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Types} \cup \{\bot\}$. More precisely, the open record type $\{\ell_1 : \mathtt{T}_1, \ldots, \ell_n : \mathtt{T}_n$ `, ..}` denotes the quasi-constant function $\{\ell_1 = \mathtt{T}_1, \ldots, \ell_n = \mathtt{T}_n, \_ = \mathtt{Any}\}$ while the closed record type $\{\ell_1 : \mathtt{T}_1, \ldots, \ell_n : \mathtt{T}_n\}$ denotes the quasi-constant function $\{\ell_1 = \mathtt{T}_1, \ldots, \ell_n = \mathtt{T}_n, \_ = \bot\}$. Similarly, the optional field notation $\{..., \ell\mathtt{?:T}, ...\}$ denotes the record typing in which ℓ is mapped either to $\bot$ or to the type `T`, that is, $\{..., \ell = \mathtt{T} \mid \bot, ...\}$. In conclusion, the open and closed record types are just syntax to denote specific quasi-constant functions in $\mathtt{string} \to\!\!\!\!\circ\ \mathbf{Types} \cup \{\bot\}$.

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 (4.3)–(4.6) the following case:

$$\bigwedge_{p\in P} \mathtt{R}_p \;\&\; \bigwedge_{n\in N} \mathtt{not}(\mathtt{R}_n) \tag{4.26}$$

where, I remind, every type $\mathtt{R}_i$ in this formula is a record type atom, that is, it is either of the form $\{\ell_1 : \mathtt{T}_1, \ldots, \ell_n : \mathtt{T}_n\}$ or of the form $\{\ell_1 : \mathtt{T}_1, \ldots, \ell_n : \mathtt{T}_n, \mathtt{..}\}$. Then ***Step 4*** simplifies the instances of this new case by using the containment property for quasi-constant functions (see Lemma 9.1 in [24]) specialized for the case in (4.26) (remember that the $\mathtt{R}$'s in the formula above denote special cases of quasi-constant functions.) In particular to decide the emptiness of the type in (4.26) we decompose the problem into checking that for every map $\iota : N \to L \cup \{\_\}$

$$\begin{array}{l} \left(\exists \ell \in L.\big(\bigwedge_{p\in P} \mathtt{R}_p(\ell) \mathtt{<:} \bigvee_{n\in N|\iota(n)=\ell} \mathtt{R}_n(\ell) \big) \right) \text{ or} \\ \left(\exists n_\circ \in N.\big((\iota(n_\circ) = \_) \text{ and } \big(\bigwedge_{p\in P} \mathsf{def}(R_p) \mathtt{<:} \mathsf{def}(\mathtt{R}_{n_\circ}) \big) \big) \right) \end{array} \tag{4.27}$$

where $L = \bigcup_{i\in P\cup N} \mathsf{dom}(\mathtt{R}_i)$.
The technique used to derive (4.10) can be adapted to this case as well (just by branching on $L \cup \{\_\}$ rather than a single element).

Type operators. Let $\mathtt{T}$ be a type and $\mathtt{R}_1, \mathtt{R}_2$ two record type atoms. The *merge* of $\mathtt{R}_1$, and $\mathtt{R}_2$ with respect to $\mathtt{T}$, noted $\oplus_{\mathtt{T}}$ and used infix, is the quasi-constant typing function defined as follows:

$$(\mathtt{R}_1 \oplus_{\mathtt{T}} \mathtt{R}_2)(\ell) \stackrel{\text{def}}{=} \begin{cases} \mathtt{R}_1(\ell) & \text{if } \mathtt{R}_1(\ell)\&\mathtt{T} \leq \mathtt{Empty} \\ (\mathtt{R}_1(\ell)\backslash\mathtt{T}) \,|\, \mathtt{R}_2(\ell) & \text{otherwise} \end{cases}$$

Record types are akin to product types (a classical interpretation of records is that of products indexed over a finite domain, *eg* [11]). 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 $\mathtt{T} \leq \{\mathtt{..}\}$, then $\mathtt{T}$ is equivalent to a finite union of the form $\bigvee_{i\in I} \mathtt{R}_i$ (see Lemma 11 in [5] for details). So the definition of *merge* can be easily extended to all record types as follows

$$(\bigvee_{i\in I} \mathtt{R}_i) \oplus_{\mathtt{T}} (\bigvee_{j\in J} \mathtt{R}'_j) \stackrel{\text{def}}{=} \bigvee_{i\in I, j\in J} (\mathtt{R}_i \oplus_{\mathtt{T}} \mathtt{R}'_j)$$

The merge operator can be used to define concatenation and field deletion on record types. Let $\mathtt{T}_1$, $\mathtt{T}_2$, and $\mathtt{T}$ range over record types (i.e., types smaller than $\{\mathtt{..}\}$). Define:

$$\mathtt{T}_1 + \mathtt{T}_2 \stackrel{\text{def}}{=} \mathtt{T}_2 \oplus_{\perp} \mathtt{T}_1 \tag{4.28}$$

$$\mathtt{T} \setminus \ell \stackrel{\text{def}}{=} \{\ell = \perp, \_ = c_\circ\} \oplus_{c_\circ} \mathtt{T} \tag{4.29}$$

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

Notice in particular that the result of the concatenation of two record type atoms $\mathtt{R}_1 + \mathtt{R}_2$ may result for each field ℓ in three different outcomes:

(1) if $\mathtt{R}_2(\ell)$ does not contain $\bot$ (i.e., the field ℓ is surely defined), then we take the corresponding field of $\mathtt{R}_2$: $(\mathtt{R}_1+\mathtt{R}_2)(\ell)=\mathtt{R}_2(\ell)$
(2) if $\mathtt{R}_2(\ell)$ is undefined (i.e., $\mathtt{R}_2(\ell)=\bot$), then we take the corresponding field of $\mathtt{R}_1$: $(\mathtt{R}_1+\mathtt{R}_2)(\ell)=\mathtt{R}_1(\ell)$
(3) if $\mathtt{R}_2(\ell)$ *may* be undefined (i.e., $\mathtt{R}_2(\ell)=\mathtt{T}\,|\,\bot$ for some type $\mathtt{T}$), then we take the union of the two corresponding fields since it can results either in $\mathtt{R}_1(\ell)$ or $\mathtt{R}_2(\ell)$ according to whether the record typed by $\mathtt{R}_2$ is undefined in ℓ or not: $(\mathtt{R}_1+\mathtt{R}_2)(\ell)=\mathtt{R}_1(\ell)\,|\,(\mathtt{R}_2(\ell)\setminus\bot)$.

This explains some weird behavior of record concatenation such as $\{a:\mathtt{Int},b:\mathtt{Int}\}+\{a?:\mathtt{Bool}\}=\{a:\mathtt{Int}|\mathtt{Bool},b:\mathtt{Int}\}$ since "a" *may* be undefined in the right hand-side record while "b" *is* undefined in it, and $\{a:\mathtt{Int}\}+\{..\}=\{..\}$, since "$a$" in the right hand-side record is defined (with $a\mapsto\mathtt{Any}$) 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 (4.25). Thanks to the definitions (4.28) and (4.29) this is really straightforward:

***record:*:** if e has type $\mathtt{T}$, then $\{\,\ell\,\mathtt{=>}\,e\,\}$ has type $\{\ell:\mathtt{T}\}$;
***concatenation:*:** if e_1 has type $\mathtt{T}_1$, e_2 has type $\mathtt{T}_2$, and both $\mathtt{T}_1$ and $\mathtt{T}_2$ are subtypes of $\{..\}$, then e_1+e_2 has type $\mathtt{T}_1+\mathtt{T}_2$;
***deletion:*:** if e has type $\mathtt{T}$ and $\mathtt{T}$ is a subtype of $\{..\}$, then $e\backslash\ell$ has type $\mathtt{T}\setminus\ell$;
***selection:*:** if e has type $\mathtt{T}$ and $\mathtt{T}$ is a subtype of $\{..\}$, then $\mathtt{T}$ is equivalent to a union type of the form $\bigvee_{i\in I}\mathtt{R}_i$ and the type of e<ℓ> is $\bigvee_{i\in I}\mathtt{R}_i(\ell)$.

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 being just constants. For instance, it is possible to define:

```
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 [5]) 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>}`, and `$x{$x{"bar"}}` all return the integer `3` (as a matter of fact, in Perl 6 e<s> is syntactic sugar for e{"s"}). 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 [24] 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 naive 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 `S` where an expression of type `T`, not compatible with `S`, was expected. In order to produce a useful error message the system can compute the type `S\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 STUFF 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 Vouillon [30, 28, 27, 29]. At this point of the presentation I can confess that I have been slightly cheating, since in Section 2.1 I presented the fact that a type is a set of values as an unshakable truth while, in reality, such an interpretation holds, in the semantic subtyping framework, only for strict languages. As a matter of fact, the type system I presented in this work is unsound for non-strict languages, insofar as the application of a well-typed function to a diverging expression of type `Empty` is always well typed (I leave as exercise to the reader [EX10] to show why this is unsound in a lazy language). In my defense, I would like to say that this is something I realized just recently and that can be fixed with minimal effort, as shown in [2].

The idea of using using multiple definitions of methods to implement covariant specialization of binary methods was first proposed in my PhD thesis [12, 14] and the covariance and contravariance paper I wrote 20 years ago [13]. This technique was dubbed *encapsulated multi-methods* in [8] and implemented in different flavors for O_2 [6] and Java [7]. It was based on the type theory Giorgio

Ghelli, Giuseppe Longo and I developed in [16, 17] 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 [23], Fortress [1], Cecil [22], and Dubious [33].

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 Benzaken, and myself in [25, 26]. 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 [15] while the most comprehensive description of the work is by far Alain Frisch's PhD thesis [24], 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 [13]. 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 [13], 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 [13] a type such as (3.1) 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 [13] 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 ℂDuce, whose distribution is free and open-source [21] 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 (2.1) 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):

```
# 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 [26]. 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 [26]). The two decompositions that transform the formula (4.5) into (4.7) and the formula (4.6) into (4.8) are the generalizations to type connectives of the classic subtyping rules for products and arrows, respectively, as they can be found in subtyping system with syntax-oriented definitions. This point can better be grasped by considering the particular cases of the intersections in (4.5) and (4.6) when both P and N contain exactly one type. Then checking the emptiness of (4.5) and (4.6) corresponds to checking the following two subtyping relations

$$\texttt{(}S_1\texttt{ , }S_2\texttt{)} \quad \texttt{<:} \quad \texttt{(}T_1\texttt{ , }T_2\texttt{)}$$
$$S_1\texttt{-->}S_2 \quad \texttt{<:} \quad T_1\texttt{-->}T_2$$

and I leave as exercises to the reader [EX11,EX12] to check that in these cases the two decomposition formulas (4.7) and (4.8) of *Step 4* become, respectively:

$$(S_1\texttt{<:Empty}) \text{ or } (S_1\texttt{<:Empty}) \text{ or } (S_1\texttt{<:}T_1 \text{ and } S_2\texttt{<:}T_2)$$
$$(T_1\texttt{<:Empty}) \text{ or } (T_1\texttt{<:}S_1 \text{ and } S_2\texttt{<:}T_2)$$

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

For defining the functions Φ and Φ' I drew my inspiration from the algorithms defined in Chapter 7 of Alain Frisch's PhD thesis [24] and those used in the compiler of $\mathbb{C}$Duce.

For an alternative presentation of the material of Section 4, I warmly recommend the on-line tutorial *Down and Dirty with Semantic Set-theoretic Types* [32] in which Andrew M. Kent reexplains the implementation details of Section 4 in his own style.

The theory of quasi-constant functions and its application to record types is described in Chapter 9 of of Alain Frisch's PhD thesis [24], 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 $\mathbb{C}$Duce and described in details in Chapter 11 of Alain Frisch's PhD thesis [24]. The only difference is that the structures used in the compiler of $\mathbb{C}$Duce 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 (4.1)) 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 [19, 18], 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 [20].

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 developed 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 and I developed twenty years ago for the covariance vs. contravariance problem. In that system you were not allowed to have a type $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$ with $\mathtt{S}_2\mathtt{<:}\mathtt{S}_1$ and $\mathtt{T}_2\not\mathtt{<:}\mathtt{T}_1$: such a type was considered "ill-formed" and thus forbidden. With (twenty years of) hindsight that was a mistake. What the theory of semantic subtyping shows is that a type as the above is and must be admissible as long as it is clear that a function with that type applied to an argument of type $\mathtt{S}_2$ will return results in $\mathtt{T}_1\mathtt{\&}\mathtt{T}_2$. 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 $\mathtt{S}_1\mathtt{<:}\mathtt{S}_2$ the type printed for and thought by the programmer for a function of type $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$ is instead $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_1\mathtt{\&}\mathtt{T}_2)$ —which, by the way, is the same type—. This is what the covariance condition does: when adding a new code for $\mathtt{S}_2$ inputs to a function of type $\mathtt{S}_1\mathtt{-->}\mathtt{T}_1$ with $\mathtt{S}_2\mathtt{<:}\mathtt{S}_1$, it forces the programmer to write this code so that the return type $\mathtt{T}_2$ declared by the programmer for this code is a subtype of $\mathtt{T}_1$, so that the types $\mathtt{S}_2\mathtt{-->}\mathtt{T}_1\mathtt{\&}\mathtt{T}_2$ and $\mathtt{S}_2\mathtt{-->}\mathtt{T}_2$ are exactly the same. Likewise, a type $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$ with $\mathtt{S}_1\mathtt{\&}\mathtt{S}_2$ non-empty is a perfectly fine type (while in the type-system of the original covariance vs. contravariance paper, we banned it because it did not respect the ambiguity free condition). Again it is a language design problem to ensure that whenever we have a function of that type, then the code executed for each combination of the types of the arguments is not only unambiguously defined, but also easily predictable for the programmer.

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. The first version of this work was made available on my web page in 2013. Since then, this work benefited from the reading and suggestions of many people, and I regularly updated it to take this feedback into account. 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 `Perl6-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 15,

Yary Hluchan who suggested the code in Footnote 22 and Todd Hepler. Andrew M. Kent found some really subtle errors in Section 4 and suggested the right fixes for some of them: to fix the others I profited of the unvaluable help of Tommaso Petrucciani. Last but not least, I want to thank Anita Badyl, Juliusz Chroboczek, Jim Newton, Kim Nguyen, and Luca Padovani who provided useful feedback.

REFERENCES

[1] E. Allen, D. Chase, J.J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*, 2008. version 1.0.

[2] Davide Ancona, Giuseppe Castagna, Tommaso Petrucciani, and Elena Zucca. Semantic subtyping for non-strict languages. In *24th International Conference on Types for Proofs and Programs (TYPES 2018)*, June 2018.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119(2):202–230, 1995.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[5] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. Static and dynamic semantics of NoSQL languages. In *POPL '13, 40th ACM Symposium on Principles of Programming Languages*, pages 101–113, 2013.

[6] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In *ECOOP '96, 10th European Conference on Object-Oriented Programming*, number 1098 in LNCS, pages 3–25. Springer, 1996.

[7] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *OOPSLA '97, 12th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 32(10) of *SIGPLAN Notices*, pages 66–76, 1997.

[8] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[9] K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphism of types. *Math. Struct. in Comp. Science*, 1:1–20, 1991.

[10] K. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, volume 1241 of *LNCS*, pages 104–127. Springer, 1997.

[11] K.B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[12] G. Castagna. *Overloading, subtyping and late binding: functional foundation of object-oriented programming*. PhD thesis, Université Paris 7, January 1994.

[13] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[14] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997. ISBN 3-7643-3905-5.

[15] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisboa, Portugal, 2005. ACM Press. Joint ICALP-PPDP keynote talk.

[16] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, San Francisco, July 1992. ACM Press. Extended abstract.

[17] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[18] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In *POPL '15, 42nd ACM Symposium on Principles of Programming Languages*, pages 289–302, January 2015.

[19] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions 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.

[20] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*, pages 94–106, September 2011.

[21] *The CDuce Programming Language*. Available at `www.cduce.org`.

[22] C. Chambers. *The Cecil language: specification and rationale, Version 3.2*, 2004.

[23] C. Clifton, T. Millstein, G.T. Leavens, and C. Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):517–575, 2006.
[24] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Université Paris Diderot, December 2004.
[25] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
[26] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008.
[27] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.
[28] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001.
[29] H. Hosoya and B.C. Pierce. XDuce: A statically typed xml processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
[30] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.
[31] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21(11) of *SIGPLAN Notices*, pages 347–349, November 1986.
[32] Andrew M. Kent. Down and dirty with semantic set-theoretic types (a tutorial). Available on-line at `https://pnwamk.github.io/sst-tutorial`.
[33] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, 2002.
[34] *Perl 6 Synopsis*. Available at `www.perl6.org`.
[35] B.C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

APPENDIX A. EXERCISE SOLUTIONS

[EX1] Problem: find a function that is in $(\mathtt{S}_1\mathtt{|}\mathtt{S}_2)\mathtt{-->}(\mathtt{T}_1\mathtt{|}\mathtt{T}_2)$ and not in $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$.

Solution: this is possible only if both $\mathtt{S}_1\mathtt{-->}\mathtt{T}_1 \not\simeq \mathtt{S}_2\mathtt{-->}\mathtt{T}_2$ and viceversa. So let us suppose that both $\mathtt{S}_1\setminus\mathtt{S}_2$ —ie, $\mathtt{S}_1\mathtt{\&not(}\mathtt{S}_2\mathtt{)}$, that is the difference between $\mathtt{S}_1$ and $\mathtt{S}_2$— and $\mathtt{T}_2\setminus\mathtt{T}_1$ are not empty types. Then, any function in $(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\mathtt{\&}(\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$ that maps at least one value in $\mathtt{S}_1\setminus\mathtt{S}_2$ into a value in $\mathtt{T}_2\setminus\mathtt{T}_1$ is a solution of the problem.

[EX2] Problem: prove the relations in Footnote 10 and their converse

Solution: Every function in $(\mathtt{S}\mathtt{-->}\mathtt{T})\mathtt{\&}(\mathtt{S}\mathtt{-->}\mathtt{U})$ 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-->T&U`. Conversely, every function in `S-->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 $(\mathtt{S}\mathtt{-->}\mathtt{T})\mathtt{\&}(\mathtt{S}\mathtt{-->}\mathtt{U})$.

Every function in $(\mathtt{S}\mathtt{-->}\mathtt{U})\mathtt{\&}(\mathtt{T}\mathtt{-->}\mathtt{U})$ 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|T` it returns a result in `U`, and thus it is in `S|T-->U`. Conversely, every function in `S|T-->U` when applied to a value in `S|T` 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 $(\mathtt{S}\mathtt{-->}\mathtt{U})\mathtt{\&}(\mathtt{T}\mathtt{-->}\mathtt{U})$.

[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 $y) { sum($x , $y>0) }
multi sub sum(Int $x, Bool $y) { sum($y , $x) }
```

has type

```
   ((Int,Int)-->Int)
&  ((Bool,Bool)-->Bool)
&  ((Bool,Int)-->Bool)
&  ((Int,Bool)-->Bool).
```
(A.1)

Solution: This is a recursive definition. So we have to prove under the hypothesis that the recursion variable `sum` has type (A.1) 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 , $x)`. By hypothesis `sum` has the type in (A.1) and it is here applied to a first argument `$y` of type `Bool`and to a second argument `$x` of type `Int`. The type in (A.1) tells us that when a function of this type is applied to argument in `(Bool,Int)`, then the function returns a result in `Bool` (third arrow of the intersection in (A.1)), which is the expected result.

[EX4] Problem: Prove that the type in (2.11) and in Footnote 14 are equivalent.

Solution: The exercise in Footnote 10 states that `S-->T&U` is equivalent to `(S-->U)&(T-->U)` for all types `S`, `T`, and `U`. So in particular, `Bool|Int-->Bool` is equivalent to `(Bool-->Bool) & (Int-->Bool)` from which the result follows.

[EX5] Problem: find a function that is in (2.16) and not in (2.17).

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 (4.5) is formed by a single product. Then we have the following recursive definition for the product case of the subtyping relation:

$$(\mathtt{S}_1\,,\mathtt{S}_2)\ \ \mathtt{<:}\ \bigvee_{(\mathtt{T}_1\,,\mathtt{T}_2)\in N}(\mathtt{T}_1\,,\mathtt{T}_2) =$$
$$\mathtt{S}_1\mathtt{<:Empty}\ \ \text{or}\ \ \mathtt{S}_2\mathtt{<:Empty}\ \ \text{or}\ \ \Phi(\mathtt{S}_1,\mathtt{S}_2,N)$$

where

$$\Phi(\mathtt{S}_1\ ,\ \mathtt{S}_2\ ,\ \varnothing) = \mathtt{false}$$

$$\begin{aligned}\Phi(\mathtt{S}_1\ ,\ \mathtt{S}_2\ ,\ N \cup \{(\mathtt{T}_1\ ,\mathtt{T}_2)\}) = \\ \quad ((\mathtt{S}_1\mathtt{<:}\mathtt{T}_1)\ \mathsf{or}\ \Phi(\mathtt{S}_1 \setminus \mathtt{T}_1\ ,\ \mathtt{S}_2\ ,\ N))\quad \mathsf{and} \\ \quad ((\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)\ \mathsf{or}\ \Phi(\mathtt{S}_1\ ,\ \mathtt{S}_2 \setminus \mathtt{T}_2\ ,\ N))\end{aligned}$$

The justification of the above definition and several possible optimizations for this algorithm can be found in Section 7.3.1 of [24].

[EX7] Problem: modify the definitions of Φ and Φ' so that they do not perform any check for (4.8) when $P' = P$.

Solution: The case $P' = P$ corresponds to a call $\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ P^\circ\ ,\ P^+\ ,\ P^-)$ in which both P° and P^- are empty, that is $P^+ = P$. In the formulation given for (4.9) this corresponds to the case in which the last parameter of Φ was never modified and, therefore, is still `Any`. Concretely, this corresponds to adding to the subsequent definition a further case of termination of the form

$$\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing\ ,\ \mathtt{D}\ ,\ \mathtt{Any}) = \mathtt{true}$$

or, equivalently, to modifying the termination case as follows

$$\Phi(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \varnothing\ ,\ \mathtt{D}\ ,\ \mathtt{C}) = (\mathtt{Any}\mathtt{<:}\mathtt{C})\ \mathsf{or}\ (\mathtt{T}_1\mathtt{<:}\mathtt{D})\ \mathsf{or}\ (\mathtt{C}\mathtt{<:}\mathtt{T}_2)$$

where the clauses are evaluated from left to right.

For what concerns Φ', the case for $P' = P$ corresponds to a call in which the second parameter was never modified (i.e., it still is the $\mathtt{not}(\mathtt{T}_2)$ of the initial call in (4.10)) and the last parameter is empty. In order to verify the first condition, we may add an extra parameter to Φ' that indicates whether the second parameter was modified or not. If this is not case and we are at the end of the recursive calls (i.e., just one element remainins to be selected), then we have to perform just one of the two recursive calls, that is:

$$\Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\ ,\ \{\mathtt{S}_1^\circ\mathtt{-->}\mathtt{S}_2^\circ\}) = \Phi'(\mathtt{T}_1\ ,\ \mathtt{T}_2\mathtt{\&}\mathtt{S}_2^\circ\ ,\ \varnothing)$$

As we see, this optimization is mildly interesting, since it just spares the check $\Phi'(\mathtt{T}_1 \setminus \mathtt{S}_1^\circ\ ,\ \mathtt{T}_2\ ,\ \varnothing)$ (in practice, it avoids performing the difference $\mathtt{T}_1 \setminus \mathtt{S}_1^\circ$) which is why it is not implemented by the compiler of $\mathbb{C}$Duce.

[EX8] Problem: prove that a disjunctive normal form of tags can always be expressed by either $(t_1 \vee \ldots \vee t_n)$ or $\neg(t_1 \vee \ldots \vee t_n)$.

Solution: First of all notice that `Any` can be represented as $\neg()$ the negation of an empty union of tags. Next, take any intersection $\bigwedge_{p\in P} t_p$ `&` $\bigwedge_{n\in N} \mathtt{not}(t_n)$. If P is not 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 $\bigwedge_{p\in P} t_p$ `&` $\bigwedge_{n\in N} \mathtt{not}(t_n)$ reduces either to `Empty` or to `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 $\bigwedge_{n\in N} \mathtt{not}(t_n)$ are exactly all the tag values, apart from all t_n for $n \in N$. That is, the set of tags in $\bigwedge_{n\in N} \mathtt{not}(t_n)$ are exactly those in the co-finite union $\mathtt{not}(\bigvee_{n\in N} t_n)$. 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 unites 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 form are removed from it and all the other co-finite unions, and the cofinite unions are merged into a single one).

[EX9] Problem: define `norm`

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

```
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,T) -> { empty with prod = (S,T)?𝟙:𝟘:𝟘 }
 | S-->T -> { empty with arrw = S-->T?𝟙:𝟘:𝟘 }
 | S|T   -> (norm S)∨(norm T)
 | S&T   -> (norm S)∧(norm T)
 | not(T)-> any\(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 `(norm S,norm T)?𝟙:𝟘:𝟘` instead of `(S,T)?𝟙:𝟘:𝟘` (and similarly for arrows).

[EX10] Problem: Show that the type system is unsound for lazy languages since the application of a well-typed function to a diverging expression of type `Empty` is well typed.

Solution: Consider the function `sub doublefirst(Int $x, Int $y){ x+x }`. This function is of type `(Int , Int)-->Int`. By subtyping it has also the type `Empty-->Int` (contravariance on the domain). So we can apply it to any argument of type `Empty`. Now consider the pair `(True,`e`)` where e is any diverging function of type `Empty`. This pair has also type `Empty`, since the product

with an empty set is itself an empty set. Therefore, the application `doublefirst(true,`e`)` is well typed with type `Int`. In a strict language this does not pose any problem since the evaluation of the argument will never terminate, and so will not the application. But in a lazy language this application reduces to `True+True`, thus yielding a type error at run-time (or, at least, it should: in Perl 6 it returns 2).

[EX11] Problem: Prove that by applying the *Step 4* of the subtyping algorithm of Section 4.2 to $(\mathtt{S}_1\,,\mathtt{S}_2)\&\mathtt{not}((\mathtt{T}_1\,,\mathtt{T}_2))$ we obtain $(\mathtt{S}_1\mathtt{<:Empty})$ or $(\mathtt{S}_1\mathtt{<:Empty})$ or $(\mathtt{S}_1\mathtt{<:}\mathtt{T}_1$ and $\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)$.

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

$$(\mathtt{S}_1\mathtt{<:Empty}\text{ or }\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)\text{ and }(\mathtt{S}_1\mathtt{<:}\mathtt{T}_1\text{ or }\mathtt{S}_2\mathtt{<:Empty})$$

By distributing the "and" we obtain:

$$\begin{array}{l}(\mathtt{S}_1\mathtt{<:Empty}\text{ and }\mathtt{S}_1\mathtt{<:}\mathtt{T}_1)\text{ or}\\(\mathtt{S}_1\mathtt{<:Empty}\text{ and }\mathtt{S}_2\mathtt{<:Empty})\text{ or}\\(\mathtt{S}_2\mathtt{<:}\mathtt{T}_2\text{ and }\mathtt{S}_2\mathtt{<:Empty})\text{ or}\\(\mathtt{S}_2\mathtt{<:}\mathtt{T}_2\text{ and }\mathtt{S}_1\mathtt{<:}\mathtt{T}_1)\end{array}$$

By observing that $\mathtt{S}_1\mathtt{<:Empty}$ implies $\mathtt{S}_1\mathtt{<:}\mathtt{T}_1$, that $\mathtt{S}_2\mathtt{<:Empty}$ implies $\mathtt{S}_2\mathtt{<:}\mathtt{T}_2$, and that both imply $(\mathtt{S}_1\mathtt{<:Empty}\text{ and }\mathtt{S}_2\mathtt{<:Empty})$, we obtain the result.

[EX12] Problem: Prove that by applying the *Step 4* of the subtyping algorithm of Section 4.2 to $\mathtt{S}_1\mathtt{-->}\mathtt{S}_2\&\mathtt{not}(\mathtt{T}_1\mathtt{-->}\mathtt{T}_2)$ we obtain $(\mathtt{T}_1\mathtt{<:Empty})$ or $(\mathtt{T}_1\mathtt{<:}\mathtt{S}_1$ and $\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)$

Solution: We have check that two conditions are satisfied, namely the condition on the domains and the "or" for the case $P' = \varnothing$. These yield:

$$(\mathtt{T}_1\mathtt{<:}\mathtt{S}_1)\text{ and }(\mathtt{T}_1\mathtt{<:Empty}\text{ or }\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)$$

By distributing the "and" we obtain

$$(\mathtt{T}_1\mathtt{<:}\mathtt{S}_1\text{ and }\mathtt{T}_1\mathtt{<:Empty})\text{ or }(\mathtt{T}_1\mathtt{<:}\mathtt{S}_1\text{ and }\mathtt{S}_2\mathtt{<:}\mathtt{T}_2)$$

By observing that $\mathtt{T}_1\mathtt{<:Empty}$ implies $\mathtt{T}_1\mathtt{<:}\mathtt{S}_1$ we obtain the result.