Typing Records, Maps, and Structs

GIUSEPPE CASTAGNA, CNRS, Université Paris Cité, France

Records are finite functions from keys to values. In this work we focus on two main distinct usages of records: structs and maps. The former associate different keys to values of different types, they are accessed by providing nominal keys, and trying to access a non-existent key yields an error. The latter associate all keys to values of the same type, they are accessed by providing expressions that compute a key, and trying to access a non-existent key usually yields some default value such as `null` or `nil`. Here, we propose a type theory that covers both kinds of usage, where record *types* may associate to different types either single keys (as for structs) or sets of keys (as for maps) and where the same record *expression* can be accessed and used both in the struct-like style and in the map-like style we just described. Since we target dynamically-typed languages our type theory includes union and intersection types, characterized by a subtyping relation. We define the subtyping relation for our record types via a semantic interpretation and derive the decomposition rules to decide it, define a backtracking-free subtyping algorithm that we prove to be correct, and provide a canonical representation for record types that is used to define various type operators needed to type record operations such as selection, concatenation, and field deletion.

CCS Concepts: • **Theory of computation** → *Type structures*; *Program analysis*; • **Software and its engineering** → *Functional languages*.

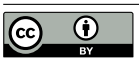
Additional Key Words and Phrases: dictionaries, subtyping, union types, intersection types, dynamic languages.

ACM Reference Format:
Giuseppe Castagna. 2023. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP, Article 196 (August 2023), 45 pages. https://doi.org/10.1145/3607838

1 INTRODUCTION

In 1965, C.A.R. Hoare proposed in a series of papers on "record handling" an extension for a general purpose language, there supposed to be ALGOL 60, to manipulate an arbitrary number of *records* each belonging to a limited number of *record classes* [Hoare 1965, 1966a,b]. Not only the proposal was rapidly adopted by ALGOL, but Dahl and Nygaard also adapted it for their language, Simula I, yielding the concepts of objects and classes (Simula 67). Ever since then, records have become the Swiss Army knife data structure of modern computer languages. They are used for a wide palette of purposes such as relations (as in relational databases), maps (a.k.a., associative arrays, dictionaries, hashes, lookup tables), modules, objects, configuration files, data serialization. JSON, the *de facto* standard format for data-interchange, is a language formed essentially of records and lists thereof. The same holds true for YAML, a different data serialization format used for configuration files.

In general terms, record values are sets of key and value pairs in which all keys are pairwise distinct and are used to access the values they are paired with. In this work, we focus on two main distinct usages of records: structs and maps. In a nutshell, the former associate different keys to values of different types, they are accessed by providing nominal keys, and trying to access a non-existent key yields an error; the latter associate all keys to values of the same type, they are accessed by providing expressions that compute a key, and trying to access a non-existent key usually yields some default value such as `null` or `nil`. More generally, the main differences from a



2475-1421/2023/8-ART196
https://doi.org/10.1145/3607838

programming language point of view can be summarized as follows (though exceptions apply in each case):

Maps:
- All keys of a single map have the same type and so do the values they are mapped to.
- It is not necessary to know all the keys at compile time: they can be dynamically discovered.
- It is sensible to give a default value.
- Keys may be indexed: it is possible to iterate over them.
- Keys are values: keys used for map selection can be results of expressions.
- Accessing a key that is not defined does not yield an error
- Maps are (often) mutable data structures with mutable components.

Structs:
- Different keys in the same structure can be mapped to values of different types.
- Keys may be restricted to a specific set (e.g., strings, atoms, identifiers, ...).
- It is necessary to know all the different keys at compile time.
- Keys do not support indexing.
- Keys are not necessarily values, they may form a separate set of names.
- Accessing a key that is not defined yields an error.
- Structs are (often) immutable data-structures but may have mutable components.

Besides these linguistic differences, maps and structs may have different implementations, typically, hashes or search trees for maps; arrays or contiguous locations for structs.[1]

When both are available, the choice to use one or the other depends on the nature of the application; in some cases, even for the same application both choices may reasonably apply, and the final choice depends on the amount of information that is available about the data to process: for instance, to parse JSON we want to use structs if we know the field name and data type of each JSON element, while for parsing unknown JSON data we should definitively use maps.

In some languages, the type used for keys is different between structs and maps (e.g., in Elixir [Elixir] struct keys must be *atoms*—cf. Footnote 7, page 14—, while map keys can be any value, even functions; differently, in Hive [Hive] struct keys are not values—just names—, while map keys are values of primitive types). Sometimes, a different terminology is used, so the identifiers for fields in a struct are usually called "labels", "field names", "attribute names", while for maps the word "key" is prevalent. We will use both "labels" and "keys", privileging the latter when speaking of maps.

Some languages do not make any syntactic distinction for defining maps and structs (e.g., Lua [Lua], Ballerina [Ballerina]), in other languages they are distinct but tightly related (e.g., in Elixir, structs are wrappers around maps providing them with further capabilities), others make them completely disjoint (e.g., Go Language [Go], Erlang [Erlang], Swift [Swift]); in the last case there is a wealth of blog posts and tutorials about when to use one rather than the other. Many languages do not provide them as primitive data types, but rather as libraries or specific classes, in which case they lack specific types for them (e.g., Ruby [Ruby] and Scala [Scala] provide classes for both structs and maps, F# [F#], JavaScript, Julia [Julia], OCaml [OCaml], and Rust [Rust] have primitive objects/records/structs and libraries/classes for maps, while Python and Perl have primitive maps/dictionaries/hashes and records/structs are provided by external libraries/classes).

While it is sensible to have different implementations for maps and structs, it is less justified to have different types for them, especially in languages in which the two data structures share the same or similar syntax for expressions and the same set of operations (as a matter of fact, both are finite functions from keys to values). This is the case for some object-oriented languages in

[1]Typically but not necessarily: for instance, Erlang makes this distinction already for maps according to whether they have more or less than 32 fields, hashes in the former case and contiguous locations in the latter.

which maps have all the same interface typing different implementations (e.g., in Java); a language like Ballerina permits to specify in a record type a default type for unspecified fields (thus mixing structs and maps characteristics) while Erlang's and Elixir's Typespecs permit mixing in the same record type the type specification of single fields (as in structs) and of mappings from key types to value types (as in maps) [Elixir; Erlang]. The latter case is the inspiration of our work.

In this article, we argue that a single (syntactic) representation for both struct and maps is possible, and that it can be typed by a unique type constructor that covers both cases. In this case, we will speak generically of "record expressions" (or just "records") as sets of associations between keys and values. The corresponding "record types" will be sets of associations from "key-types" to types, each key-type being either a single key (as in the types for struct) or a set of keys (as in the types for maps). What distinguishes records used as maps from records used as structs is the way they are accessed. We distinguish two kinds of accesses: map-like access and struct-like access. When using records as maps, it must be possible to compute the key to access a field (i.e., to obtain keys as the result of an expression) and trying to access a key that is not present in the record should *not* produce a run-time error. When using records as structs, fields are accessed directly by the nominal key and trying to access a key that is not present in the struct yields an error (preferably, a statically-detected one). Of course, it will be possible to use both kinds of access on a same record: the different semantics will be reflected by different typing rules for the access expressions. Some languages already make this distinction for accesses: for instance in Elixir (but it is the same syntax as in JavaScript and in Ballerina) if r is a record expression, `key` is a nominal key, and *keyexp* an expression that computes a key, then r`.key` raises an exception if a field with key `key` is not present in the result of r (i.e., struct-like access), while r`[`*keyexp*`]` returns `nil` if the key resulting from *keyexp* is not present in (the result of) r (i.e., map-like access). Likewise, Scala uses the syntax r`(key)` for struct-like access and r `get` *keyexp* for map-like access; however Scala also permits the use of generic expressions in the former—as in r`(`*keyexp*`)`—while Elixir has a specific access function for this case—i.e., `fetch!(`r`,`*keyexp*`)` (cf. Section 4.5).

Contributions. *Per se*, defining a type system that encompasses both structs and maps does not look like an insurmountable task, and certainly it is not so hard to do it for languages with simple or no subtyping relation. The challenge here is that we aim at defining types for dynamic languages such as JavaScript, Erlang, or Elixir. As all current attempts demonstrate (e.g., Flow, TypeScript, Hack, ...) this requires types that include union and intersection type connectives, which imply the presence of a sophisticated subtyping relation. Thus, the real challenge we tackle here is to define a type system with union, intersection, and record (i.e., struct+map) types. This means to show how to define and efficiently decide the subtyping relation for these types, and how to type record operations for expressions whose types may be arbitrary combinations of unions and intersections of record types. In particular, our technical contributions can be summarized as follows: we introduce record types that cover both struct-like and maps-like usages (Section 4); we define a new family of functions that we call "quasi $\mathbb{K}$-step functions" (Definition 4.6); we define the subtyping relation of our record types by interpreting them as sets of quasi $\mathbb{K}$-step functions (Formula (21)), derive from this interpretation the decomposition rules to be used to decide the subtyping relation, and prove it correct (Lemma 4.7); we define a backtracking-free subtyping algorithm Φ for different variations of our record types (formulas (17–19), and (22)) and prove it correct (Theorem 4.5); we introduce and formalize different operations on records such as map/struct field selection, record concatenation, and map/struct field deletion and provide a canonical representation for record types which is used to define various type operators needed to type these record operations (Sections 4.2 and 4.4). Finally, we also study how to extend this theory to account for mutation since, in modern languages, records are often also mutable structures, in

which fields can be added removed, and modified; for space reasons this extension is not presented in the main text, but the motivated reader will find its presentation in Appendix F.

We want to conclude this discussion on our contributions, by stressing that we think that the interest of this work is less in defining a catch-them-all theory for the different usage of records, than in developing some general implementation guidelines that, with minimal adaptions, can capture all the aspects outlined above. This is not just a theoretical challenge: at the moment of writing the very implementation techniques introduced here are being experimented by the development teams of Elixir and Ballerina languages to be included in their compilers.

Outline. Section 2 gives the background for our presentation—i.e., semantic subtyping [Frisch et al. 2008] and the theory of *quasi-constant* functions [Frisch 2004]—and covers relevant related work. Section 3 introduces a simple record calculus where records are typed as structs. Section 4 is the core of our contribution: we extend the expressions and types of the previous calculus to deal with generic maps; we give a representation of record types as unions of some specific record type atoms (§4.1) and use this representation to define type operators used to type various record operations, namely, map and struct selection, map and struct deletion, and record concatenation (§4.2); we use the representation to prove the correctness of a backtracking-free subtyping algorithm (§4.3); we refine the theory to allow record types to specify maps from specific sets of keys, which in particular requires generalizing *quasi-constant* functions to *quasi* $\mathbb{K}$*-step* functions together with the corresponding decomposition rule (§4.4); finally, we discuss our design choices and several possible variations of the features presented (§4.5). Section 5 discusses related work. Section 6 concludes the presentation by a more detailed analysis of the contributions of this work and by describing current and future work on the subject. This work includes an appendix containing material that, for space constraints, could not be presented in the main text, namely, extra definitions, several proofs, and the extension of the theory to account for mutatable data structures (the appendix is available on the ACM Digital Library in supplemental material section).

2 BACKGROUND

In this section we outline the two pillars on which our record system is built. The first is semantic subtyping [Frisch et al. 2008], a technique to endow set-theoretic type connectives in a type system. The second is the theory of quasi-constant functions [Frisch 2004] that we use to give semantics to record expressions and types. The reader can refer to the cited works for more details.

2.1 Semantic subtyping

Semantic subtyping [Frisch et al. 2002, 2008], [Castagna and Frisch 2005] is a technique to add union, intersection, and negation type connectives to a type system so that the types satisfy all the commutative and distributive laws we expect from their set-theoretic interpretation. It is a general approach on which records types are grafted in next section. The key of the approach is the subtyping relation which is defined by giving an interpretation $[\![\cdot]\!]$ of types as sets and then defining $t_1 \leq t_2$ as the inclusion of the interpretations, that is, $t_1 \leq t_2 \stackrel{\text{def}}{=} [\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values that inhabit the type t in the language. By interpreting union, intersection, and negation as the corresponding operations on sets and by giving appropriate interpretations to the other constructors, we ensure that subtyping will satisfy all expected laws.

2.1.1 Types. Formally, we proceed as follows. We first fix two countable sets: a set $\mathcal{C}$ of *language constants* (ranged over by c) and a set $\mathcal{B}$ of *basic types* (ranged over by b). For example, we can take constants to be Boolean values and integers: $\mathcal{C} = \{\texttt{true}, \texttt{false}, \texttt{0}, \texttt{1}, \texttt{-1}, \dots\}$. $\mathcal{B}$ might then contain Bool and Int; however, we also assume that, for every constant c, there is a "singleton" basic type

which corresponds to that constant alone (for example, a type for `true`, which will be a subtype of Bool). We assume that a function $\mathbb{B} : \mathcal{B} \to \mathcal{P}(\mathcal{C})$ assigns to each basic type the set of constants of that type and that a function $b_{(\cdot)} : \mathcal{C} \to \mathcal{B}$ assigns to each constant c a basic type b_c such that $\mathbb{B}(b_c) = \{c\}$.

Definition 2.1 (Types). *The set $\mathcal{T}$ of* types *is the set of terms t that are coinductively produced by*

$$t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

and that satisfy two additional constraints: (1) regularity*: the term must have a finite number of different sub-terms;* (2) contractivity*: every infinite branch must contain an infinite number of occurrences of the product or arrow type constructors.*

We use the abbreviations $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$ (in particular, $\mathbb{1}$ is the supertype of all types, sometimes denoted by `any` or `top`). We refer to b, $\times$, and $\to$ as *type constructors*, and to $\vee$, $\neg$, $\wedge$, and $\setminus$ as *type connectives*. As customary, connectives have priority over constructors and negation has the highest priority—e.g., $\neg s \vee t \to u \wedge v$ denotes $((\neg s) \vee t) \to (u \wedge v)$.

Coinduction accounts for recursive types, and it is coupled with a contractivity condition which excludes infinite terms that do not have a meaningful interpretation as types or sets of values: for instance, the trees satisfying the equation $t = t \vee t$ (which gives no information on which values are in it) or $t = \neg t$ (which cannot represent any set of values) do not satisfy the contractivity condition. Contractivity also gives an induction principle on $\mathcal{T}$ that allows us to apply the induction hypothesis below type connectives (union and negation), but not below type constructors (product and arrow): we use it in Section 2.1.2 in the definition of a relation noted $(d : t)$. As a consequence of contractivity, types cannot contain infinite unions or intersections. The regularity condition is necessary to ensure the decidability of the subtyping relation.

To define semantic subtyping we must define a type interpretation $[\![.]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ which interprets types into sets of elements of $\mathcal{D}$ for some suitable domain $\mathcal{D}$. To ensure that type connectives have a set-theoretic semantics, the definition must satisfy (i) $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$, (ii) $[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$, and (iii) $[\![\mathbb{0}]\!] = \varnothing$. Next we must find a domain $\mathcal{D}$ in which it is possible to give a set-theoretic interpretation of the type constructors and in particular of function spaces. Intuitively, if we interpret functions as binary relations on $\mathcal{D}$, then the interpretation of $t_1 \to t_2$ should be the set of binary relations in which if the first projection is in (the interpretation of) t_1, then the second projection is in (the interpretation of) t_2. In other words, $[\![t_1 \to t_2]\!]$ should denote the set $\mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$, where the over-line denotes set complement.[2] But this would imply $\mathcal{P}(\mathcal{D}^2) \subseteq \mathcal{D}$, which is impossible for cardinality reasons. Frisch et al. [2008] proved that one obtains the same subtyping relation by considering the set of finite approximations of the functions in $t_1 \to t_2$. This corresponds to replacing $\mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$ by $\mathcal{P}_{\text{fin}}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$ where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets.[3] This yields the interpretation that we define next.

2.1.2 Type interpretation and subtyping relation. The interpretation domain $\mathcal{D}$ can be defined inductively as the set of terms produced by the following grammar

$$d ::= c \mid (d, d) \mid \{(d, d), ..., (d, d)\}$$

[2] Strictly speaking, the outermost is the complement w.r.t. $\mathcal{D} \times (\mathcal{D} \cup \{\Omega\})$, that is, $\mathcal{P}\left((\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}} \cap (\mathcal{D} \times (\mathcal{D} \cup \{\Omega\}))\right)$ where Ω is a constant such that $\Omega \notin \mathcal{D}$. As explained by Castagna [2023, Section 3.2], Ω is a constant denoting a type-error whose purpose is to avoid $\mathbb{1} \to \mathbb{1}$ to denote the set of all functions (see Frisch et al. [2008, Section 4.2] for full details). Since Ω does not play any role in the interpretation of record types we omit it and use this approximation all the paper long.

[3] Intuitively, this interpretation with finite powersets induces the same subtyping relation as the one induced by generic powersets, since $\mathcal{P}(X) \subseteq \mathcal{P}(Y) \Leftrightarrow \mathcal{P}_{\text{fin}}(X) \subseteq \mathcal{P}_{\text{fin}}(Y)$: see [Frisch et al. 2008] for details.

where $c \in C$. In other terms, the domain is the solution of $\mathcal{D} = C + (\mathcal{D}\times\mathcal{D}) + \mathcal{P}_{\text{fin}}(\mathcal{D}\times\mathcal{D})$. The interpretation function is then defined so that it satisfies the following equalities:

$$\llbracket\mathbb{0}\rrbracket = \varnothing \qquad \llbracket\neg t\rrbracket = \mathcal{D} \setminus \llbracket t\rrbracket \qquad \llbracket t_1 \vee t_2\rrbracket = \llbracket t_1\rrbracket \cup \llbracket t_2\rrbracket$$
$$\llbracket b\rrbracket = \mathbb{B}(b) \qquad \llbracket t_1 \times t_2\rrbracket = \llbracket t_1\rrbracket \times \llbracket t_2\rrbracket \qquad \llbracket t_1 \to t_2\rrbracket = \mathcal{P}_{\text{fin}}(\llbracket t_1\rrbracket \times \overline{\llbracket t_2\rrbracket})$$

where, in particular, we interpret function spaces by considering the finite approximations of their functions. We cannot take the equations above directly as an inductive definition of $\llbracket\cdot\rrbracket$ because types are not defined inductively but coinductively. Therefore, we give a definition which validates these equalities and which uses the aforementioned induction principle on types and structural induction on $\mathcal{D}$. We define $\llbracket.\rrbracket : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ as $\llbracket t\rrbracket = \{d \in \mathcal{D} \mid (d : t)\}$ where $(d : t)$ is the following binary predicate defined by induction on the pair (d, t) ordered lexicographically:

$$\begin{aligned}
(c : b) &= c \in \mathbb{B}(b)\\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2)\\
(\{(d_1, d'_1), \ldots, (d_n, d'_n)\} : t_1 \to t_2) &= \forall i \in \{1, \ldots, n\}.\ \text{if } (d_i : t_1) \text{ then } (d'_i : t_2)\\
(d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2)\\
(d : \neg t) &= \text{not } (d : t)\\
(d : t) &= \text{false} \qquad \text{otherwise}
\end{aligned}$$

The pair $\mathcal{D}$ and $\llbracket.\rrbracket$ is a *set-theoretic model* of types (see [Frisch et al. 2008, Definition 4.4] for the formal definition). It induces the subtyping relation defined as $t_1 \le t_2 \overset{\text{def}}{\iff} \llbracket t_1\rrbracket \subseteq \llbracket t_2\rrbracket$. This particular model is called the *universal* since it induces the best (i.e., largest) possible subtyping relation definable by a set-theoretic model (see Frisch et al. [2008, Section 5.3]).

2.1.3 Subtyping algorithm. The subtyping relation is decidable. Deciding whether t_1 is a subtype of t_2 is equivalent to deciding whether $t_1 \wedge \neg t_2$ is the empty type, insofar as $t_1 \le t_2 \Leftrightarrow \llbracket t_1\rrbracket \subseteq \llbracket t_2\rrbracket \Leftrightarrow \llbracket t_1\rrbracket \cap (\mathcal{D} \setminus \llbracket t_2\rrbracket) \subseteq \varnothing \Leftrightarrow \llbracket t_1 \wedge \neg t_2\rrbracket \subseteq \varnothing \Leftrightarrow t_1 \wedge \neg t_2 \le \mathbb{0}$. The subtyping algorithm relies on the property that every type can be put into a particular disjunctive normal form, that is, it is equivalent to (i.e., it has the same interpretation as) a union of uniform intersections of atoms and negations of atoms, where an atom is either a product, an arrow, or a basic type, and intersections are uniform when they are composed only of atoms of the same constructor (i.e., all arrows, all products, all basic types). To decide $t_1 \le t_2$ the system puts $t_1 \wedge \neg t_2$ in normal form, that is, it transforms it into a union of intersections that have one of the following three forms

$$\bigwedge_{b\in P} b \wedge \bigwedge_{b'\in N} \neg b' \qquad \bigwedge_{t_1\times t_2\in P} t_1\times t_2 \wedge \bigwedge_{t'_1\times t'_2\in N} \neg(t'_1\times t'_2) \qquad \bigwedge_{t_1\to t_2\in P} t_1\to t_2 \wedge \bigwedge_{t'_1\to t'_2\in N} \neg(t'_1\to t'_2) \tag{1}$$

and checks the emptiness of the union, which is equivalent to checking the emptiness of all intersections that compose it. Emptiness for the first form in (1) can be checked directly. The emptiness of the other two forms is checked by decomposing them into simpler subtyping problems and recording them as hypothesis for coinduction. More precisely, the intersection of products in (1) is empty if and only if: [4]

$$\forall N' \subseteq N.\quad \left(\bigwedge_{t_1\times t_2\in P} t_1 \le \bigvee_{t'_1\times t'_2\in N'} t'_1\right) \quad \text{or} \quad \left(\bigwedge_{t_1\times t_2\in P} t_2 \le \bigvee_{t'_1\times t'_2\in N\setminus N'} t'_2\right) \tag{2}$$

A similar decomposition for the intersections of arrows can be found in [Frisch et al. 2008, Section 6.2] (we also recall it in Appendix A) together with the proofs of soundness, correctness, and termination of the algorithm (the last follows from the regularity of the types).

[4] To understand the rationale of this transformation the reader can consider the case in which both P and N contain just one atom, namely, the case for $t_1\times t_2 \le t'_1\times t'_2$. There are just two cases to check ($N'=\varnothing$ and $N'=N$) and it is not difficult to see that the condition above becomes: ($t_1 \le \mathbb{0}$) or ($t_2 \le \mathbb{0}$) or ($t_1 \le t'_1$ and $t_2 \le t'_2$), as expected.

2.1.4 Implementation. A naive implementation of the decomposition (2) would compute all the subsets of N and check for each of them the "or" clauses in (2). The problem of such an implementation is that if one of the "or" clauses fails, then this invalidates all the hypotheses we added to check it (e.g., to implement coinduction for checking the emptiness of a recursive type, one has to add the hypothesis that the type is empty, and then proceed by applying the decomposition to the type which, in turn, looks for the emptiness of some subterms or the type, and so on and so forth); we thus have to backtrack to the point where the failed hypothesis was added and remove all the hypotheses subsequently introduced, before checking the next "or" clause. Frisch [2004] proved that this can be avoided by defining a Boolean function Φ as follows

$$\Phi(t_1, t_2, \varnothing) = \texttt{false}$$
$$\Phi(t_1, t_2, N \cup \{t_1' \times t_2'\}) = (\,(t_1 \leq t_1') \text{ or } \Phi(t_1 \setminus t_1', t_2, N)\,) \text{ and } (\,(t_2 \leq t_2') \text{ or } \Phi(t_1, t_2 \setminus t_2', N)\,)$$

which clearly does not use backtracking. Φ takes two non-empty types t_1 and t_2 and a set of product atoms N and returns whether $t_1 \times t_2 \leq \bigvee_{t_1' \times t_2' \in N} t_1' \times t_2'$. It is then easy to use Φ to implement (2). Notice that $\bigwedge_{t_1 \times t_2 \in P} t_1 \times t_2 = (\bigwedge_{t_1 \times t_2 \in P} t_1) \times (\bigwedge_{t_1 \times t_2 \in P} t_2)$. Let $s_i = \bigwedge_{t_1 \times t_2 \in P} t_i$ for $i = 1, 2$. Then

$$s_1 \times s_2 \wedge \bigwedge_{t_1' \times t_2' \in N} \neg(t_1' \times t_2') \leq \mathbb{0} \iff s_1 \times s_2 \leq \bigvee_{t_1' \times t_2' \in N} t_1' \times t_2' \iff s_1 \leq \mathbb{0} \text{ or } s_2 \leq \mathbb{0} \text{ or } \Phi(s_1, s_2, N)$$

It is possible to define an analogous Φ-function for arrow atoms. Its definition can be found in [Castagna 2020, Sections 4.2, 4.3] together with a detailed description of how Φ-functions work and of the data-structures to be used to implement them efficiently. The proof of the correctness of their definition can be found in [Frisch 2004, Chapter 7].

2.2 Quasi-constant functions

From their earliest formalizations (e.g., Hoare [1965, 1966a,b], Cardelli [1984], Bruce and Longo [1988]) record values are considered finite mappings from a given set of labels $\mathcal{L}$ to values, that is, functions that map a finite set of labels $\{\ell_1, ..., \ell_n\} \subseteq \mathcal{L}$ into values and are undefined on the other labels, that is, on $\mathcal{L} \setminus \{\ell_1, ..., \ell_n\}$. In order to simplify the presentation, we depart from this view and follow the approach introduced by Alain Frisch in his PhD dissertation [Frisch 2004, Chapter 9] which considers record values (and record types) to be functions that are total on $\mathcal{L}$ and constant on nearly all labels (i.e., on a cofinite subset of $\mathcal{L}$). Frisch calls such functions the *quasi-constant functions*. The idea being that a record value maps nearly all labels to a specific constant (denoting undefined) apart from a finite set of labels which are mapped to the values of the language.

Definition 2.2 ([Frisch 2004]). *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 we denote this set by* $\mathrm{dom}(r)$ *(i.e., the **dom**ain of r) and the element z by* $\mathrm{def}(r)$ *(i.e., the **def**ault value of r). We 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 $z_i \neq z$ and the ℓ_i's to be pairwise distinct), this is largely sufficient for the purposes of this work. If $(Z_\ell)_{\ell \in \mathcal{L}}$ is a family of subsets of Z indexed by $\mathcal{L}$, we denote by $\bowtie_{\ell \in \mathcal{L}} Z_\ell$ the subset of $\mathcal{L} \to\!\!\!\!\!\circ\ Z$ formed by all quasi-constant functions r such that $r(\ell) \in Z_\ell$ for all $\ell \in \mathcal{L}$ (intuitively, $\bowtie_{\ell \in \mathcal{L}} Z_\ell$ is a "type" of quasi-constant functions).

2.2.1 Containment. We are going to use quasi-constant functions to embed records in our type system. More precisely, we will add to types some quasi-constant functions from labels to types,

which will be interpreted set-theoretically as subsets of $\mathcal{L} \to \mathcal{D}$ (precisely, of $\mathcal{L} \to \mathcal{D}_\perp$: see Section 3.2.1). Therefore, we need to extend the subtyping algorithm to check the emptiness of Boolean combinations of such sets. For that we need a decomposition formula analogous to what (2) does for products. This is given by Lemma 9.1 in [Frisch 2004], which is stated as follows (see the cited reference for the proof):

Lemma 2.3 (Frisch [2004]). *Let $(X_p)_{p\in P}$ and $(X_n)_{n\in N}$ be two families of elements of $\mathcal{L} \to \mathcal{P}(\mathcal{D})$. Let $L = \bigcup_{i\in P\cup N} \mathrm{dom}(X_i)$. Then $\left(\bigcap_{p\in P} \boxplus_{\ell\in\mathcal{L}} X_p(\ell)\right) \subseteq \left(\bigcup_{n\in N} \boxplus_{\ell\in\mathcal{L}} X_n(\ell)\right)$ if and only if either $\bigcap_{p\in P} \mathrm{def}(X_p) = \emptyset$, or for every map $\iota : N \to L\cup\{\_\}$*

$$\left(\exists \ell\in\mathcal{L}.\left(\bigcap_{p\in P} X_p(\ell) \subseteq \bigcup_{n\in N|\iota(n)=\ell} X_n(\ell)\right)\right) \quad \text{or} \quad \left(\exists n_\circ\in N.\left(\iota(n_\circ) = \_\right) \text{ and } \left(\bigcap_{p\in P} \mathrm{def}(X_p) \leq \mathrm{def}(X_{n_\circ})\right)\right)$$

A detailed explanation of the formula in the statement is given by Castagna [2020, Section 4.5].

2.2.2 Merge operator. The last ingredient we need to define is a specific operator on quasi-constant functions, that we will use to give semantics to record concatenation and field deletion.

Let Z denote some set, we endow the set $\mathcal{L} \to Z$ with a merge operator parametric in an element of Z. Given $z \in Z$ and $r_1, r_2 \in \mathcal{L} \to Z$, the operator $r_1 \oplus_z r_2$ returns the quasi-constant function $r \in \mathcal{L} \to Z$ such that $r(\ell) = r_1(\ell)$ if $r_1(\ell) \neq z$ and $r(\ell) = r_2(\ell)$ if $r_1(\ell) = z$. That is, the r_1 mappings equal to z are replaced by the r_2's corresponding ones.

3 A SIMPLE STRUCT CALCULUS

In this section we present the basic record calculus, in which records play just the role of structs and on which we build the rest of our theory (in particular, records as maps: cf. Section 4). It is a lambda-calculus typed with set-theoretic types to which we add some specific quasi-constant functions on types and expressions. We also add pattern matching, but we do it just at the end of the section so as to focus on records first.

3.1 Expressions

Let $\mathcal{L}$ be a countable set of labels ranged over by ℓ and C a countable set of constants ranged over by c. We consider the set of expressions $\mathcal{E}$ ranged over by e and the set of values $\mathcal{V}$ ranged over by v defined as follows (where I is finite):

$$\begin{aligned} \textbf{Expressions} \quad & e ::= c \mid \lambda^{\wedge_{i\in I} t_i \to t_i} x.e \mid x \mid e\ e \mid \{\ell = e, \dots, \ell = e\} \mid e + e \mid e.\ell \mid e\backslash\ell \\ \textbf{Values} \quad & v ::= c \mid \lambda^{\wedge_{i\in I} t_i \to t_i} x.e \mid \{\ell = v, \dots, \ell = v\} \end{aligned}$$

This is the functional core of CDuce [Benzaken et al. 2003] where pairs have been replaced by records.[5] The functional part is a λ-calculus with constants in which λ-abstractions are explicitly annotated by their (intersection) type. Record expressions have the form $\{\ell_1 = e_1, \dots, \ell_n = e_n\}$: they are possibly empty finite sets of fields associating pairwise distinct labels to expressions. We use $e.\ell$ to denote (struct-like) field selection, $e\backslash\ell$ to denote (struct-like) field deletion, and $e_1 + e_2$ for record concatenation with priority given to the fields in e_2.

Let X be a set and $\perp$ a constant not in X, we use the notation $X_\perp$ to denote the set $X \cup \{\perp\}$ and the fact that $\perp \notin X$. In what follows, we use the constant $\perp$ to indicate that a field (in a record expression, in a record type, or in their interpretations) is undefined. In this sense the record expressions defined by the grammar above are just syntactic sugar to denote quasi-constant

[5] As a matter of fact, products and, more generally, tuples expressions and types can be encoded as record expressions and closed record types (see later on) whose domain is an initial segment of natural numbers.

functions in $\mathcal{L} \to \mathcal{E}_\perp$ whose default value is $\perp$, that is, $\{\ell_1 = e_1, \dots, \ell_n = e_n\}$ is syntactic sugar for $\{\!|\ell_1 = e_1, \dots, \ell_n = e_n, \_ = \perp|\!\}$. The reduction semantics is defined by the following rules

$$(\lambda^{\wedge_{i\in I} t_i \to t_i} x.e)v \rightsquigarrow e\{v/x\} \tag{3}$$
$$\{\ell_1 = v_1, \dots, \ell_n = v_n\}.\ell \rightsquigarrow v_i \qquad \text{if } \ell \equiv \ell_i \tag{4}$$
$$v_1 + v_2 \rightsquigarrow v_2 \oplus_\perp v_1 \tag{5}$$
$$v\backslash\ell \rightsquigarrow \{\!|\ell = \perp, \_ = c_\circ|\!\} \oplus_{c_\circ} v \tag{6}$$

where $c_\circ$ is a constant not in $\mathcal{E}_\perp$ (the choice of $c_\circ$ is not important as long as it is different from $\perp$), plus the rules for a leftmost-outermost weak reduction strategy, that is, $E[e] \rightsquigarrow E[e']$ if $e \rightsquigarrow e'$, where $E ::= [\,] \mid Ee \mid vE \mid E.\ell \mid E\backslash\ell \mid \{\ell_1 = v, ..., \ell_i = E, ..., \ell_n = e\}$.

The reduction in (3) is the classic call-by-value beta reduction. Selection is implemented by (4) and it is undefined if $\ell \notin \{\ell_1, ...\ell_n\}$. (5) and (6) use the merge operator $\oplus_z$ of Section 2.2.2, to define the reductum of concatenations and deletions, respectively. Since merge is defined only for quasi-constant functions, then the semantics is undefined if in (5) and (6) either v, or v_1, or v_2 is not a record value. The reductum of (5) is the record value formed by all the fields in v_2 plus all the fields in v_1 that are undefined in v_2. The reductum of (6) is the record value in which the field ℓ is undefined and all other fields are as in v.

3.2 Types

We consider two kinds of record types, open and closed, formed by two kinds of field types, mandatory and optional:

$$\textbf{Types} \quad t ::= c \mid b \mid t \to t \mid \{f, \dots, f\} \mid \{\!|f, \dots, f|\!\} \mid t \vee t \mid \neg t \mid \mathbb{0}$$
$$\textbf{Field types} \quad f ::= \ell = t \mid \ell \Rightarrow t$$

A field type maps a label into a type. A mandatory field type "$\ell = t$" means that a field for the label ℓ must be present and contain a value of type t. An optional field type "$\ell \Rightarrow t$" means that *if* a field for the label ℓ is present, then it must contain a value of type t (the syntax is inspired by Erlang, which uses `:=` and `=>` for mandatory and optional field types, respectively [Erlang, Section 7.2]).

Similarly to expressions, record types are just syntactic sugar for some specific quasi-constant functions in $\mathcal{L} \to \mathcal{T}_\perp$. Precisely, consider a record type with field types, f_1,...,f_n, where each f_i is either $\ell_i = t_i$ or $\ell_i \Rightarrow t_i$; then the *closed record type* $\{\!|f_1, ..., f_n|\!\}$ is syntactic sugar for the quasi-constant function $\{\!|\ell_1 = \tau_1, ..., \ell_n = \tau_n, \_ = \perp|\!\}$, while the *open record type* $\{f_1, ..., f_n\}$ is syntactic sugar for the quasi-constant function $\{\!|\ell_1 = \tau_1, ..., \ell_n = \tau_n, \_ = \mathbb{1}\vee\perp|\!\}$ where, in both cases, $\tau_i = t_i$ if f_i is mandatory, and $\tau_i = t_i \vee \perp$ if f_i is optional. In other words, a mandatory field $\ell = t$ states that the label ℓ is associated to a value of type t, while an optional field $\ell \Rightarrow t$ indicates that either the field for ℓ is undefined or it contains a value of type t; all the other fields are undefined if the record is closed, and are either undefined or contain some value (of any type) if the record is open.

Formally, we just added to our types a new type constructor with new atoms, the *record type atoms*, that are the quasi-constant functions from $\mathcal{L}$ to $\mathcal{T}_\perp$ whose default value is either $\perp$ or $\mathbb{1}\vee\perp$. In this section we use r to range over these atoms, that is, r will denote a quasi-constant function in $\mathcal{L}\to\mathcal{T}_\perp$ such that either $\mathrm{def}(r) = \perp$ (i.e., r is a closed record type atom) or $\mathrm{def}(r) = \mathbb{1}\vee\perp$ (i.e., r is an open record type); in Section 4 we will allow also the case $\mathrm{def}(r) = t\vee\perp$, for any type t.

3.2.1 Subtyping. It is easy to modify the definitions of Section 2.1 for the above types. Since records replace products, then the definition of the interpretation for products is replaced by $[\![r]\!] = \mathbb{\Pi}_{\ell\in\mathcal{L}} [\![r(\ell)]\!] \subseteq \mathcal{L} \to \mathcal{D}_\perp$ where r is a record type atom and $[\![\perp]\!] = \{\perp\}$.

The universal model for these types uses the interpretation domain $\mathcal{D} = C + \mathcal{L}\to\mathcal{D}_\perp + \mathcal{P}_{\mathrm{fin}}(\mathcal{D}\times\mathcal{D})$, and we interpret types as sets composed of elements of this domain inductively defined by:

$$d ::= c \mid \{\!|\ell = \partial, ..., \ell = \partial, \_ = \bot|\!\} \mid \{(d,d), ..., (d,d)\} \qquad \partial ::= d \mid \bot$$

The interpretation uses the binary predicate $(d : t)$ definition of Section 2.1.2 where we replace the clause for products with:

$$(\{\!|\ell_1 = \partial_1, ..., \ell_n = \partial_n, \_ = \bot|\!\} : r) \quad = \quad \big(\forall i{\in}[1..n]\,.\,(\partial_i : r(\ell_i)) \text{ or } (\partial_i = \bot \text{ and } \bot \leq r(\ell_i))\big) \text{ and } \big(\forall \ell \in \mathcal{L}\setminus\{\ell_1, ..., \ell_n\}\,.\,\bot \leq r(\ell)\big).$$

with $r \in \mathcal{L} \to \mathcal{T}_\bot$. As before every type is equivalent to a union of uniform intersections of atoms or their negation, and to decide the emptiness of intersections of record type atoms and their negations we use Lemma 2.3: just notice that for each atom r, since $\mathrm{def}(r)$ is either $\bot$ or $\mathbb{1}\vee\bot$, then $\bigwedge_{r\in P}\mathrm{def}(r)$ contains $\bot$ and, thus, is never empty (i.e., $\bigcap_{p\in P}\mathrm{def}(X_p) = \varnothing$ in the lemma's statement never holds).

3.2.2 Typing. Typing the functional part is standard. The algorithmic typing rules are:

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathrm{dom}(\Gamma)$$

$$[\to\textsc{I}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i' \leq t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \wedge_{i\in I}(s_i \to t_i)} \qquad [\to\textsc{E}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1 \circ t_2}\ \begin{array}{l} t_1 \leq \mathbb{0} \to \mathbb{1} \\ t_2 \leq \mathrm{dom}(t_1) \end{array}$$

where the operators $\mathrm{dom}()$ and $\circ$ are defined as $\mathrm{dom}(t) \stackrel{\text{def}}{=} \max\{u \mid t \leq u \to \mathbb{1}\}$ and $t\circ s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$. In short, $\mathrm{dom}(t)$ is the largest domain of any single arrow that subsumes t while $t \circ s$ is the smallest codomain of any single arrow that subsumes t and has domain s. These operators are needed because the type inferred for the function e_1 in [→E] may be different from a single arrow. In general, this type (if smaller than $\mathbb{0}\to\mathbb{1}$, that is, the type of all functions) will be a disjunctive normal form of arrow atoms for which computing the domain and the result of an application is not straightforward (see Castagna [2023, §4.1.2] for a simple explanation and Castagna [2020, §4.4.3 in the online extended version] for a detailed description on how to compute the $\circ$ operator). A similar problem happens for record operations since the types of the records involved in selection, concatenation, and deletion are, in general, disjunctive normal forms of record type atoms. To address this problem we define three type operators, one for each operation on records which gives:

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : t \leq \{\ell = \mathbb{1}\}}{\Gamma \vdash e.\ell : t.\ell} \qquad [\textsc{Del}]\ \frac{\Gamma \vdash e : t \leq \{\}}{\Gamma \vdash e\backslash\ell : t\backslash\ell} \qquad [\textsc{Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\} \qquad \Gamma \vdash e_2 : t_2 \leq \{\}}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}$$

Record operations require expressions to be typed by a subtype of the empty open record $\{\}$, which is the type of all records;[6] furthermore, selection requires the selected field to be defined (i.e., the type of e must be a subtype of $\{\ell = \mathbb{1}\}$ and, thus, cannot map ℓ to $\bot$). The selection type operator is defined as $t.\ell \stackrel{\text{def}}{=} \min\{u \mid t \leq \{\ell = u\}\}$; we show in Section 4 how to compute it. The concatenation and deletion operators are more complex. To type concatenation and deletion we need to type the merge operator of Section 2.2.2. It is easy to define it for record atoms: if $r_1, r_2 : \mathcal{L}\to\mathcal{T}_\bot$, we define

$$(r_1 +_t r_2)(\ell) = \begin{cases} r_2(\ell) & r_2(\ell) \wedge t \leq \mathbb{0} \\ (r_2(\ell) \setminus t) \vee r_1(\ell) & \text{otherwise} \end{cases} \tag{7}$$

It is then easy to see that, $r_1 + r_2$ is defined as $r_1 +_\bot r_2$: it returns the record type with all the fields surely defined in r_2 (i.e., those for which $r_2(\ell) \wedge \bot \leq \mathbb{0}$) and where the other fields have as type the union of the type of the field in r_1 (i.e., $r_1(\ell)$) and of the non-optional part (if any) of the field in r_2 (i.e., $r_2(\ell) \setminus \bot$). For instance, $\{|\, a = \mathsf{Int}, b = \mathsf{Bool}\,|\} + \{|\, a \Rightarrow \mathsf{Bool}\,|\} = \{|\, a = \mathsf{Int}\vee\mathsf{Bool}, b = \mathsf{Bool}\,|\}$ since a *may* be undefined on the right hand-side while b is undefined in it, and $\{|\, a = \mathsf{Int}\,|\} + \{\} = \{a = \mathbb{1}\}$

[6] We use the shorthand $\Gamma \vdash e : t \leq t'$ for a rule with premises $\Gamma \vdash e : t$ and side condition $t \leq t'$.

since a is surely defined though we cannot know whether its definition is taken from the right operand or left one. We postpone the definition of $t_1 \mathbin{+_t} t_2$ for generic (record) types to Section 4 where we generalize it for maps. With such a definition the concatenation type operator $t_1 + t_2$ will be defined as $t_1 \mathbin{+_\bot} t_2$ while the deletion type operator will be defined as $t \backslash \ell \stackrel{\text{def}}{=} t \mathbin{+_{c_\circ}} \{\!\![\ell = \bot, \_ = c_\circ]\!\!\}$ (with $c_\circ$ any constant different from $\bot$): that is, at type level, deletion sets the type of the field ℓ to $\bot$ and leaves the type of the other fields unchanged.

Finally, expressions defining records are typed by the corresponding closed record type:

$$[\textsc{Recd}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\!|\, \ell_1 = t_1, \ldots, \ell_n = t_n \,|\!\}}$$

3.2.3 Variations. We have seen in the introduction that there exists a lot of variations on the semantics of record operations. Since we cannot consider all of them, we just give an example of a common one. In our calculus, $e\backslash\ell$ works also when ℓ is not defined in e, but some languages (e.g., Elixir) require the field to be present for deletion. It is possible to encode such a discipline by a suitable pattern matching (see Section 3.3) or by directly modifying the static and dynamic semantics to do so. The modifications in the latter case are really minimal: it suffices to use for the typing rule [Del] the same premise as in rule [Sel].

Also, the generic $t_1 \mathbin{+_t} t_2$ can be used to type record operations other than concatenations and fields deletions. For instance, Frisch [2004, Section 9.5] shows that, given an expression e of record type t and a set of labels $L = \{\ell_1, \ldots, \ell_n\}$, the record obtained by restricting e to the fields in L is typed by $t \mathbin{+_{c_\circ}} \{\!\![\ell_1 = c_\circ, \ldots, \ell_n = c_\circ, \_ = \bot]\!\!\}$ and the record obtained by deleting in e all fields that are not of type t' has type $\{\!\![\_ = \bot]\!\!\} \mathbin{+_{\neg t'}} t$.

3.3 Pattern matching

The calculus defined in this section still misses an essential ingredient, that is, a way to test whether a record field is defined or not: without it, optional field types are useless. We could add an *ad-hoc* expression to perform this check, but we prefer to implement it via pattern matching since it is more general and it plays an important role for the definition of the types for mutable records (omitted in this presentation for space reasons and included for completeness in Appendix F: see in particular Section F.3). Thus, we add to our expressions a matching expression formed by one or more "$|$"-separated branches, each composed by a pattern p and an expression e that is executed when the pattern is matched:

$$\begin{aligned} \textbf{Expressions} \quad & e ::= \ldots \mid \texttt{match}\, e\, \texttt{with}\, p \rightarrow e \mid \ldots \mid p \rightarrow e \\ \textbf{Patterns} \quad & p ::= x \mid t \mid \{\ell = p\} \mid (x := c) \mid p \wedge p \mid p \mid p \end{aligned}$$

with the restriction that p_1 and p_2 must have distinct capture variables in $p_1 \wedge p_2$ and the same capture variables in $p_1 \mid p_2$.

The expression $\texttt{match}\, e\, \texttt{with}\, p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$ evaluates e to a value and matches it against the patterns p_i's in a left-to-right order. If the pattern p_i matches the value, then this produces a substitution for the capture variables in p_i that is applied to e_i before its evaluation. A pattern is either a capture variable x which matches any value and binds the value to x; or a type t, which matches any value of type t yielding an empty substitution; or the (open) record pattern $\{\ell = p\}$ that matches any record value containing *at least* a field ℓ with a value v that matches p (whose resulting substitution is returned); or the constant pattern $(x := c)$ which matches any value and binds x to the constant c; or the intersection pattern $p_1 \wedge p_2$ that matches values that match both p_1 and p_2 (and returns the union of the substitutions); or the union pattern $p_1 \mid p_2$ which matches values that match either p_1 or p_2 testing them in the order.

$$
\begin{array}{rcll}
v/x & = & \{v/x\} & \\
v/t & = & \{\,\} & \text{if } \vdash v : t \\
v/\{\ell = p\} & = & \sigma & \text{if } v = \{..., \ell = v', ...\} \text{ and } v'/p = \sigma \\
v/(x:=c) & = & \{c/x\} & \\
v/p_1 \wedge p_2 & = & \sigma_1 \cup \sigma_2 & \text{if } v/p_1 = \sigma_1 \text{ and } v/p_2 = \sigma_2 \\
v/p_1 \,|\, p_2 & = & v/p_1 & \text{if } v/p_1 \neq \text{fail} \\
v/p_1 \,|\, p_2 & = & v/p_2 & \text{if } v/p_1 = \text{fail} \\
v/p & = & \text{fail} & \text{otherwise}
\end{array}
$$

Fig. 1. Semantics of patterns

Formally, we define a function $(\cdot)/(\cdot)$ that, given a value v and a pattern p, yields a result v/p which is either fail or a substitution σ mapping the capture variables in p to values (subterms of v). This function is defined in Figure 1. Then, we augment the reduction rules with

$$\texttt{match}\, v\, \texttt{with}\, p_1 \to e_1 \,|\, \ldots \,|\, p_n \to e_n \;\rightsquigarrow\; e_i\sigma \qquad \text{if } v/p_i = \sigma \text{ and } v/p_j = \text{fail for } j < i$$

and add $E ::= \texttt{match}\, E\, \texttt{with}\, p_1 \to e_1 \,|\, \ldots \,|\, p_n \to e_n$ to the evaluation contexts. Our patterns are not standard, but they allow us to encode a large variety of different patterns. For instance, a more common syntax would be $p ::= \_ \mid c \mid x \mid \{\ell = p, \ldots, \ell = p\} \mid p \text{ as } x \mid p\,|\,p$, with wild-cards and constants instead of types, with as-patterns "p as x" (in OCaml syntax; x@p in Haskell) instead of conjunction, and with multi-field record patterns. We can encode _ and c as $\mathbb{1}$ and b_c; "p as x" is equivalent to $p \wedge x$; a multi-field (open) record pattern $\{\ell_1 = p_1, \ldots, \ell_n = p_n\}$ is encoded by the intersection $\{\ell_1 = p_1\} \wedge \ldots \wedge \{\ell_n = p_n\}$, while the *closed* record pattern $\{|\, \ell_1 = p_1, \ldots, \ell_n = p_n \,|\}$ (i.e., the pattern that matches records having *exactly* the fields $\ell_1, \ldots, \ell_n$ matching the respective patterns) can then be encoded as $\{\ell_1 = p_1, \ldots, \ell_n = p_n\} \wedge \{|\, \ell_1 = \mathbb{1}, \ldots, \ell_n = \mathbb{1} \,|\}$. Given any pattern p, we can define a type $\lceil p \rfloor$ that characterizes exactly the set of values that match the pattern:

$$
\begin{array}{ccc}
\lceil x \rfloor = \mathbb{1} & \lceil t \rfloor = t & \lceil p_1 \wedge p_2 \rfloor = \lceil p_1 \rfloor \wedge \lceil p_2 \rfloor \\
\lceil (x:=c) \rfloor = \mathbb{1} & \lceil \{\ell = p\} \rfloor = \{\ell = \lceil p \rfloor\} & \lceil p_1 \,|\, p_2 \rfloor = \lceil p_1 \rfloor \vee \lceil p_2 \rfloor
\end{array}
$$

It can be shown that, for every well-typed value v and every pattern p, we have $v/p \neq \text{fail}$ if and only if $\varnothing \vdash v : \lceil p \rfloor$. This allows us to formalize purely at the level of types, exhaustiveness and redundancy checks performed on pattern matching. The typing rule for match is the following.

$$
[\textsc{Match}]\ \frac{\Gamma \vdash e_0 : t_0 \qquad \text{for } i = 1..n, \text{ either } \ t_i \leq \mathbb{0} \ \text{ or } \ \Gamma, t_i/p_i \vdash e_i : t}{\Gamma \vdash \texttt{match}\, e_0\, \texttt{with}\, p_1 \to e_1 \,|\, \ldots \,|\, p_n \to e_n : t}
\quad
\begin{array}{l}
t_0 \leq \bigvee_{i=1}^{n} \lceil p_i \rfloor \\
t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor
\end{array}
$$

The rule deduces the type t_0 of the matched expression. The side condition $t_0 \leq \bigvee_{i=1}^{n} \lceil p_i \rfloor$ ensures that the matching is exhaustive, that is, that all the values that can be produced by e_0 (i.e., the values in t_0) are accepted by some pattern. Then it computes the type t_i that contains all the values that are captured by the i-th branch. These are the values that can be produced by e_0 (i.e., those in t_0) minus those that are captured by a preceding branch (i.e., those in $\bigvee_{j<i} \lceil p_j \rfloor$) and (intersected those) that match p_i (i.e., those in $\lceil p_i \rfloor$). When t_i is empty, then the branch is redundant (see Castagna [2023, §5.1] for details on the redundancy check) and the typing of e_i is skipped. Otherwise, e_i is typed under the hypothesis Γ extended with a type environment produced by t_i/p_i. Given a type t and a pattern p with $t \leq \lceil p \rfloor$, the operator t/p produces the type environment assumed for the variables in p when a value of type t is matched against p and the matching succeeds. It is defined as:

$$
\begin{array}{ccc}
t/t' = \varnothing & t/(x:=c) = x : b_c & p_1 \wedge p_2 = (t/p_1) \cup (t/p_2) \\
t/x = x : t & t/\{\ell = p\} = t.\ell/p & t/p_1 \,|\, p_2 = ((t \wedge \lceil p_1 \rfloor)/p_1) \cup ((t \setminus \lceil p_1 \rfloor)/p_2)
\end{array}
$$

and satisfies the property that for every t, p, and v, if $\varnothing \vdash v : t$ and $v/p = \sigma$, then, for every variable x in p, the judgment $\varnothing \vdash x\sigma : (t/p)(x)$ holds.

Thanks to pattern matching we can now check whether an optional field with label ℓ is present before selecting it, as simply as "$\texttt{match}\, e\, \texttt{with}\, \{\ell = \mathbb{1}\} \rightarrow \ldots$". Also, we can capture its content in a capture variable x and in case of absence give x a default value, say, `nil` by using the constant pattern "$\texttt{match}\, e\, \texttt{with}\, \{\ell = x\} \,|\, (x\texttt{:=nil}) \rightarrow \ldots$". Also notice that the expression $e.\ell$ for selection is no longer needed (though, we keep it for convenience) since it can be encoded as "$\texttt{match}\, e\, \texttt{with}\, \{\ell = x\} \rightarrow x$": for this expression the rule [Match] requires the type t of e to be a subtype of $\{\ell = \mathbb{1}\}$ (exhaustiveness condition) and deduces for the expression the type $t.\ell$, which coincides with the rule [Sel]. Likewise, to define the delete operation that requires the presence of the field ℓ to be deleted, we no longer need to modify the typing rules as we outlined in Section 3.2.3: it suffices to encode it as "$\texttt{match}\, e\, \texttt{with}\, \{\ell = \mathbb{1}\} \rightarrow e\backslash\ell$" which returns a static type error if ℓ is not present in e.

The type system presented in this section is sound: every well-typed expression either diverges or returns a value of the same type (see [Frisch 2004]). It describes records (and, partially, pattern matching) as they are currently implemented in the language CDuce.

4 UNIFYING STRUCTS WITH MAPS

Extending the previous calculus to handle maps, is conceptually simple. First, labels must be computable, that is, it must be possible to obtain them as results of expressions: therefore $\mathcal{L}$, the set of all labels, must be a subset of $\mathcal{V}$, the set of all values of the language. Second, we must add expressions to delete and select labels that are computed by expressions: therefore we supplement the expressions $e.\ell$ and $e\backslash\ell$ for structs with two new expressions $e.[e]$ and $e\backslash[e]$ that are map oriented. Third, since record concatenations of Section 3.1 cannot compute labels, then we introduce the expression $e_1{\leftarrow}\langle[e_2]{=}e_3\rangle$ that adds (or updates) the field with label computed by e_2 and content computed by e_3 to the map computed by e_1. Finally, we allow record types—which are quasi-constant functions in $\mathcal{L}\to\mathcal{T}_\perp$—to specify their default value: we add to field types the *default field* "$\_ \Rightarrow t$". The rest remains unchanged:

$$\begin{array}{lll} \textbf{Expressions} & e ::= & \cdots \mid e.[e] \mid e\backslash[e] \mid e{\leftarrow}\langle[e]{=}e\rangle \\ \textbf{Field Types} & f ::= & \cdots \mid \_ \Rightarrow t \end{array}$$

The semantics of new expressions of map-selection and map-deletion is defined as follows:

$$\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.[\ell] \rightsquigarrow \begin{cases} v_i & \text{if } \ell \equiv \ell_i \text{ for some } i \in [1..n] \\ \texttt{nil} & \text{otherwise} \end{cases} \tag{8}$$

$$v\backslash[\ell] \rightsquigarrow \{\!|\ell = \bot, \_ = c_\circ|\!\}\oplus_{c_\circ} v \tag{9}$$

$$v{\leftarrow}\langle[\ell]{=}e\rangle \rightsquigarrow v + \{\ell = e\} \tag{10}$$

where `nil` is a distinguished constant returned when the selected field is undefined (e.g., Ballerina, Elixir, and Lua use `nil`, JavaScript uses `undefined`, Scala uses `None`, ...) and the evaluation contexts are updated by adding the productions $E ::= E.[e] \mid v.[E] \mid E\backslash[e] \mid v\backslash[E] \mid E{\leftarrow}\langle[e]{=}e\rangle \mid v{\leftarrow}\langle[E]{=}e\rangle$.

The default field "$\_ \Rightarrow t$" defines the type of the fields that are not already specified in a record type: for example, $\{\ell_1 \Rightarrow t_1, \ell_2 = t_2, \_ \Rightarrow t_3\}$ specifies that every label different from ℓ_1 and ℓ_2 is mapped to t_3. Note that for the default field we used the syntax "$\Rightarrow$" of optional fields since the default fields always contain at least $\bot$. Thus, $\{\ell_1 \Rightarrow t_1, \ell_2 = t_2, \_ \Rightarrow t_3\}$ denotes the quasi-constant function $\{\!|\ell_1 = t_1 \vee \bot, \ell_2 = t_2, \_ = t_3 \vee \bot|\!\}$. Observe that we no longer need to differentiate between open and closed record types, since this difference can be expressed by specifying either "$\_ \Rightarrow \mathbb{0}$" (for a closed record type) or "$\_ \Rightarrow \mathbb{1}$" (for an open one) for the default field. As for label selection (which can be encoded by matching), we keep for convenience the syntax of both record types (but

use mainly open ones), although this is a source of redundancy since $\{f_1, ..., f_n\} = \{f_1, \dots, f_n, \_ \Rightarrow \mathbb{1}\} = \{|\, f_1, \dots, f_n, \_ \Rightarrow \mathbb{1} \,|\}$ and $\{|\, f_1, \dots, f_n \,|\} = \{|\, f_1, \dots, f_n, \_ \Rightarrow \mathbb{0} \,|\} = \{f_1, \dots, f_n, \_ \Rightarrow \mathbb{0}\}$.

With this new syntax we can now express the type of maps from labels to, say, integers as $\{\_ \Rightarrow \mathsf{Int}\}$ (or, equivalently, $\{|\_ \Rightarrow \mathsf{Int}|\}$). This syntax covers the simple case of maps whose domain is always the whole set of labels. This is compatible with what happens for JavaScript objects where the set of labels (i.e., the properties names) is the set of all strings. Likewise, in Ballerina the type `map<T>` is the set of mappings from keys of type `string` to values of type `T`: we show later in Section 4.4 how to modify the calculus to refine domains. The new syntax also allows us to combine structs with maps in the same type, as we showed by the example $\{\ell_1 \Rightarrow t_1, \ell_2 = t_2, \_ \Rightarrow t_3\}$. Thus, for instance, it is easy to express by a type the fact that a specific label is used to determine the kind of the map, as in $\{output = \texttt{"int"}, \_ \Rightarrow \mathsf{Int}\} \vee \{output = \texttt{"str"}, \_ \Rightarrow \mathsf{String}\}$.

The interpretation and the subtyping relation for the new types are still the ones given in Section 3.2.1: the only difference is that given a record atom $r \in \mathcal{L} \to \mathcal{T}_\perp$, with $\mathsf{def}(r) = t \vee \perp$ now t can be any type while, before, t could be only $\mathbb{0}$ or $\mathbb{1}$ (from now on r will range on these newer atoms). Typing instead needs important modifications. We require not only that $\mathcal{L} \subseteq \mathcal{V}$ but also that $\mathcal{L}$ can be expressed as a type. For simplicity, we suppose $\mathcal{L}$ to be a basic type, but equivalently we might imagine $\mathcal{L}$ to be some type expression (e.g., in Ballerina and JavaScript $\mathcal{L} = \texttt{string}$, in Go $\mathcal{L} = \mathbb{1} \setminus (\{\} \vee \texttt{slice} \vee (\mathbb{0} \to \mathbb{1}))$, in Lua $\mathcal{L} = \mathbb{1} \setminus (\texttt{nil} \vee \texttt{NaN})$, in Elixir $\mathcal{L} = \texttt{atom}$ for struct-like records and, as in Erlang, $\mathcal{L} = \mathbb{1}$ for maps)[7]. The typing rules for the new selection and delete expressions are similar to the corresponding ones for structs:

$$\text{[M-Sel]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathcal{L}}{\Gamma \vdash e_1.[e_2] : t_1.[t_2]} \qquad \text{[M-Del]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathcal{L}}{\Gamma \vdash e_1 \backslash [e_2] : t_1 \backslash [t_2]}$$

with the notable difference that they use new type operators for selection and deletion that are defined for sets of labels rather than single ones. Likewise, map update uses a specific type operator:

$$\text{[M-Upd]}\ \frac{\Gamma \vdash e_1 : t_1 \leq \{\} \qquad \Gamma \vdash e_2 : t_2 \leq \mathcal{L} \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1 \leftarrow \langle [e_2] = e_3 \rangle : t_1 \leftarrow \langle [t_2] = t_3 \rangle}$$

We define all these new type operators in Section 4.2, but first we need to give a suitable representation of our record types.

4.1 Representation of record types.

All record types (i.e., all subtypes of $\{\}$) can be represented as disjunctive normal forms of *record type atoms* (see §3.2). This representation is used when deciding subtyping on record types (cf. Lemma 2.3). However, to compute the type operators used by the typing rules for record selection, deletion, concatenation, and update, disjunctive normal forms are inappropriate because of the presence of negated record type atoms in them. It is thus convenient to use a different representation based on a generalization of the record type atoms of Section 3.2: in this representation, negation is directly integrated in the atoms. To that end we use and adapt some results by Frisch [2004].

Let τ range over $\mathcal{T}_\perp$—i.e., the "types" of field contents—while t still ranges over $\mathcal{T}$. When working with the type of the fields of a record type, all set-theoretic operations are relative to $\mathcal{T}_\perp$. In particular, the complement of τ —denoted by $\texttt{not}(\tau)$ to distinguish it from type negation "$\neg$"— is performed in $\mathcal{T}_\perp$: for instance $\texttt{not}(\mathbb{0}) = \mathbb{1} \vee \perp$, $\texttt{not}(t) = \neg t \vee \perp$, and $\texttt{not}(\perp) = \mathbb{1}$.

Definition 4.1 (Frisch [2004]). *Let $L = \{\ell_1, \dots, \ell_n\} \subset \mathcal{L}$ be a finite set of labels. If $(\tau_\ell)_{\ell \in L}$ is a family of types indexed on L, S is a finite set of types, and t_0 is a type, we use the notation*

[7] In Go *slices* are descriptors for contiguous segments of arrays; in Elixir *atoms* are colon-prefixed user-defined constants.

$\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$ to denote the type $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, \_ \Rightarrow t_0\} \setminus \bigvee_{s\in S}\{\ell_1 \Rightarrow \mathbb{1}, \ldots, \ell_n \Rightarrow \mathbb{1}, \_ \Rightarrow \neg s\}$ (where, with an abuse of notation, the field "$\ell = t \vee \bot$" stands for the field "$\ell \Rightarrow t$").

In practice, every type s in S adds to the type $\{\ell_1 = \tau_{\ell_1}, \ldots, \ell_n = \tau_{\ell_n}, \_ \Rightarrow t_0\}$ a constraint interpreted as "... and there exists a label not in L whose value is of type s". The record types of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$ are the building blocks to represent all record types, that is all subtypes of $\{\}$, as a single union, since they satisfy the following properties (cf. Frisch [2004, §9.1.4]):

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \not\simeq \mathbb{0} \Leftrightarrow \forall \ell \in L, \tau_\ell \not\simeq \mathbb{0} \text{ and } \forall s \in S, t_0 \wedge s \not\simeq \mathbb{0} \tag{11}$$

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \wedge \langle\!\langle (\tau'_\ell)_{\ell\in L} \;;\; t'_0 \;;\; S' \rangle\!\rangle = \langle\!\langle ((\tau_\ell \wedge \tau'_\ell)_{\ell\in L} \;;\; t_0 \wedge t'_0 \;;\; S \cup S' \rangle\!\rangle \tag{12}$$

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \setminus \langle\!\langle (\tau'_\ell)_{\ell\in L} \;;\; t'_0 \;;\; \emptyset \rangle\!\rangle = \langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \cup \{\neg t'_0\} \rangle\!\rangle \vee \bigvee_{\ell_0\in L} \langle\!\langle (\tau_\ell^{\ell_0})_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \tag{13}$$

where $\tau_\ell^{\ell_0}$ is defined as $\tau_\ell \wedge \mathtt{not}(\tau'_\ell)$ if $\ell = \ell_0$ and τ_ℓ otherwise.

Using the last two properties it is not difficult to transform every record type in disjunctive normal form into a union of records of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$:

Theorem 4.2 (Frisch [2004]). *Let $L \in \mathcal{P}_{\text{fin}}(\mathcal{L})$ a finite set of labels and t a type such that $t \leq \{\}$ and $\mathrm{dom}(t) \subseteq L$. Then there exists a finite set $\pi^L_{\mathbf{rec}}(t)$ of types of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$ such that*

$$t \simeq \bigvee_{\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \in \pi^L_{\mathbf{rec}}(t)} \langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$$

Proof. If $t \leq \{\}$, then it is equivalent to $\bigvee_{i\in I}(\bigwedge_{p\in P_i} r_p \wedge \bigwedge_{n\in N_i} \neg r_n)$ where the r's are record type atoms, that is, record types of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; \emptyset \rangle\!\rangle$. By applying (12) we can transform $\bigwedge_{p\in P_i} r_p$ into a record of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$. By successively applying (13) to the intersection of the record thus obtained and each record in $\bigwedge_{n\in N_i} \neg r_n$ we obtain the result. □

Hereafter we use R to range over records of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; t_0 \;;\; S \rangle\!\rangle$ and consider them our new record type atoms (of which our previous atoms—ranged over by r—are just the special case for $S = \emptyset$). They can be used for the internal representation of record types since, as per Theorem 4.2, every subtype of $\{\}$ is equivalent to a union of the form $\bigvee_{i\in I} \mathsf{R}_i$: this property yields definitions for type operators that are simpler than with disjunctive normal forms, as we show next.

4.2 Type operators

Theorem 4.2 allows us to formally define and compute the type operators we used in the typing rules. Let us start with $t.\ell$, the projection of a record type t on a single label ℓ.

Struct-selection. Let ℓ be a label in $\mathcal{L}$, t be a type such that $t \leq \{\}$, and L be a finite set of labels such that $\mathrm{dom}(t) \subseteq L$. Define $\boldsymbol{\pi}_\ell(t) = \min\{\tau \mid t \leq \{\ell = \tau\}\}$, that is, the projection on ℓ of t. Then, it is clear that we have

$$\boldsymbol{\pi}_\ell(t) = \begin{cases} \bigvee_{\langle\!\langle (\tau_m)_{m\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \in \pi^L_{\mathbf{rec}}(t)} \tau_\ell & \text{if } \ell \in L \\ \bigvee_{\langle\!\langle (\tau_m)_{m\in L} \;;\; t_0 \;;\; S \rangle\!\rangle \in \pi^L_{\mathbf{rec}}(t)} t_0 \vee \bot & \text{if } \ell \notin L \end{cases} \tag{14}$$

It is straightforward to see that the $t.\ell$ operator we used in the rule [Sel] is defined as $\boldsymbol{\pi}_\ell(t)$ if $\boldsymbol{\pi}_\ell(t) \wedge \bot \simeq \mathbb{0}$ (i.e., if $\boldsymbol{\pi}_\ell(t)$ is a type, viz., if the field ℓ is defined) and is undefined otherwise, and to compute it, it suffices to take $L = \mathrm{dom}(t)$.

Map-selection. Regarding the map-selection operator $t.[t']$ used in [M-sel], let $t \leq \{\}$ and $t' \leq \mathcal{L}$ and consider the following union $\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t)$. Notice that even if t' may be an infinite set of labels (e.g., `string`), it is always possible to express this union as a finite union. This results from two properties, namely that $\mathrm{dom}(t)$ is finite, and that for all $\ell_1, \ell_2 \in t' \setminus \mathrm{dom}(t)$, we have $\boldsymbol{\pi}_{\ell_1}(t) = \boldsymbol{\pi}_{\ell_2}(t)$. Therefore, $\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t)$ is equivalent to the finite union $\boldsymbol{\pi}_{\bar\ell}(t) \vee \bigvee_{\ell\in \mathrm{dom}(t)\wedge t'} \boldsymbol{\pi}_\ell(t)$ where $\bar\ell$ is any

label in $t' \setminus \mathrm{dom}(t)$, if any.[8] Then $t.[t']$ is defined as $\bigvee_{\ell \in t'} \boldsymbol{\pi}_\ell(t)$ if $\bot \wedge \bigvee_{\ell \in t'} \boldsymbol{\pi}_\ell(t) \simeq \mathbb{0}$ and as $\mathtt{nil} \vee (\bigvee_{\ell \in t'} \boldsymbol{\pi}_\ell(t) \setminus \bot)$ otherwise. Also, we may want the type-checker to emit a warning when $\bigvee_{\ell \in t'} \boldsymbol{\pi}_\ell(t) \leq \bot$ (i.e., when the selection will surely yield $\mathtt{nil}$), as well as when $\mathbb{1} \leq \bigvee_{\ell \in t'} \boldsymbol{\pi}_\ell(t)$ (i.e., when the selection may spill over the "open" part of the record type t, whose type is $\mathbb{1} \vee \bot$), since both are situations in which, even though the expressions are well-typed (i.e., they cannot produce stuck expressions at run-time) they may conceal some problem (selecting a field that is surely undefined does not make much sense).

Map-deletion. Regarding the map-deletion operator $t \setminus [t']$ used in [M-Del], it just adds $\bot$ to all the fields whose label is in t', since they all *may* become undefined (though, just one of them will actually become undefined: the one for the label computed in the delete expression), that is: let $t \leq \{\!\}$ with $\mathrm{dom}(t) \subseteq L$ and $t' \leq \mathcal{L}$, then $t \setminus [t'] = \bigvee_{\langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle \in \pi^L_{\mathbf{rec}}(t)} \langle\!\langle (\tau'_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ where τ'_ℓ is $\tau_\ell \vee \bot$ if $\ell \in L \wedge t'$ and is τ_ℓ otherwise.

Map-update. The map-update operator $t \leftarrow \langle [t_1] = t_2 \rangle$ is very similar to the map-deletion one, but it uses t_2 instead of $\bot$ to update the fields (and the default t_0 types), that is: let $t \leq \{\!\}$ with $\mathrm{dom}(t) \subseteq L$ and $t_1 \leq \mathcal{L}$, then $t \leftarrow \langle [t_1] = t_2 \rangle = \bigvee_{\langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle \in \pi^L_{\mathbf{rec}}(t)} \langle\!\langle (\tau'_\ell)_{\ell \in L} \,;\, t'_0 \,;\, S \rangle\!\rangle$ where τ'_ℓ is $\tau_\ell \vee t_2$ if $\ell \in L \wedge t_1$ and is τ_ℓ otherwise, and t'_0 is $t_0 \vee t_2$ if $t_1 \setminus L \not\simeq \mathbb{0}$ and t_0 otherwise.[9]

Record concatenation (and struct-deletion). Finally, it remains to define the $t_1 \mathbin{+_t} t_2$ operator where t is any type and t_1 and t_2 are non-empty subtypes of $\{\!\}$ (the operator is undefined if t_1 or t_2 is not a record type). This operator defines both the record type concatenation operator $t_1 + t_2$ used in [Conc] (defined as $t_1 \mathbin{+_\bot} t_2$), and the struct deletion operator $t \setminus \ell$ (defined as $t \mathbin{+_{c_\circ}} \langle\!\langle \{\bot_\ell\} \,;\, c_0 \,;\, \emptyset \rangle\!\rangle$). Equation (7) defines the concatenation for two atoms, which in the new representation are two atoms of the form $\langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, \emptyset \rangle\!\rangle$. In that specific case the operator is *exact*, in the sense that it yields the type formed by all record values obtained by the concatenation of records of the first type with records of the second type. Frisch [2004, Theorem 9.6] proves that, in general, it is not possible to give an exact definition for the operator, the problem happening, of course, with types $\langle\!\langle (\tau_\ell)_{\ell \in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ in which S is not empty. However, it is possible to define a sound approximation that is operationally indistinguishable from the exact definition [Frisch 2004, Lemmas 9.12–9.17]: just disregard S sets.[10] Formally, this is obtained by defining the $+_t$ operator on two atoms in the new representation as follows:

$$\langle\!\langle (\tau^1_\ell)_{\ell \in L} \,;\, t_1 \,;\, S_1 \rangle\!\rangle \mathbin{+_t} \langle\!\langle (\tau^2_\ell)_{\ell \in L} \,;\, t_2 \,;\, S_2 \rangle\!\rangle = \langle\!\langle (\tau^3_\ell)_{\ell \in L} \,;\, t_3 \,;\, \emptyset \rangle\!\rangle \tag{15}$$

where for all $\ell \in L$, τ^3_ℓ is τ^2_ℓ if $\tau^2_\ell \wedge t \leq \mathbb{0}$ and is $(\tau^2_\ell \setminus t) \vee \tau^1_\ell$ otherwise and, likewise, t_3 is t_2 if $t_2 \wedge t \leq \mathbb{0}$ and is $(t_2 \setminus t) \vee t_1$ otherwise (notice that S_1 and S_2 were simply discarded). Finally, for the concatenation of two record types, since each type is equivalent to a union of record atoms, this is given by the formula

$$\Big(\bigvee_{i \in I} \mathsf{R}_i\Big) \mathbin{+_t} \Big(\bigvee_{j \in J} \mathsf{R}_j\Big) = \bigvee_{i \in I, j \in J} (\mathsf{R}_i \mathbin{+_t} \mathsf{R}_j) \tag{16}$$

Remark 4.3 (Internal representation of records atoms). *Transforming any subtype of $\{\!\}$ into a union of records by applying the proof of Theorem 4.2 to the disjunctive normal form of the type is easy. The formulæ above, then, provide an effective definition of the type operators. In the*

[8] Here, and in what follows we use the ambiguity that a subtype of $\mathcal{L}$ denotes a set of labels and, thus, we mix set-theoretic (e.g., $\in$) with type-theoretic (e.g., $\wedge$) notations, to lighten the presentation.

[9] For a more precise typing, both map-deletion and map-update (or their corresponding typing rules) should be specialized to work as the corresponding struct cases, when the type of the expression computing the label is a singleton type.

[10] The intuition for this property is that, as the definition of $\boldsymbol{\pi}_\ell(t)$ shows, the S constraints do not give any information about the value obtained from a selection, which is the only possible observation we can perform on records.

language of Section 3 the exact content of S for a type $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ *is not needed: what is needed is just to know whether S is empty or not in order to apply (11) (since* t_0 *is either* $\bot$ *or* $\mathbb{1}\vee\bot$*, then for a non-empty type, by (13), the only possible non-empty type s in S is* $\mathbb{1}$*). So the internal representation of record type atoms can use just a simple Boolean to flag if S is non-empty (and another for* t_0*) and the type pretty printer will display in the list of fields "*`(+ others)`*" for the types in which this flag is on. This is no longer true for maps since the full information of S is needed to check that, say,* $\{|\_ \Rightarrow \mathsf{Int}|\} \setminus \{|\_ \Rightarrow \mathsf{Int} \vee \mathsf{Bool}|\} = \langle\!\langle \{\} \,;\, \mathsf{Int}\vee\bot \,;\, \{\neg\mathsf{Int}\wedge\neg\mathsf{Bool})\} \rangle\!\rangle$ *is empty.*

The system presented in this section subsumes the one of Section 3 and preserves its meta-theoretic properties. In particular, soundness of the type system can be proved by a routine extension of the inductive proof of the soundness of the system of Section 3 (see Appendix B).

4.3 Implementation of the Subtyping Algorithm

The formulas in Section 4.2 give an effective definition of the type operators used in the typing rules. To complete the description of how typing can be implemented, it still remains to describe how to implement subtyping for our new record types. Lemma 2.3 describes the formal decomposition one has to perform to check subtyping for a disjunctive normal form of records (i.e., a union of intersections of record atoms or their negations). But in order to implement this check we have to define for the formula in the lemma's statement, a (backtrack free) Φ function that is analogous to the Φ function we gave in Section 2.1.4 for the product decomposition formula (2).

Since the implementation of the subtyping algorithm manipulates directly types in disjunctive normal form (see Castagna [2020, §4.3]) we define Φ to work on them (the transformation into a union of record atoms described by Theorem 4.2 must be applied only to compute record type operators). More precisely, we need Φ to decide whether $\bigwedge_{\mathsf{R}\in P}\mathsf{R}\wedge\bigwedge_{\mathsf{R}\in N}\neg\mathsf{R}$ is empty, where the R's in $P\cup N$ are of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t \,;\, \emptyset \rangle\!\rangle$. Thanks to (12), we can move the first intersection inside the records and consider it as a unique record $\mathsf{R}_\circ$. Then we have

$$\mathsf{R}_\circ \;\leq\; \bigvee_{\mathsf{R}\in N}\mathsf{R} \qquad\Longleftrightarrow\qquad (\mathsf{R}_\circ \leq \mathbb{0}) \;\text{ or }\; \Phi(\mathsf{R}_\circ, N) \tag{17}$$

with $\Phi(\mathsf{R}_\circ, N)$ defined as:

$$\begin{array}{rcl} \Phi(\mathsf{R}_\circ, \emptyset) & = & \texttt{false} \\ \Phi(\mathsf{R}_\circ, N\cup\{\mathsf{R}\}) & = & (\ \mathsf{def}(\mathsf{R}_\circ) \;\leq\; \mathsf{def}(\mathsf{R}) \text{ and} \\ & & \quad \forall \ell\in L\,.\,(\mathsf{R}_\circ(\ell)\leq\mathsf{R}(\ell) \;\text{ or }\; \Phi(\mathsf{R}_\circ\wedge\{\!|\ell : \mathtt{not}(\mathsf{R}(\ell))|\!\}, N)) \\ & &)\ \text{or}\ (\ \mathsf{def}(\mathsf{R}_\circ) \not\leq \mathsf{def}(\mathsf{R}) \text{ and } \Phi(\mathsf{R}_\circ, N)\ ) \end{array} \tag{18}$$

where $L = \bigcup_{\mathsf{R}\in N\cup\{\mathsf{R}_\circ\}}\mathsf{dom}(\mathsf{R})$.

Notice that in the second clause of (18) the two propositions that form the "or" are mutually exclusive. Therefore, this clause can be equivalently expressed in a programming-oriented way as

$$\begin{array}{rcl} \Phi(\mathsf{R}_\circ, N\cup\{\mathsf{R}\}) & = & \texttt{if}\ \ \mathsf{def}(\mathsf{R}_\circ) \;\leq\; \mathsf{def}(\mathsf{R}) \\ & & \quad \texttt{then}\ \ \forall \ell\in L\,.\,(\ \mathsf{R}_\circ(\ell)\leq\mathsf{R}(\ell) \;\text{ or }\; \Phi(\mathsf{R}_\circ\wedge\{\!|\ell : \mathtt{not}(\mathsf{R}(\ell))|\!\}, N)\ ) \\ & & \quad \texttt{else}\ \ \Phi(\mathsf{R}_\circ, N) \end{array} \tag{19}$$

This definition of Φ works both in the case we have structs (i.e., when $\mathsf{def}(\mathsf{R})$ is either $\mathbb{1}\vee\bot$ or $\bot$) and more generally in the case for maps (i.e., when $\mathsf{def}(\mathsf{R})$ is $t\vee\bot$ for some type t).[11]

To prove the correctness of (17) we use the representation and properties introduced in the previous section. In particular, by property (13) it is easy to prove the following lemma

[11] Just a caveat. The intersection of two "pure" maps—e.g. $\{\!|\_ \Rightarrow \mathsf{Int}|\!\}$ and $\{\!|\_ \Rightarrow \mathsf{String}|\!\}$—is never empty since it always contains at least the empty record expression $\{\}$.

Lemma 4.4. *Let $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \not\simeq \mathbb{0}$ and N be a set of types of the form $\langle\!\langle (\tau'_\ell)_{\ell\in L} \,;\, t'_0 \,;\, \emptyset \rangle\!\rangle$, then*

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \bigvee_{t\in N} t \quad \Longleftrightarrow \quad \langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, \emptyset \rangle\!\rangle \leq \bigvee_{t\in N} t$$

Proof. Since $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, \emptyset \rangle\!\rangle$, then the right to left implication is obvious. The converse implication is proved by induction on the cardinality of N. The case for $|N| = 0$ is immediate since both containment relations are false. The inductive case follows from the application of property (13) and the induction hypothesis.

The previous lemma plays a key role to prove the correctness of (17):

Theorem 4.5. *Let N be a set of types of the form $\langle\!\langle (\tau'_\ell)_{\ell\in L} \,;\, t'_0 \,;\, \emptyset \rangle\!\rangle$. Then*

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \bigvee_{t\in N} t \quad \Longleftrightarrow \quad (\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \mathbb{0}) \text{ or } \Phi(\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle, N)$$

Proof. If $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ is empty then the result holds. Otherwise, we proceed by induction on the size of N.

If N is empty then $\Phi(\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle, N) = \text{false}$ and $\bigvee_{t\in N} t \leq \mathbb{0}$. Therefore, as stated by definition (18), the left-hand side of the statement holds if and only if $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \mathbb{0}$

If $N = N' \cup \{\langle\!\langle (\tau'_\ell)_{\ell\in L} \,;\, t'_0 \,;\, \emptyset \rangle\!\rangle\}$, then we apply the definition (18) where $\mathsf{R}_\circ = \langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ and $\mathsf{R} = \langle\!\langle (\tau'_\ell)_{\ell\in L} \,;\, t'_0 \,;\, \emptyset \rangle\!\rangle$. By (13) the left-hand side of the theorem's statement is equivalent to

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \cup \{\neg t'_0\} \rangle\!\rangle \vee \bigvee_{\ell_0\in L} \langle\!\langle (\tau^{\ell_0}_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \quad \leq \quad \bigvee_{t\in N'} t \tag{20}$$

where for each $\ell_0 \in L$ the type $\langle\!\langle (\tau^{\ell_0}_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ is equivalent to $\mathsf{R}_\circ \wedge \{\!\{\ell_0 : \mathtt{not}(\mathsf{R}(\ell_0))\}\!\}$, with R and $\mathsf{R}_\circ$ defined as just above formula (20).

A union of types is smaller than a type if and only if every summand of the union is smaller than that type, so we are going to prove it by applying the induction hypothesis on N'.

Let us examine the first summand of the left-hand side of (20) and check whether

$$\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \cup \{\neg t'_0\} \rangle\!\rangle \quad \leq \quad \bigvee_{t\in N'} t$$

By induction hypothesis this holds if and only if either (i) $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \cup \{\neg t'_0\} \rangle\!\rangle$ is empty or (ii) $\Phi(\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \cup \{\neg t'_0\} \rangle\!\rangle, N')$. For the (i) case, since $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ is not empty, then by (11) the type is empty only if $t_0 \wedge \neg t'_0$ is empty, that is $\mathsf{def}(\mathsf{R}_\circ) \leq \mathsf{def}(\mathsf{R})$. If $\mathsf{def}(\mathsf{R}_\circ) \not\leq \mathsf{def}(\mathsf{R})$, then we are in case (ii) and have to check $\Phi(\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \cup \{\neg t'_0\} \rangle\!\rangle, N')$ which by Lemma 4.4 is equivalent to $\Phi(\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, \emptyset \rangle\!\rangle, N')$ and thus to $\Phi(\mathsf{R}_\circ, N')$.

For the other summands we have to prove for every $\ell_0 \in L$ that

$$\langle\!\langle (\tau^{\ell_0}_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle \leq \quad \bigvee_{t\in N'} t$$

By induction hypothesis this is equivalent to proving that either (i) the left-hand side is empty, that is, that $\tau^{\ell_0}_{\ell_0}$ is empty, that is—by the definition of $\tau^{\ell_0}_{\ell_0}$—, $\mathsf{R}_\circ(\ell_0) \leq \mathsf{R}(\ell_0)$, or (ii) proving the truth of $\Phi(\langle\!\langle (\tau^{\ell_0}_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle, N')$, that is, of $\Phi(\mathsf{R}_\circ \wedge \{\!\{\ell_0 : \mathtt{not}(\mathsf{R}(\ell_0))\}\!\}, N')$

Summing up, we have proved by induction that if $\langle\!\langle (\tau_\ell)_{\ell\in L} \,;\, t_0 \,;\, S \rangle\!\rangle$ is not empty then (20) is equivalent to checking both these two proposition disjunctions:

(1) $(\mathsf{def}(\mathsf{R}_\circ) \leq \mathsf{def}(\mathsf{R}))$ or $(\mathsf{def}(\mathsf{R}_\circ) \not\leq \mathsf{def}(\mathsf{R})$ and $\Phi(\mathsf{R}_\circ, N'))$
(2) $\forall \ell_0 \in L \,.\, (\, (\mathsf{R}_\circ(\ell_0) \leq \mathsf{R}(\ell_0))$ or $\Phi(\mathsf{R}_\circ \wedge \{\!\{\ell_0 : \mathtt{not}(\mathsf{R}(\ell_0))\}\!\}, N')\,)$

To conclude notice that if ($\mathsf{def}(\mathsf{R}_\circ) \not\leq \mathsf{def}(\mathsf{R})$ and $\Phi(\mathsf{R}_\circ, N')$) holds, then by induction hypothesis we have $\mathsf{R}_\circ \leq \bigvee_{t\in N'} t$ and, *a fortiori*, $\mathsf{R}_\circ \leq \bigvee_{t\in N'\cup\{\mathsf{R}\}} t$. It is therefore useless to check the other proposition (2) above, which thus must be checked only when $\mathsf{def}(\mathsf{R}_\circ) \leq \mathsf{def}(\mathsf{R})$. This yields

$$\begin{array}{l}(\ \mathsf{def}(\mathsf{R}_\circ)\ \not\leq\ \mathsf{def}(\mathsf{R})\ \text{and}\ \Phi(\mathsf{R}_\circ, N')\ )\\ \text{or}\ (\ \mathsf{def}(\mathsf{R}_\circ)\ \leq\ \mathsf{def}(\mathsf{R})\ \text{and}\ (\forall \ell_0 \in L\,.\,(\mathsf{R}_\circ(\ell_0) \leq \mathsf{R}(\ell_0)\ \text{ or }\ \Phi(\mathsf{R}_\circ \wedge \{\!|\ell_0 : \mathtt{not}(\mathsf{R}(\ell_0))|\!\}, N'))\end{array}$$

that is (modulo renaming) the second clause in (18), which proves the theorem. □

4.4 Erlang-style maps

In the first part of this section we defined a very simple case of maps whose domain was always the set of all labels. This suits languages in which record keys are of a specific and limited type (e.g., strings for Ballerina's maps and JavaScript's objects, integers and strings for PHP). However, there exist more refined type-systems that allow a larger range of keys in maps, and for which it makes sense to specify the domain of the maps. For instance, in Go the syntax `map[int]string` denotes the type of maps from strings to integers, which is the same as Scala's `Map[Int,String]` and Swift's `[Int: String]`. We want to extend the previous system so that it is possible to restrict the domain of the maps and still be able to combine them with single field declarations as in $\{\!|\ \mathit{input} = \texttt{"int"}, \mathsf{Int} \Rightarrow \mathsf{Int}\ |\!\} \vee \{\!|\ \mathit{input} = \texttt{"str"}, \mathsf{String} \Rightarrow \mathsf{Int}\ |\!\}$ which is a slightly modified version of the type we defined at the beginning of Section 4, except we replaced the wildcard "_" with a domain type. Let us call this union type `S`. The idea that the (mandatory) field *input* in the records of type `S` is used as a flag to specify the kind of map the record contains. It becomes then possible to deduce the type `S⟶(integer|nil)` for the following function (in pseudo-syntax):

```
function foo(x : S) =
  match x with
    | { input = "int" } -> x.[42]
    | _ -> x.["42"]
```

The match expression checks whether the field `input` contains the string `"int"` and if so it deduces that `x` maps integers to integers, otherwise it deduces that it maps strings to integers. Since the union `S` is formed by closed record types, trying to select in `x` anything that is not an integer in the first branch and that is not a string in the second branch would yield a static type error.

In some languages, like Erlang and Elixir, it is possible to define the union type above. For instance, in Elixir Typespec's syntax [Elixir] the type `S` can be defined as follows:[12]

```
%{input: "str", optional(string)=>integer} | %{input: "int", optional(integer)=>integer}.
```

However, such types are not actually used (apart from for documentation): Dialyzer [Lindahl and Sagonas 2004], the current default static analysis tool for Erlang and Elixir ignores them, since it does not even blink when the function `foo` above is given type `S⟶(function())` which states that the result of `foo` is a function.

Erlang and Elixir allow specifying map types from any type to any type. Thus, it is possible to define fine-grained map types such as `%{1..* =>integer, *..0 => boolean}` which maps positive integers to integers and negative ones to Boolean values (`n..m` is Elixir notation for the interval between n and m where `*` is the symbol for infinite). Since there is no constraint on the domain of the map, it is possible to define maps in which different fields have overlapping domains, such as `%{1..* => integer, *..5 => boolean}`. In case of overlap Typespec states that the leftmost field has the priority. Clearly this is not compatible with the theory we defined so far (and, more generally, with the semantic subtyping approach) which disregards the order of the fields. If we take the declaration face-value and read it at the light of the interpretation we gave so far, the type

[12] In Typespec, record types are prefixed by `%` and are closed (open ones need to specify `optional(any)=>any`), while basic types are suffixed with open-closed parentheses—e.g., `integer()`, `string()`—: we omitted the latter to enhance readability.

`%{1..* => integer, *..5 => boolean}` must be interpreted as the set of records where every field labeled by a number larger than 1 is either undefined or associated to an integer value, and every field labeled a number smaller than 5 is ever undefined or associated to a Boolean value. Thus, in these records every field labeled by a number between 1 and 5 must be undefined, since otherwise it should be associated to a value that is both an integer and a Boolean, which is impossible. In other words the type above is equivalent to `%{6..* => integer, *..0 => boolean, 1..4 => none}` (`none` is Typespec's empty type). We think that an unbridled use of the domains of maps could be a source of confusion (practical examples of this kind of confusion are discussed in Section 5 on related work). Therefore, we propose to forbid any overlapping between domains in the same map, still preserving the possibility for a domain to overlap with single labels, as it was the case with the maps defined so far. For instance, a type such as the above will be forbidden, but it will be possible to define the type $\{1 = \mathsf{Int}, 2 \Rightarrow \mathsf{Bool}, \mathsf{Int} \Rightarrow \mathsf{String}\}$ whose records have a mandatory field 1 of type integer, an optional field 2 of type boolean, and map *any integer other than* 1 *and* 2 to a string (since it is an open record type, these records may contain other fields for non-integer labels). In other words, we want to allow a record type to specify several fields of the form $t \Rightarrow t$, provided that the domains that are not singleton have pairwise empty intersections. However, enforcing distinct domains to have empty intersections is not straightforward. Even if we require that for each record type atom the domains of its fields do not overlap, this is not enough since the overlap may take place when we compare, intersect, or union different record types. For instance, simple maps with just one field definition such as $\{1..* \Rightarrow \mathsf{Int}\}$ and $\{*..5 \Rightarrow \mathsf{Bool}\}$ obviously satisfy the condition, but their intersection is the type $\{1..* \Rightarrow \mathsf{Int}, *..5 \Rightarrow \mathsf{Bool}\}$, that is, the very type that we discussed as being problematic since it has overlapping domains. To avoid this problem we propose to restrict the domains of maps to be drawn from a predefined fixed set of *key-types* that are mutually non overlapping. In practice, this amounts to modify the definition of fields as follows:

$$\begin{array}{lrcl} \textbf{Fields} & f & ::= & \ell = t \mid \ell \Rightarrow t \mid k \Rightarrow t \\ \textbf{Key Types} & k & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathbb{1}\times\mathbb{1} \mid \mathbb{0}\to\mathbb{1} \mid \{\} \mid \_ \end{array}$$

where ℓ denotes a value of type $\mathcal{L}$. For the sake of simplicity, we used in the definition of key-types only three basic types (Int, Bool, and String) but in practice each language will choose some specific basic types (e.g., in Elixir key-types will also include `atom()`, the type of all atoms, but only String in JavaScript). For the sake of the example we also added $\mathbb{1}\times\mathbb{1}$ the type of all pairs (introduced in Section 2.1). What does matter is that all the key-types (different from _) are pairwise disjoint and that their union is equivalent to $\mathcal{L}$, that is it covers all the admitted record labels.[13] In the case above if we suppose that we specified in the key-types all basic types, then we also have that the union of all key-types is equivalent to $\mathbb{1}$ (i.e., all values can be used as keys for maps, as in Erlang).

As we already hinted, if a field of the form $k \Rightarrow t$ is present in a record type, it means that the records of that type must map every value in k *other than those already defined in the record type* to a value of type t or to be undefined. The presence of a field $\_ \Rightarrow t$ in a record type means that the records of that type map any value of $\mathcal{L}$ *not already specified by a field or a key-type in the type* to a value of type t or to be undefined. For instance, the record value $\{1 = \mathsf{true}, 2 = 42\}$ has many possible types among which $\{1 = \mathsf{Bool}, \mathsf{Int} \Rightarrow \mathsf{Int}\}$, $\{\mathsf{Int} \Rightarrow \mathsf{Bool}, 2 = \mathsf{Int}\}$, $\{\mathsf{Int} \Rightarrow \mathsf{Int}\vee\mathsf{Bool}, \_ \Rightarrow \mathsf{String}\}$; however it has neither type $\{\mathsf{Int} \Rightarrow \mathsf{Int}\}$—since 1 is mapped to a Boolean—, nor type $\{\mathsf{Int} \Rightarrow \mathsf{Bool}, \mathsf{Int} \Rightarrow \mathsf{Int}\}$, since all fields must have non-overlapping left-hand sides.

The approach we propose allows maps from any value (to any value). It is less flexible than Erlang's unrestricted approach since, for instance, we cannot define maps from pairs of integers, but

[13] Since the union of all key-types is equivalent to $\mathcal{L}$, the presence of "_" among them constitutes just a convenient syntactic sugar. We could require that the union of all key types is just contained in $\mathcal{L}$: in that case the presence of "_" would be necessary. Here we preferred to consider only the first case since it yields simpler definitions and simpler proofs.

only maps from pairs of values: $\{\!|\mathbb{1}\times\mathbb{1} \Rightarrow \mathsf{Int}|\!\}$ but not $\{\!|\mathsf{Int}\times\mathsf{Int} \Rightarrow \mathsf{Int}|\!\}$ (see however Section 4.5 on design choices). But it has two important advantages: first, it greatly simplifies the definition and decision procedure of the subtyping relation since, as we show next, we can reuse all the definitions, formulas, and algorithms defined so far for records with just minimal modifications; second and foremost, it eliminates a bunch of ambiguities that overlapping domains would surely generate.

In order to account for these new map types, we have to switch from quasi-constant functions to the more general *quasi* $\mathbb{K}$*-step functions*: these are step functions[14] that are constants on a predefined finite partition $\mathbb{K}$ of the domain apart from on a finite set of elements.

In the previous sections, we interpreted record atoms into quasi-constant functions of the form $\{\!|\ell_1 = \tau_1, ..., \ell_n = \tau_n, \_ = \tau_w|\!\}$ where the τ_i's are of the form t or $t\vee\bot$. The idea now is that the new record atoms of this section will be interpreted in quasi $\mathbb{K}$-step functions of the form:

$$\{\!|\ell_1 = \tau_1, ...\ell_n = \tau_n, \mathit{bool} = \tau'_1, \mathit{int} = \tau'_2, \mathit{string} = \tau'_3, \mathit{prod} = \tau'_4, \mathit{arrow} = \tau'_5, \mathit{recd} = \tau'_6|\!\}$$

denoting the quasi $\mathbb{K}$-step function that maps ℓ_i to τ_i and the key-types Bool, Int, String, etc., in the respective τ'_i's. In other words, the "_" field for the default value that in quasi-constant functions covered all the keys in $\mathcal{L}$ not specified in a type, is in quasi $\mathbb{K}$-step functions partitioned into the various key-types (6 in our example, forming our $\mathbb{K}$), whose union yields $\mathcal{L}$. For instance, an open record type such as $\{$"a" = Int, String $\Rightarrow$ Bool$\}$—which maps the string "a" into integers, any other string into Boolean values, and does not set any constraint on the remaining keys—, denotes the quasi $\mathbb{K}$-step function $\{\!|$"a" = Int, bool=$\mathbb{1}\vee\bot$, int=$\mathbb{1}\vee\bot$, string=Bool$\vee\bot$, prod=$\mathbb{1}\vee\bot$, arrow=$\mathbb{1}\vee\bot$, recd=$\mathbb{1}\vee\bot|\!\}$, while its closed record type counterpart $\{\!|$"a" = Int, String $\Rightarrow$ Bool$|\!\}$ denotes the quasi $\mathbb{K}$-step function $\{\!|$"a" = Int, bool=$\bot$, int=$\bot$, string=Bool$\vee\bot$, prod=$\bot$, arrow=$\bot$, recd=$\bot|\!\}$. In both cases notice that records of these types map all strings to Boolean values or undefined, except "a" which must be mapped to an integer. As before the wild-card "_" key-type, denotes all the other keys not specified in the record type, but here it is just some syntactic sugar to avoid repeating the fields: e.g., $\{\_ \Rightarrow \mathsf{Int}\}$ is equivalent to $\{\mathsf{Int} \Rightarrow \mathsf{Int}, \mathsf{Bool} \Rightarrow \mathsf{Int}, \mathsf{String} \Rightarrow \mathsf{Int}, \mathbb{1}\times\mathbb{1} \Rightarrow \mathsf{Int}, \mathbb{0}\to\mathbb{1} \Rightarrow \mathsf{Int}, \{\} \Rightarrow \mathsf{Int}\}$.[15]

Extending the previous theory to account for quasi $\mathbb{K}$-step functions requires minimal modifications. The main difference is that the default value function $\mathsf{def}(\mathsf{R})$ now becomes a family of types indexed over the set of key-types $\mathbb{K} = \{\mathit{bool}, \mathit{int}, \mathit{string}, \mathit{prod}, \mathit{arrow}, \mathit{recd}\}$. Formally,

Definition 4.6 (Quasi $\mathbb{K}$-step function). *Let $\mathcal{L}$ denote a set of keys, and $\mathbb{K}$ a finite partition of $\mathcal{L}$ (i.e., all $k \in \mathbb{K}$ are pairwise disjoint and $\mathcal{L} = \bigcup_{k\in\mathbb{K}} k$). Let Z denote some set, a function $r : \mathcal{L} \to Z$ is a* quasi $\mathbb{K}$-step *function, if for all $k \in \mathbb{K}$ there exists $z_k \in Z$ such that the set $\{\ell \in k \mid r(\ell) \neq z_k\}$ is finite; we denote by* $\mathsf{dom}(r)$ *the union of all these finite sets, and by* $\mathsf{def}(r)_k$ *the element z_k. We use $\mathcal{L} \xrightarrow{\mathbb{K}} Z$ to denote the set of quasi $\mathbb{K}$-step functions from $\mathcal{L}$ to Z.*

If $\mathbb{K} = \{k_1, ..., k_m\}$ we use the notation $\{\!|\ell_1 = z_1, \ldots, \ell_n = z_n, k_1 = z'_1, \ldots, k_m = z'_m|\!\}$ to denote the quasi $\mathbb{K}$-step function $r : \mathcal{L} \xrightarrow{\mathbb{K}} Z$ defined by $r(\ell_i) = z_i$ for $i = 1..n$ and $r(\ell) = z'_j$ for $\ell \in k_j \setminus \{\ell_1, \ldots, \ell_n\}$ and $j = 1..m$. The universal model for Erlang-style maps is the same as the one defined in Section 3.2.1 and so is the definition of the binary predicate $(d : t)$, with the only difference that we have $r \in \mathcal{L} \xrightarrow{\mathbb{K}} \mathcal{T}_\bot$ (instead of $r \in \mathcal{L} \to \mathcal{T}_\bot$) in the following clause:

$$(\{\!|\ell_1 = \partial_1, ..., \ell_n = \partial_n, \_ = \bot|\!\} : r) \;=\; \begin{array}{l}(\forall i{\in}[1..n]\,.\,(\partial_i : r(\ell_i)) \text{ or } (\partial_i = \bot \text{ and } \bot \leq r(\ell_i))) \text{ and}\\ (\forall \ell \in \mathcal{L}\setminus\{\ell_1, ..., \ell_n\}\,.\,\bot \leq r(\ell)).\end{array} \tag{21}$$

If $(Z_\ell)_{\ell\in\mathcal{L}}$ is a family of subsets of Z indexed by $\mathcal{L}$, we denote by $\boxtimes^{\mathbb{K}}_{\ell\in\mathcal{L}} Z_\ell$ the subset of $\mathcal{L} \xrightarrow{\mathbb{K}} Z$ formed by all quasi $\mathbb{K}$-step functions r such that $r(\ell) \in Z_\ell$ for all $\ell \in \mathcal{L}$. Without loss of generality,

[14] A step function is a piecewise constant function having only finitely many pieces.

[15] We could also use union on key-types to avoid repetition: e.g., $\{\mathsf{Int}\vee\mathsf{Bool} \Rightarrow \mathsf{Int}\}$ for $\{\mathsf{Int} \Rightarrow \mathsf{Int}, \mathsf{Bool} \Rightarrow \mathsf{Int}\}$.

we just consider the case in which $\mathbb{K}$ contains only infinite sets (since every finite key-type can be dealt by directly specifying the type of each of its key). Under this hypothesis Lemma 2.3 becomes:

Lemma 4.7 (Map Containment). *Let $(X_p)_{p\in P}$ and $(X_n)_{n\in N}$ be two families of elements of $\mathcal{L} \xrightarrow{\mathbb{K}} \mathcal{P}(\mathcal{D})$. Let $L = \bigcup_{i\in P\cup N} \mathsf{dom}(X_i)$. Then $\left(\bigcap_{p\in P} \fatbslash_{\ell\in\mathcal{L}} X_p(\ell)\right) \subseteq \left(\bigcup_{n\in N} \fatbslash_{\ell\in\mathcal{L}} X_n(\ell)\right)$ if and only if either $\bigcap_{p\in P} \mathsf{def}(X_p) = \emptyset$, or for every map $\iota : N \to L \cup \mathbb{K}$*

$$\left(\exists \ell\in\mathcal{L}.\left(\bigcap_{p\in P} X_p(\ell) \subseteq \bigcup_{n\in N|\iota(n)=\ell} X_n(\ell)\right)\right) \text{ or } \left(\exists n_\circ\in N, k\in\mathbb{K}.\left(\iota(n_\circ) = k\right) \text{ and } \left(\bigcap_{p\in P} \mathsf{def}(X_p)_k \leq \mathsf{def}(X_{n_\circ})_k\right)\right)$$

Proof. See Appendix C.1 □

The reader who went through the proof in Appendix C.1 will have noticed that the hypothesis that every key-type k is infinite is actually used. This means that the case of a field such as Bool ⇒ Int must be checked by considering it as syntactic sugar for two fields true ⇒ Int and false ⇒ Int.

Checking the emptiness of an intersection of the new record atoms and their negation requires checking the default values component-wise. In particular, to verify the containment of the new maps it suffices to change the definition of (19) into

$$\begin{array}{rcl}\Phi(\mathsf{R}_\circ, N\cup\{\mathsf{R}\}) &=& \mathtt{if}\ \ \forall k\in\mathbb{K}\,.\,(\mathsf{def}(\mathsf{R}_\circ))_k \leq (\mathsf{def}(\mathsf{R}))_k \\ && \mathtt{then}\ \forall \ell\in L\,.\,(\,\mathsf{R}_\circ(\ell) \leq \mathsf{R}(\ell)\ \ \mathtt{or}\ \ \Phi(\mathsf{R}_\circ \wedge \{\!\{\ell : \mathtt{not}(\mathsf{R}(\ell))\}\!\}, N)\,) \\ && \mathtt{else}\ \ \Phi(\mathsf{R}_\circ, N)\end{array} \tag{22}$$

Finally, we have to define the type operators for these new maps. Again, we have to make minimal modifications with respect to Section 4.1 and 4.2. We can use the same notation for record atoms as in Definition 4.1, that is, $\langle\!\langle (\tau_\ell)_{\ell\in L}\ ;\ t_0\ ;\ S\rangle\!\rangle$ but with the restriction that t_0 and the types in S are tuples whose arity is the cardinality of $\mathbb{K}$. In particular, if $\mathbb{K} = \{k_1, ..., k_n\}$, then $\langle\!\langle (\tau_\ell)_{\ell\in L}\ ;\ (t_1\times...\times t_n)\ ;\ \emptyset\rangle\!\rangle$ represents the record $\{\!\{\ell_1 = \tau_{\ell_1}, \dots, \ell_n = \tau_{\ell_n}, k_1 \Rightarrow t_1, \ ..., \ k_n \Rightarrow t_n\}\!\}$ while for $\langle\!\langle (\tau_\ell)_{\ell\in L}\ ;\ (t_1\times...\times t_n)\ ;\ S\rangle\!\rangle$ every $(s_1\times...\times s_n)$ in S adds to the type above the constraint "... and there exists a key-type k_i and a label ℓ in $k_i \setminus L$ that is mapped into a value of type $\neg s_i$" (since, a tuple is in the negation of a tuple type, if one of its projections does so). Properties (11–13) and Theorem 4.2, continue to hold, and so do Lemma 4.4 and Theorem 4.5: it suffices to specialize their proofs to the case in which the default type and the types in the S's sets of the atoms are all tuples of the same arity. In particular, Theorem 4.5 proves the correctness of (22), since this formula is the specialization of (19) when the default type is a tuple.

Regarding type operators, we have to change the definition of label projection as follows

$$\boldsymbol{\pi}_\ell(t) = \begin{cases} \bigvee_{\langle\!\langle (\tau_m)_{m\in L}\ ;\ t_0\ ;\ S\rangle\!\rangle\in\pi^L_{\mathbf{rec}}(t)} \tau_\ell & \text{if } \ell\in L \\ \bigvee_{\langle\!\langle (\tau_m)_{m\in L}\ ;\ t_0\ ;\ S\rangle\!\rangle\in\pi^L_{\mathbf{rec}}(t)} \pi_i(t_0)\vee\bot & \text{if } \ell\in(k_i\setminus L)\end{cases} \tag{23}$$

where π_i denotes tuple projection. As before, $t.\ell$ is $\boldsymbol{\pi}_\ell(t)$ if $\boldsymbol{\pi}_\ell(t)\wedge\bot \simeq \mathbb{0}$ and is undefined otherwise. Notice that the definition is well-given, since the k_i's key-types partition the set of labels and if $\ell\notin L$, then there exists one and only one k_i such that $\ell\in(k_i\setminus L)$.

Regarding map-selection, again, as before we have that $t.[t']$ is defined as $\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t)$ if $\bot \wedge \bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t) \simeq \mathbb{0}$ and as $\mathtt{nil} \vee (\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t) \setminus \bot)$ otherwise. The only difference is that to express $\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t)$ as a finite union we have to pick a random label $\bar{\ell}_i$ not in L for each key-type k_i.

Finally, while map-deletion does not change (the operator does not use the t_0 parts), the map-update operator must update the t_0 parts component-wise, that is: let $t \leq \{\!\{\}\!\}$ with $\mathsf{dom}(t)\subseteq L$ and $t' \leq \mathcal{L}$, then $t\leftarrow[t']=t''\rangle = \bigvee_{\langle\!\langle (\tau_\ell)_{\ell\in L}\ ;\ (t_1\times...\times t_n)\ ;\ S\rangle\!\rangle\in\pi^L_{\mathbf{rec}}(t)} \langle\!\langle (\tau'_\ell)_{\ell\in L}\ ;\ (t'_1\times...\times t'_n)\ ;\ S\rangle\!\rangle$ where τ'_ℓ is $\tau_\ell\vee t''$ if $\ell\in L\wedge t'$ and is τ_ℓ otherwise, and for $i = 1..n$ the type t'_i is $t_i\vee t''$ if $t'\wedge k_i \not\simeq \mathbb{0}$ and t_i otherwise. Likewise, for the concatenation type operator we can apply the same formula as before,

that is (15), but where the set-theoretic operations to compute t_3 are intended to be performed on tuples and computed component-wise.

4.5 Design and variations

The decision of restricting the domains of maps to a predefined set of key-types is both a design and an implementation choice. It is possible to define a theory for maps with overlapping domains, but in that case, there would not be any difference between record types and an intersection of function types whose codomain may contain an undefined value. Indeed, when comparing two records, we would have to compare mappings with possibly partially overlapping domains, which would require the level of sophistication used to compare intersections of arrow types. For example, the type `%{1..* => integer, *..0 => boolean}` would be akin to the function type $([1..*] \to \mathsf{Int}\vee\bot) \wedge ([*..0] \to \mathsf{Bool}\vee\bot)$, while its open record type counterpart would correspond to $([1..*] \to \mathsf{Int}\vee\bot) \wedge ([*..0] \to \mathsf{Bool}\vee\bot) \wedge (\neg\mathsf{Int} \to \mathbb{1}\vee\bot)$, yielding all the possible ambiguities we already pointed out (see also Section 5 on related work for examples of such ambiguities in real-world languages). The advantage of our choice is not only that it avoids all such ambiguities, but also that it yields a compact representation of records types (i.e., as unions of atoms of the form $\langle\!\langle (\tau_\ell)_{\ell\in L} \;;\; (t_1\times\ldots\times t_n) \;;\; S \rangle\!\rangle$: see also Remark 4.3), an intuitive implementation of type operators for records, and a simple and efficient backtracking-free subtyping algorithm, via the function Φ defined in (22). Last but not least, this solution provides a solid starting point for future work: in particular, it does not seem conceptually difficult to extend this system with row polymorphism (though, the technical development looks hard: see Section 6) by adding to the representation of record types of Theorem 4.2 information about row variables; instead, we do not have a clue about how this extension could be done if record types were generic intersections of arrow types.

The price to pay is lesser freedom than the one permitted by Erlang's Typespec syntax for maps (but freedom that, as far as we know, is used only for documentation purposes, rather than for performing precise type analysis) and less precise typing of maps resulting to fewer statically-detected errors—but, without hindering soundness. For instance, in the formulation given in this section it is not possible to specify that the keys of a given map are, say, only pairs of integers: statically, we can just enforce that all keys are pairs of values (of any type) and using a pair of strings is accepted, even if it will always return `nil`. That said, the theory leaves a lot of margin for variations.

Key-types partitions. To apply Lemma 4.7 on map containment, we need the set of keys to be partitioned into a finite number of infinite key-types. This partition must be the same for the whole program, but nothing prevents us to use different partitions in different programs. Therefore, this partitioning can be user-defined. Of course, this raises modularity concerns when combining programs with different partitions. But again, this is a problem of language design and implementation, not of types. For instance, at the moment of writing we are studying with the development team of Elixir the possibility of having user-defined partitions for maps whose keys are tuples. The idea is that the programmer will be allowed to declare the tuple key-types used in her/his program (e.g., pairs of integers, triplets of a string and two integers,...). Record type atoms will store in their tuple component a binary tree (instead of a single default type) that represents the finite partition of the tuple space, and when composing two different programs the subtyping algorithm will merge the two corresponding trees to compute the meet of the two partitions (finite partitions form a complete lattice). Once this will be achieved, then we will study how to make the system *infer* this partition, without the need for the programmer to declare them, thus making their use transparent.

Default values. Another obvious variation that our theory can account for are record expressions with a default value (which is a feature of Ballerina's records). For instance, one could allow the program to initialize the constant part of a record expression, by using the syntax $\{\ell_1 = e_1, ...\ell_n = e_n, \_ = e\}$. It would then be possible to relax the constraint that fields for (infinite) key-types must be optional—e.g., $\{\texttt{"uno"} = 1, \texttt{"due"} = 2, \_ = 0\}$ is a map of type $\mathbf{\{}\mathsf{String} = \mathsf{Int}\mathbf{\}}$ (notice the mandatory field in the type). The selection of a string key for a record of this type will then have type Int (rather than $\mathtt{nil}\vee\mathsf{Int}$) and we should modify the map-selection operator $t.[t']$ to be undefined when the projection contains $\bot$. To obtain the same behavior as before it will suffice to initialize all maps to $\{\_ = \mathtt{nil}\}$. If we are working with quasi $\mathbb{K}$-step functions, then record expressions could also specify default values just for some specific key-types, such as in $\{\texttt{"uno"} = 1, \texttt{"due"} = 2, \mathsf{String} = 0\}$.

Access primitives. Our theory can account for access primitives other than those defined at the beginning of Section 4. This is the case, for instance, of Elixir's `fetch!(e1,e2)` which returns the value associated to the key returned by $\mathtt{e}_2$ in the field of the map returned by $\mathtt{e}_1$ and raises an error if there is no such a field. We see that `fetch!()` performs a selection that lies between the struct-like access of $e.\_$ and the map-like access of $e.[\_]$: as in map-like accesses the key may be the result of an expression; as in struct-like accesses if the key is absent, then it yields a run-time error. Likewise, the typing of `fetch!()` is half-way between the two typing disciplines: as a struct-like access it yields a static type error if $\mathtt{e}_2$ may return a key which is always undefined; as a map-like access it accepts a selection if $\mathtt{e}_2$ will always return a key that may be undefined. For instance, if $e_1 : \mathbf{\{|}\,\mathsf{Int} \Rightarrow \mathsf{String}\,\mathbf{|\}}$ and $e_2 : \mathbb{1}$, then $e_1.3$ is ill typed, $e_1.[3]$ and $e_1.[e_2]$ have both type $\mathsf{String}\vee\mathtt{nil}$, `fetch!(`$e_1, 3$`)` has type String, `fetch!(`e_1, e_2`)` is ill-typed, $e_1.[\texttt{"3"}]$ returns a warning, and `fetch!(`e_1, "3"`)` is ill-typed. The rationale of such a typing discipline is that a typical use of `fetch!()` in Elixir is to access maps from references to process identifiers (both being unique internal identifiers) used to monitor processes: when a process dies, a message with a reference is received and the programmer knows that this reference must be present in the map, thought it is impossible to statically ensure it; the absence of such a reference pinpoints a bug in the implementation of monitoring which must thus raise a run-time error (and not just result in `nil`); however, trying to access such maps with a key other than a reference is an error that must be captured at compile time. It is a simple exercise to use the label projection type operator to define a fetch type operator implementing this discipline.

Selection primitives. Finally, we may want to devise a different type discipline for the selection operations we already defined. Consider again $e_1 : \mathbf{\{|}\,\mathsf{Int} \Rightarrow \mathsf{String}\,\mathbf{|\}}$: we have seen that if $e_2 : \mathbb{1}$, then $e_1.[e_2]$ has type $\mathsf{String}\vee\mathtt{nil}$. This is sound, since whatever e_2 returns, the result of the selection will be either a string (i.e., e_2 returns an integer key defined in the result of e_1) or `nil` (in all other cases). A warning is issued only if the type of e_2 is contained in $\neg\mathsf{Int}$ (since the selection will have type `nil`, that is, it will always return `nil`). However, we may want to implement a stricter type discipline for map selection and raise an error (or issue a warning) whenever the type of e_2 is *not* completely contained in Int. Again, this can be obtained by a straightforward modification of the $t.[t']$ operator in terms of the projection operator given in (23), namely: $t.[t']$ is defined as $\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t)$ if $\bot \wedge \bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t) \simeq \mathbb{0}$; else as $\mathtt{nil} \vee (\bigvee_{\ell\in t'} \boldsymbol{\pi}_\ell(t) \setminus \bot)$ if $\forall k \in \mathbb{K}$ and $\forall \ell \in (k\wedge t')\setminus\mathsf{dom}(t)$, $\boldsymbol{\pi}_\ell(t) \not\leq \bot$; and undefined otherwise.[16] Likewise, we can modify the second constraint into $\forall k \in \mathbb{K}$ and $\forall \ell \in (k\wedge t')\setminus\mathsf{dom}(t)$, $\mathbb{1} \not\leq \boldsymbol{\pi}_\ell(t) \not\leq \bot$ if we want to impose the same stricter discipline on open maps, too.

In this section, we presented some examples of variations that are possible in our framework; a few more were discussed in the previous sections, but many others are possible. This same variety

[16] Once again, we do not need to check all the labels in $(k\wedge t')\setminus\mathsf{dom}(t)$, a single one will suffice.

can be found among the different definitions and implementations of records to be found in actual programming languages. The common denominator of all these variations and the core of our presentation is the interpretation of record values and record types as quasi $\mathbb{K}$-step functions and sets thereof, together with the various definitions this interpretation yields: the decomposition of disjunctive normal forms of record types (Lemma 4.7), the backtracking-free implementation of record subtyping (function Φ and Theorem 4.5), the compact representation of record type atoms inducing a simple definition of record type projection and concatenation (Theorem 4.2, and formulas (14), (23), (15), and (16)). The definition of typing rules for the various variations we discussed and of the type-operators they use play a secondary role, and they are an accessory to the presentation of the characteristics of the type interpretation and of the variety it permits.

5 RELATED WORK

It is impossible in the remaining space to give an even approximated overview of the literature on records and record types. Thus, we limit to list what we consider some important milestone, acknowledging that the list is far from being complete.

We already recalled in the introduction that records were first proposed in 1965 by C.A.R. Hoare in a series of papers on "record handling" whose adoption in Simula 67 yielded the concepts of objects and classes. Cardelli [1984] introduced the basic notions of record types, as intended nowadays, and the formal study of subtyping. Then, Wand [1987] introduced the concept of row-variables to solve the type inference problem for records. His system was later refined and shown to have principal types in [Jategaonkar and Mitchell 1988; Rémy 1989; Wand 1989] thus providing a flexible integration of record types and Hindley-Milner type systems. Another important milestone is the work on operations on records by Cardelli and Mitchell [1990] who describe a second-order type system that incorporates extensible records. They give up type inference and principal typing but gain in expressiveness: their work opens up the design space of operations on records (e.g., field extraction, deletion, extension, update, akin to the operations we used here) and row-variables fall out naturally from second-order type variables. They also identify static typing of record concatenation as a particularly difficult problem because of label conflicts. Pottier [2000], building on the work by Rémy [1995], types record concatenation by solving subtyping constraint systems in which the type "`Abs`" for undefined fields is separated from all other types, akin to what we did here by considering the (disjoint) union $t \vee \bot$; this union in [Pottier 2000] is denoted by $\texttt{Either}\, t$ and is (meta-theoretically) equivalent to $\texttt{Abs} \vee \texttt{Pre}\, t$. A categorical model for records (and bounded polymorphism) was given by Bruce and Longo [1988] whose approach unifies the mathematical understanding of polymorphic, dependent, and record types. In particular, they interpret records as indexed products in a PER model, formalizing the idea that record types may be viewed as dependent types. Interestingly, they stress that a key point of their semantics is its "set-theoretic" flavor. The view of records as dependent types was put into practice by Chlipala [2010] who introduces first-class type-level keys and records, making it possible to write record-manipulating metaprograms and build domain specific languages (e.g., for the development of web applications).

The types for records in all the cited works cover only the struct usage scenario and not the one for maps (according to the terminology we adopted in this work). Furthermore, none considers either union or intersection types. The only work that studies records with set-theoretic types and semantic subtyping is Frisch's PhD thesis [Frisch 2004], which our work is built upon. Frisch [2004] interprets records as quasi-constant functions and uses the interpretation to define the subtyping relation and the decomposition rule to decide it, and to type the calculus with structs we presented in Section 3. We conservatively extended his work to uniformly type both structs and maps, including what we called the Erlang-style maps. For the latter, we conservatively extended both the theory of quasi-constant functions (needed to define the subtyping relation) and the

internal representation of records types (needed to define and compute the type operators used in the typing rules). We also formalized for records the backtracking free algorithm—obtained by adapting the one for product subtyping defined by Frisch [2004]—and proved it sound and complete.

TypeScript and Flow are two popular static type systems on the top of JavaScript that use unions, intersections, and records types. They use a syntactic approach to subtyping and account for mutability, which yields a subtyping relation different from the one studied here, therefore our algorithms do not directly apply to them. Their languages of types can express the mix of structs and maps in a limited form. For instance, Flow introduces "indexer properties" to type objects with fields whose keys are not statically known, as exemplified by the listings below:

```
1 type A = { a: string, [string]: number, ... };
2 type B = { b: number,
3            [c: [string,number]]: number, ... };
4 type C = { [string]:string, ... };
5 const x : A = { a: "ok", b: 42 };
6 const y : A = { a: 42, b: "ok" };      // error
```

```
7 const z : B = { b: 3, e: "ok" };       // error
8 const u : A & C = { a: "ok", b: 3 }    // error
9 const w : A & C = { a: "ok" }          // error
10 function f(x: A & C) {
11     x.e++; x.e=3; x.e="ok"
12 }
```

The type A (line 1) types records in which label a is mapped to a string and any other label (by default all labels are strings) is mapped to a number: this is shown by the definition of x (line 5) which type-checks; trying to associate a to a value that is not a string *or* b to a value that is not a number yields a static type error (line 6). Contrary to our approach, indexer properties are not restricted to a finite set of key-types: this is shown by type B which maps both the label b (line 2) and pairs of string and numbers (line 3) to numbers.[17] To avoid overlapping key-types, Flow allows at most one indexer property in a record type, a restriction absent in our system. Indexer properties work fine in simple cases, but problems arise when types are composed. The notation "..." in the listing (lines 1 and 3) indicates that the record types are open, but this is overridden by the presence of an indexer, as shown by the definition of z (line 7), which is statically rejected: since B is open, then any key other than b or a string-number pair should be allowed to have any type, but Flow motivates its rejection by the fact that e is not of type `[string,number]`; if we remove the indexer (line 3) from the definition of B, then z type-checks. Furthermore, the ambiguity of overlapping key-types can be reintroduced by using an intersection such as A&C, which ambiguously requires labels other than a to be mapped both to numbers and to strings. Flow does not permit to build a value of this type: the definition of u (line 8) fails since 3 is not of type string as required by C; more surprisingly also w (line 9) also fails, apparently because not compatible with the indexer in C, which suggests that indexers are not really optional fields. However, Flow allows a type like A&C to be used contravariantly, with a rather unclear semantics, as the function f (lines 10-12) shows: the function type-checks (even though no arguments can be defined for it); the two assignments in its body suggest that the type checker consider x.e to be of type `number|string`, thus that it approximated the intersection of the two indexers by their union; however, this is contradicted by the expression x.e++ which types only if x.e is of type number; even stranger, if we replace x.e++ by x.e.length (which works only for strings), then it is rejected because the type-checker expects e to be a `number` as specified in A, thus giving a priority to A over C in the intersection. These examples show the kind of ambiguities that the system we presented in Section 4.4 is designed to avoid.

Typescript uses a much stricter approach, closer to the one in Section 4.4. It uses the same syntax as Flow's indexer properties (there, called *indexed signatures*) but it imposes two important restrictions: first, as in our approach, only types from a predefined finite set of types (i.e., `string`, `number`, `symbol`, or unions of their literals) can be used as key-types; second, if single fields are present, then their type must refine the indexer type. So while the type A (line 1) will be rejected

[17] The c prefixing the tuple type is an optional name used for documentation.

because declaring a to be of type `string` is not compatible with the indexer, a type such as `{ a: string, [string]: number|string }` will be accepted. Indexers can be repeated, but they must obey the same refinement rules as for single fields. TypeScript provides syntactic sugar for record types with only one indexer: type `C` (line 4) can be equivalently written as `Record<string,string>`. Our approach is more general, since it imposes no restriction on the types of single fields or of multiple indexers.

Luau [Luau], a gradually typed language by Roblox that recently adopted semantic subtyping [Jeffrey 2022], uses the same solution as Flow, with more or less the same syntax but lesser problems and clearer semantics: as Flow, it forbids both multiple indexers and the creation of values in the intersection `A&C` (with clearer error messages); it also type-checks a function `f` with a parameter `x` of type `A&C` (line 10); however, in the body of such a function an expression like `x.e` is (correctly) given type `number&string`, which is why expressions such as `x.e++` and `x.e.length` type check, while any assignment to `x.e` is statically flagged as a type error. Another language whose type system was inspired by semantic subtyping is Julia [Julia], but it provides only struct record types, while maps are handled by a separate `Dict` collection. A formalization of Julia type system is given by [Zappa Nardelli et al. 2018] but it does not cover record types.

Typed Clojure [Bonnaire-Sergeant et al. 2016] is an optional type system with union types and struct-like record types which essentially behave as the records of Section 3 (without the operations of concatenation and deletion). The interest of the work is that records are combined with multiple dispatching and occurrence typing. While the former can be partially simulated by functions with intersection types and pattern matching (as in Section 3.3), full-fledged occurrence typing for semantic subtyping requires more sophisticated techniques introduced by Castagna et al. [2022].

Finally, quite recently Xu et al. [2023] proposed a calculus with unions and intersection where records are built and typed by a *merge* operator which can be seen as a generalization to all types of our record type merge operator of Section 2.2.2. The generalization allows the authors to encode a variety of operations on traits and a rich set of record operators but, as for all formal studies we cited above, their work covers only the records-as-structs usage scenario.

6 CONCLUSION

The motivation of this work comes from our recent and ongoing collaborations with the development teams of two languages, Elixir and Ballerina, aiming at defining new type systems based on semantic subtyping to be implemented in their compilers. In both cases, an important amount of time was and is being spent on record types. Although the two languages are quite different, both needed to type single records uniformly both as maps and as structs in the way presented here. For this, the typing of records as implemented by the language CDuce and which is based on the theory of quasi-constant functions by Frisch [2004, Section 9.1] was not enough, and we had to extend and generalize both the theory and the algorithmic aspects of quasi-constant functions, by devising the quasi $\mathbb{K}$-step functions we introduced here. For the sake of the presentation, in this article we recalled some results on records by Frisch [2004], such as the decomposition law of Lemma 2.3, the properties (11-13), Definition 4.1, and Theorem 4.2: in those cases the original work is explicitly cited; in all the other cases the results presented here are new. In particular, the function Φ for the subtyping algorithm of Section 4.3 is a high-level formal specification and a generalization of what happens in the (highly-optimized) code of the CDuce compiler, and that was never formalized; the proof of its correctness (Theorem 4.5), thus, is a new result. Also, obviously, everything that concerns maps (as opposed to structs) is original to this work: the definitions of the type operators for map-like operations; the generalization of quasi-constant functions into quasi $\mathbb{K}$-step functions; the derivation of the corresponding decomposition law stated in Lemma 4.7, which generalizes Lemma 2.3; the generalization of the Φ function to work with quasi $\mathbb{K}$-step functions, and the fact that the latter are defined so that they preserve all the previous meta-theoretic properties.

The work presented here lays the foundation to type-check records in Ballerina, Elixir, and, we hope, other languages, but both for Ballerina and Elixir, the problem of typing records goes well beyond what we presented here.

For Ballerina, we had to formalize the fact that record fields in Ballerina may be mutable. The challenging part was that the design principles of Ballerina require the subtyping relation to satisfy properties at first sight counterintuitive to a functional programmer (immutable pointers can be used where mutable ones are expected, reference types are covariant, pattern matching disregards mutability) and this yielded an original and intriguing theory for records with mutable fields that space constraints did not allow us to present here: the curious reader can find it in Appendix F.

For Elixir, apart from the conundrum of adapting types to its parser (meta-programming in Elixir heavily restricts the possible syntactic choices for types), an important challenge was to take into account the pervasive use of guards, which determine the types of functions and case expressions and whose analysis must be adapted to our record types (see [Castagna et al. 2023]).

Another challenge for Elixir is polymorphism, without which libraries cannot be typed in any sensible and practical way. While it is possible to graft on Elixir the polymorphic types defined for semantic subtyping by Castagna et al. [2015, 2014], this is not enough to type even simple record operations. We leave the reader as an exercise to understand why a definition as simple as $\lambda x.\lambda y.(x \mathbin{+} \{\ell = y\})$ cannot be given type $\forall \alpha, \beta\,.\,(\alpha \wedge \{\!\}) \to \beta \to (\alpha \wedge \{\ell = \beta\})$. This would be unsound (see Appendix E) and the only way we see to type this function—while preserving the type of all fields of its arguments—is to use *row polymorphism* [Rémy 1989; Wand 1989] and to type it as $\forall \alpha, \beta\,.\,\{\alpha\} \to \beta \to \{\ell = \beta, \alpha\}$ where α is a row variable. This is the reason why we are currently studying how to combine semantic subtyping and row polymorphism, which is particularly challenging: polymorphism is added to semantic subtyping by giving a set-theoretic interpretation of types parametric in the interpretation of their type variables and requiring that subtyping holds for all possible interpretations of the variables [Castagna and Xu 2011]. But this is hard with row variables because the meaning of substitutions depends on the context in which they are applied. For example, if we instantiate a variable α by the row "$\_ \Rightarrow \mathsf{Bool}$", then the instantiation will have different meanings if we consider, say $\{\alpha\}$ or $\{\ell = \mathsf{Int}, \alpha\}$, since the instance of the latter will require ℓ to be defined and mapped to an integer, while in the instance of the former ℓ is either undefined or bound to a Boolean. Still some intriguing work to cut out.

ACKNOWLEDGMENTS

This research is issued from the author's collaboration with the Ballerina and Elixir development teams. I am particularly indebted to José Valim (designer, main developer, and omnipotent wizard of Elixir) and James Clark (who is supervising the design and implementation of the next generation Ballerina type system) for the long discussions and exchanges we had on record types. I am grateful to Loïc Peyrot who made an extensive reading of an earlier version of this work and spotted a couple of errors that are now fixed. The presentation benefited from important feedback provided by Daniele Varacca, Guillaume Duboc, and Mickaël Laurent.

This work is dedicated, on the occasion of his retirement, to Kim Bruce whose paper on records, inheritance, and bounded quantification [Bruce and Longo 1988] was one of the first scientific papers I ever read. I am indebted to Kim for all his feedback, encouragement, discussions at the dawn of my career, but in particular because when I was just starting my PhD., during a visit to Williams College to see Prof. Bruce, he made me discover both the beauty and richness of the theory of records (and its application to objects) and how delicious falafels are (the latter with the complicity of his wife Fatma). I still have fond memories of that visit, and since I am not sure whether an essay on falafels written by me would be appreciated, I opted to write on records, hoping to have more success. Thanks Kim.

REFERENCES

Ballerina. *Ballerina Language Specification.* https://ballerina.io/learn/ballerina-specifications/

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-Centric General-Purpose Language. In *ICFP '03, 8th ACM International Conference on Functional Programming*. ACM Press, Uppsala, Sweden, 51–63.

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *Programming Languages and Systems - 25th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 68–94. https://doi.org/10.1007/978-3-662-49498-1_4

Kim B. Bruce and Giuseppe Longo. 1988. A Modest Model of Records, Inheritance and Bounded Quantification. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*. IEEE Computer Society, 38–50. https://doi.org/10.1109/LICS.1988.5099

Luca Cardelli. 1984. A Semantics of Multiple Inheritance. In *Semantics of Data Types (Lecture Notes in Computer Science, Vol. 173)*. Springer, 51–67. https://doi.org/10.1007/3-540-13346-1_2

Luca Cardelli and John C. Mitchell. 1990. Operations on Records. In *Proceedings of the Fifth International Conference on Mathematical Foundations of Programming Semantics* (New Orleans, Louisiana, USA). Springer-Verlag, Berlin, Heidelberg, 22–52. https://doi.org/10.1007/BFb0040253

Giuseppe Castagna. 2020. Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science* 16, 1 (2020), 15:1–15:58. https://doi.org/10.23638/LMCS-16(1:15)2020 Corrected and extended version available at https://www.irif.fr/~gc/papers/covcon-again.pdf.

Giuseppe Castagna. 2023. Programming with union, intersection, and negation types. In *The French School of Programming*, Bertrand Meyer (Ed.). Springer. ISBN 978-3-031-34517-3. Preprint at arXiv:2111.03354.

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. *The Design Principles of the Elixir Type System.* Technical Report. Arxiv. arXiv:2306.06391 [cs.PL]

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming,* pages 198-208, ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science n. 3580, pages 30-34, Springer (summary). Lisboa, Portugal. https://doi.org/10.1145/1069774.1069793 Joint ICALP-PPDP keynote talk.

Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. 2022. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (Jan. 2022), 31 pages. https://doi.org/10.1145/3498674

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. https://doi.org/10.1145/2676726.2676991

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14)*. 5–17. https://doi.org/10.1145/2676726.2676991

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*. 94–106. https://doi.org/10.1145/2034773.2034788

Adam Chlipala. 2010. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/1806596.1806612

Elixir. *Elixir Typespecs.* https://hexdocs.pm/elixir/typespecs.html

Erlang. *Erlang Reference Manual User's Guide.* Ericsson AB. https://www.erlang.org/doc/reference_manual/users_guide.html

F#. *F# documentation.* https://learn.microsoft.com/en-us/dotnet/fsharp/

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* Ph. D. Dissertation. Université Paris Diderot. http://www.cduce.org/papers/frisch_phd.pdf

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 137–146. https://doi.org/10.1109/LICS.2002.1029823

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM* 55, 4 (Sept. 2008), 19:1–19:64. https://doi.org/10.1145/1391289.1391293

Go. *The Go Programming Language Specification.* https://go.dev/ref/spec

Hive. *Apache Hive Language Manual.* https://cwiki.apache.org/confluence/display/Hive/LanguageManual

C. A. R. Hoare. 1965. Record Handling. *ALGOL Bull.* 21 (nov 1965), 39–69.

C. A. R. Hoare. 1966a. Further Thoughts on Record Handling. *ALGOL Bull.* 23 (mar 1966), 5–11.

C. A. R. Hoare. 1966b. Record Handling. Lecture Notes, NATO Summer School.

Lalita Jategaonkar and John Mitchell. 1988. ML with Extended Pattern Matching and Subtypes. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. Association for Computing Machinery, New York, NY, USA, 198–211. https://doi.org/10.1145/62678.62702

Alan Jeffrey. 2022. Semantic Subtyping in Luau. Blog post. https://blog.roblox.com/2022/11/semantic-subtyping-luau Accessed on May 6th 2023.

Julia. *Julia Documentation*. https://docs.julialang.org/

Tobias Lindahl and Konstantinos Sagonas. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 91–106.

Lua. *Lua Reference Manual*. https://www.lua.org/manual

Luau. *Luau*. https://luau-lang.org/

OCaml. *Learn OCaml*. https://ocaml.org/docs

François Pottier. 2000. A Versatile Constraint-Based Type Inference System. *Nordic J. of Computing* 7, 4 (dec 2000), 312–347.

Didier Rémy. 1989. Type Checking Records and Variants in a Natural Extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/75277.75284

Didier Rémy. 1995. A case study of typechecking with constrained types: Typing record concatenation. (August 1995). Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK.

Ruby. *What is Ruby?* https://ruby-doc.org/3.2.1

Rust. *The Rust Reference*. https://doc.rust-lang.org/stable/reference/

Scala. *The Scala Programming Language*. https://docs.scala-lang.org/

Swift. *The Swift Programming Language Book*. https://www.swift.org/documentation/

Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 37–44.

Mitch Wand. 1989. Type inference for record concatenation and multiple inheritance. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 92–97. https://doi.org/10.1109/LICS.1989.39162

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. *Proc. ACM Program. Lang.* 7, POPL, Article 31 (jan 2023), 28 pages. https://doi.org/10.1145/3571224

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (oct 2018), 27 pages. https://doi.org/10.1145/3276483

APPENDIX

A SUBTYPING DECOMPOSITION FOR ARROWS

The intersection of arrows in (1) is empty if and only if there exists $t'_1 \to t'_2 \in N$ such that $t'_1 \leq \bigvee_{t_1 \to t_2 \in P} t_1$ and for all $P' \subsetneq P$ (notice that the containment relation is strict)

$$\left(t'_1 \leq \bigvee_{t_1 \to t_2 \in P'} t_1 \right) \text{ or } \left(\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \leq t'_2 \right) \tag{24}$$

The correctness of this decomposition is proved by Frisch [2004, Section 4.4].

B TYPE SOUNDNESS

The type soundness of the system in Section 4 can be proved by a routine extension of the inductive proofs of type preservation and progress of the system of Section 3.

Type preservation is proved by induction on the derivation of the type of the reducendum (cf. [Frisch et al. 2008, Theorem 5.1]) and a case analysis on the last applied rule. Let us outline how to extend the proof for the new selection operation introduced in Section 4. If we have that the derivation of $\Gamma \vdash e : t$ ends by [M-Sel] and $e \rightsquigarrow e'$, then we can deduce that $\Gamma \vdash e' : t$:

- if e' is obtained from e as a context reduction, then the thesis follows from the induction hypothesis and the monotony of the type projection operator $\boldsymbol{\pi}_\ell(\_)$ and thus of the map-selection type operator $\_.[\_]$;
- if e' is obtained by (8), then $e \equiv \{\ell_1 = v_1, \ldots, \ell_n = v_n\}.[\ell']$ and $e' \equiv v_k$ if $\ell' \equiv \ell_k$ for some $k \in [1..n]$ and $e' \equiv \mathtt{nil}$ otherwise. Since e is well typed, then $\Gamma \vdash \{\ell_1 = v_1, \ldots, \ell_n = v_n\} : t_1$, $\Gamma \vdash \ell' : t_2$, and $t = t_1.[t_2]$. By inversion, we have that $v_i : \boldsymbol{\pi}_{\ell_i}(t_1)$ for $i = 1..n$ and that $\boldsymbol{\pi}_\ell(t_1)$ is either $\bot$ or $\bot \vee \mathbb{1}$ for $\ell \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}$. Since $\ell' : t_2$, then $\boldsymbol{\pi}_{\ell'}(t_1) \leq \bigvee_{\ell \in t_2} \boldsymbol{\pi}_\ell(t_1)$. By definition of $t_1.[t_2]$, if $\ell' \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}$, then $\bot \leq \bigvee_{\ell \in t_2} \boldsymbol{\pi}_\ell(t_1)$ and therefore $\mathtt{nil} \leq t$ and the thesis follows from $e' \equiv \mathtt{nil}$. Otherwise $e' \equiv v_k : \boldsymbol{\pi}_{\ell_k}(t_1) \leq (\bigvee_{\ell \in t_2} \boldsymbol{\pi}_\ell(t_1) \setminus \bot) \leq t$.

For progress, we prove that if e is closed and $\vdash e : t$ then either e is a value or it can be reduced. The proof is an extension of the proof of [Frisch et al. 2008, Theorem 5.12] and is done by induction on the derivation of $\vdash e : t$. The cases for the rules [M-Sel] and [M-Del] are both straightforward and follow the pattern of the rule for pair projection in the proof of [Frisch et al. 2008, Theorem 5.12].

C MAP CONTAINMENT

C.1 Proof of Lemma 4.7

We consider two cases. If the cardinality of $\mathbb{K}$ is 1, then it means that $\mathbb{K} = \{\_\}$, that is, $\mathbb{K}$ is the singleton that contains only the wildcard that represents $\mathcal{L}$. In this case it is immediate to see that the statement of the Lemma coincides with the one of Lemma 2.3 and hence it holds.

If the cardinality of $\mathbb{K}$ is strictly greater than 1 then we have $\mathbb{K} = \{k_1, \ldots, k_m\}$ with $m > 1$. We can see a quasi $\mathbb{K}$-step function $r : \mathcal{L} \to Z$ on $\mathbb{K}$ as a finite product. Let $L \xrightarrow{c} Z$ denote the set of *constant* functions from L to Z, that is the set of all functions $f : L \to Z$, such that $f(\ell) = f(\ell')$ for all $\ell, \ell' \in L$. A quasi $\mathbb{K}$-step function r in $\mathcal{L} \xrightarrow{\mathbb{K}} Z$ can then be identified as an element of the following product

$$\left(\prod_{\ell \in \mathrm{dom}(r)} Z \right) \times ((k_1 \setminus \mathrm{dom}(r)) \xrightarrow{c} Z) \times \ldots \times ((k_m \setminus \mathrm{dom}(r)) \xrightarrow{c} Z)$$

For instance the function $\{\!\{\ell_1 = z_1, \ldots, \ell_n = z_n\,,\ k_1 = z'_1, \ldots, k_m = z'_m\}\!\}$ corresponds to the product $(\prod_{\ell \in \{\ell_1, \ldots, \ell_n\}} f(\ell)) \times r_1 \times \ldots \times r_m$ where $f(\ell_i) = z_i$ (for $i = 1..n$) and r_i is the constant function that

maps all labels in $k_i \setminus \{\ell_1, ..., \ell_n\}$ into z'_i. Notice that this holds true for any quasi $\mathbb{K}$-step function whose domain is contained in $\{\ell_1, ..., \ell_n\}$. In other words, given a finite set $L \subset \mathcal{L}$ every quasi $\mathbb{K}$-step function $r : \mathcal{L} \xrightarrow{\mathbb{K}} Z$ for which $\mathrm{dom}(r) \subseteq L$ holds, can be identified to an element of the product

$$(\prod_{\ell \in L} Z) \times ((k_1 \setminus L) \xrightarrow{c} Z) \times ... \times ((k_m \setminus L) \xrightarrow{c} Z)$$

We have thus reduced the problem of inclusion in the statement of the lemma to the problem of inclusion between an intersection and a union of finite Cartesian products of the form above. Formula (2) solves the problem for products of arity 2, by considering all partitions of the set N over the two projections of the products (N' for the first and $N \setminus N'$ for the second). The formula in the statement of the lemma generalizes formula (2) to products of arity n (where n is the cardinality of $L \cup \mathbb{K}$) by considering all partitions of N over the set $L \cup \mathbb{K}$ (i.e., all mappings ι in $N \to (L \cup \mathbb{K})$: see Appendix D for an explanation). For the first components of the product, it applies the same check as in (2), which yields the

$$\left(\exists \ell \in \mathcal{L}.\left(\bigcap_{p \in P} X_p(\ell) \subseteq \bigcup_{n \in N \mid \iota(n) = \ell} X_n(\ell)\right)\right)$$

part of the formula. For the last m components it applies a different formula which checks the inclusion between an intersection and a union of function spaces all with the same domain; in particular, for a given ι it checks whether there exists $k \in \mathbb{K}$ such that

$$\bigcap_{p \in P}((k \setminus L) \to \mathrm{def}(X_p)_k) \;\subseteq\; \bigcup_{n \in N_j}((k \setminus L) \to \mathrm{def}(X_n)_k)$$

where $N_k = \{n \in N \mid \iota(n) = k\}$. We can thus apply the decomposition rule for arrows as defined by Frisch et al. [2008, Section 6.2] (i.e., formula (24) in Appendix A) which, since in this case all the arrows have the same domain $k \setminus L$, becomes:

$$\exists n_\circ \in N_k \,.\, \forall P' \subsetneq P \,.\left((k \setminus L) \;\subseteq\; \bigcup_{p \in P'}(k \setminus L)\right) \text{ or } \left(\bigcap_{p \in P \setminus P'} \mathrm{def}(X_p)_k \subseteq \mathrm{def}(X_{n_\circ})_k\right)$$

Also notice that the left-hand side of the or's above all have the form $(k \setminus L) \subseteq \bigcup_{p \in P'}(k \setminus L)$, which always holds except when $P' = \emptyset$, for which the containment is false (since it is equivalent to requiring $k \setminus L$ to be empty which is impossible since k is infinite and L is always finite). Thus, in the formula above what needs to be checked is only the right-hand side of the or for $P' = \emptyset$, that is, $\exists n_\circ \in N_k \,.\, \bigcap_{p \in P} \mathrm{def}(X_p)_k \subseteq \mathrm{def}(X_{n_\circ})_k$ which, by the definition of N_k, yields the second part of the formula of the statement, that is:

$$\left(\exists n_\circ \in N, k \in \mathbb{K}.\left(\iota(n_\circ) = k\right) \text{ and } \left(\bigcap_{p \in P} \mathrm{def}(X_p)_k \leq \mathrm{def}(X_{n_\circ})_k\right)\right)$$

D TUPLE SUBTYPING

In this section we are going to show how to generalize the subtyping decomposition (2) for the case of tuples. Let us first show how it works with triplets. In particular, we want to determine when

$$\bigwedge_{(s_1,\, s_2,\, s_3) \in P} (s_1\,,\, s_2\,,\, s_3) \;\wedge\; \bigwedge_{(t_1,\, t_2,\, t_3) \in N} \neg(t_1\,,\, t_2\,,\, t_3) \tag{25}$$

is empty. The generalization of formula (2) is then defined as follows. The type in (25) is empty if and only if:

For all *pairwise disjoint* (and possibly empty) sets N_1, N_2, N_3 such that $N_1 \cup N_2 \cup N_3 = N$

$$\left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_1 \;\leq\; \bigvee_{(t_1,\,t_2,\,t_3)\in N_1} t_1\right) \text{ or } \left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_2 \;\leq\; \bigvee_{(t_1,\,t_2,\,t_3)\in N_2} t_2\right) \text{ or } \left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_3 \;\leq\; \bigvee_{(t_1,\,t_2,\,t_3)\in N_3} t_3\right) \tag{26}$$

holds. Why? In a nutshell because

$$\neg(t_1\,,\,t_2\,,\,t_3) \wedge (\mathbb{1}\,,\,\mathbb{1}\,,\,\mathbb{1}) = (\neg t_1\,,\,\mathbb{1}\,,\,\mathbb{1}) \vee (\mathbb{1}\,,\,\neg t_2\,,\,\mathbb{1}) \vee (\mathbb{1}\,,\,\mathbb{1}\,,\,\neg t_3) \tag{27}$$

and therefore

$$\bigwedge_{(t_1,\,t_2,\,t_3)\in N} \neg(t_1\,,\,t_2\,,\,t_3) \tag{28}$$

becomes an intersection of the union of three different sets: we distribute intersections over unions and factorize the summands that have $\mathbb{1}$ in the same position. More precisely, we just consider the case in which P is not empty (otherwise the emptiness of (26) cannot hold) and that all s_i's are not empty (otherwise the emptiness of (26) is trivial). Since we are intersecting the negated products with at least one product then we can also consider just their product parts, whence the intersection with $(\mathbb{1}\,,\,\mathbb{1}\,,\,\mathbb{1})$ in (27).

We want to expand (28) into a union by distributing intersections over the unions shown in (27). For that it may be simpler to think of intersections as multiplications and unions as sums. We are thus trying to expand a multiplication of the form

$$\prod_{i\in N}(a_i + b_i + c_i)$$

where a, b and c have the form $(\neg t_1\,,\,\mathbb{1}\,,\,\mathbb{1})$, $(\mathbb{1}\,,\,\neg t_2\,,\,\mathbb{1})$, and $(\mathbb{1}\,,\,\mathbb{1}\,,\,\neg t_3)$ respectively.

Now if the cardinality of N is, say 2, we have

$$(a_1 + b_1 + c_1)(a_2 + b_2 + c_2) = a_1a_2 + a_1b_2 + a_1c_2 + b_1a_2 + b_1b_2 + b_1c_2 + c_1a_2 + c_1b_2 + c_1c_2$$

That is, it is the sum of all products of 2 factors obtained by choosing either a_1 or b_1 or c_1 for the first and a_2 or b_2 or c_2 for the second. Since we are working with positive numbers we have that the sum is zero if and only if all products are zero, that is, for all mapping $h : 1, 2 \to \{a, b, c\}$ we must have $h(1)_1h(2)_2 = 0$. If the cardinality of N is more than 2, say n, then we will obtain a sum of products of n factors, obtained by picking some summand in each sum in N. That is, if n is the cardinality of N

$$\prod_{i\in N}(a_i + b_i + c_i) = \sum_{h:\{1..n\}\to\{a,b,c\}} h(1)_1 \cdot \ldots \cdot h(n)_n$$

In other words we considered all partitions of N in three sets (that in formula (26) we called N_1, N_2, N_3) which tell for each element in N whether to take as factor the first, second, or third summand of the element.

In the specific case we are considering, the set in which a triple $(t_1\,,\,t_2\,,\,t_3)$ of N is mapped into, tells us whether we have to take as factor $(\neg t_1\,,\,\mathbb{1}\,,\,\mathbb{1})$, $(\mathbb{1}\,,\,\neg t_2\,,\,\mathbb{1})$, or $(\mathbb{1}\,,\,\mathbb{1}\,,\,\neg t_3)$. And clearly this multiplication (intersection) can be further simplified into a single set-theoretic product by intersecting the factors component-wise. That is, in our case the elements of the sum (union) will be of the following form:

$$\left(\bigwedge_{(t_1,\,t_2,\,t_3)\in N_1} \neg t_1 \quad,\quad \bigwedge_{(t_1,\,t_2,\,t_3)\in N_2} \neg t_2 \quad,\quad \bigwedge_{(t_1,\,t_2,\,t_3)\in N_3} \neg t_3\right)$$

therefore we have that (28) is equivalent to

$$\bigvee_{\forall N_1,N_2,N_3 \text{ partition of } N} \left(\bigwedge_{(t_1,\,t_2,\,t_3)\in N_1} \neg t_1 \quad , \quad \bigwedge_{(t_1,\,t_2,\,t_3)\in N_2} \neg t_2 \quad , \quad \bigwedge_{(t_1,\,t_2,\,t_3)\in N_3} \neg t_3 \right) \tag{29}$$

From this and from the observation that

$$\bigwedge_{(s_1,\,s_2,\,s_3)\in P} (s_1\,,\,s_2\,,\,s_3) = \left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_1\,,\quad \bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_2\,,\quad \bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_3 \right)$$

it is easy to deduce (26) (just distribute the outer intersection over the union (29) and check emptiness component-wise).

Finally, notice that what the formula (26) does is to generate all possible partitions of N into three sets.

Considering all possible partitions of N into three sets is equivalent to consider all possible (total) functions from N to $\{1,2,3\}$.

So an equivalent way to write (26) is the following:

For all $h : N \to \{1,2,3\}$

$$\left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_1 \;\leq \bigvee_{h((t_1,\,t_2,\,t_3))=1} t_1 \right) \text{or} \left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_2 \;\leq \bigvee_{h((t_1,\,t_2,\,t_3))=2} t_2 \right) \text{or} \left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_3 \;\leq \bigvee_{h((t_1,\,t_2,\,t_3))=3} t_3 \right) \tag{30}$$

holds (where the unions are on all $(t_1,\,t_2,\,t_3)$ such that $h((t_1,\,t_2,\,t_3)) = i$ for $i = 1, 2, 3$ ... sorry for the ugly abbreviation).

And again equivalently:

For all $h : N \to \{1,2,3\}$, there exists $i \in \{1,2,3\}$ such that:

$$\left(\bigwedge_{(s_1,\,s_2,\,s_3)\in P} s_i \;\leq \bigvee_{h((t_1,\,t_2,\,t_3))=i} t_i \right) \tag{31}$$

holds.

Now it is very easy to generalize the last formulation to the case of tuples of n arity. We have that

$$\bigwedge_{(s_1,\,\ldots,\,s_n)\in P} (s_1\,,\,\ldots,\,s_n) \;\wedge \bigwedge_{(t_1,\,\ldots,\,t_n)\in N} \neg(t_1\,,\,\ldots,\,t_n) \tag{32}$$

is empty if and only if for all $h : N \to [1..n]$, there exists $i \in [1..n]$ such that:

$$\left(\bigwedge_{(s_1,\,\ldots,\,s_n)\in P} s_i \;\leq \bigvee_{h((t_1,\,\ldots,\,t_n))=i} t_i \right) \tag{33}$$

holds.

E SOLUTION TO THE EXERCISE IN SECTION 6

Typing $\lambda x.\lambda y.(x \mathbin{+} \{\ell = y\})$ by $\forall \alpha, \beta\,.\,(\alpha \wedge \mathbf{\{\}}) \to \beta \to (\alpha \wedge \mathbf{\{}\ell = \beta\mathbf{\}})$ is unsound. If the function had this type, then the following application $(\lambda x.\lambda y.(x \mathbin{+} \{\ell = y\}))\,\{\ell = 42\}\,\mathsf{true}$ could be typed by instantiating α to $\mathbf{\{}\ell = \mathsf{Int}\mathbf{\}}$ and β to Bool. In this way, we would have deduced that the application has type $\mathbf{\{}\ell = \mathsf{Int}\mathbf{\}} \wedge \mathbf{\{}\ell = \mathsf{Bool}\mathbf{\}}$, that is, $\mathbb{0}$. Therefore, we would have deduced that the result of the application, $\{\ell = 42\}$, has type $\mathbb{0}$ and thus, by subsumption, any possible type.

F MUTATION

Records and mutations are often intertwined, especially in dynamic languages. In JavaScript, for instance, every field is highly mutable: it can be added, modified, erased. Some languages introduce a more controlled way of mutating: in Ballerina, lists and records are by default mutable but only within certain limits established by the type statically declared for them (fields can be declared to be readonly and there are ways to transform mutable fields into readonly ones and viceversa); in Julia, it is exactly the opposite since records (i.e., composite types) are immutable by default (even if their fields can contain mutable values) but they can be declared mutable. Some functional languages introduce mutation via records: in OCaml a value of type `ref int` is a record with a field `contents` that contains an integer, that is, a record of type `{contents: int}`; in CDuce a value of type `ref int` is also a record, but with two fields instead, `get` (to read the location) and `set` (to write the location), that is a record of type `{set: int -> (), get: () -> int}`, since this directly enforces the classic invariant subtyping relation for reference types (see below).

In this section we define a unified theory of mutable locations that encompasses such different type discipline as those defined for OCaml/CDuce references and those for Ballerina locations, and then embed them in the record/map theory of the previous sections to type the structures of Ballerina and similar languages. In particular, we define a unique framework in which we will be able to interpret both the classic invariant mutable reference types that can be found in Rust or CDuce and the mutable record types of Ballerina. These constitute two extreme points in the design space for subtyping mutable locations. On one end of this design spectrum, invariant references form a stricter discipline with stronger static safety guarantees, but they break subtyping insofar as invariance is the only possible subtyping relation. On the other end of the spectrum there is Ballerina, whose design principles require the subtyping relation to satisfy properties at first sight counter-intuitive for a functional programmer (immutable pointers can be used where mutable ones are expected, reference types are covariant, pattern matching disregards mutability). We will see the details of it later on.

F.1 A theory of locations

The most general way to implement the ideas above is to add *locations*. Here we develop a general theory of locations and then show how to use it to interpret Ballerina's mutable structures and classic invariant reference types.

We imagine a notion of location as a cell containing a value v and which comes with two sets: a set X_1 of values that can be read from it, and a set X_2 of values that can be written to it. These two sets are used to drive the semantics of writes into the cell. We can for instance design the operational semantics such that if we try to write a value v in it, it simply discards it if $v \notin X_1$; if $v \notin X_2$, then either we can make the runtime raise an exception (as in Ballerina or the JVM) or design the type system so that it statically ensures that such a situation will never happen (as in languages with invariant reference types like Rust or CDuce). When $X_2 = \emptyset$, then the location is readonly: we can read values from it but we cannot write any. The need of the two distinct sets is to avoid the paradoxes explained by Frisch et al. [2008, Appendix A].

If we interpret values as elements of a certain domain $\mathcal{D}$, then a location is interpreted as a triple $(d, X_1, X_2) \in \mathcal{D} \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})$. Next we define some types to capture some specific sets of locations. In what follows, we distinguish two different kinds of types: the classic invariant reference types (as formalized by Frisch et al. [2008, Appendix A.2]) and Ballerina's structures components (i.e., the content of Ballerina tuples, arrays, maps, records, and tables). In Ballerina, we will consider location types of the form $\mathbf{loc}(t, b)$ where t is a type and b is a flag set to 1 for

mutable locations and set to 0 for readonly locations. For classic invariant reference types we will use $\mathbf{ref}(t)$.

Let $X \subseteq \mathcal{D}$, we define the following operators that return a subset of $\mathcal{D} \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})$:

$$\mathbf{ref}(X) = \{(d, X_1, X_2) \mid d \in X_1 \subseteq X \subseteq X_2\}$$

$$\mathbf{loc}(X, 1) = \{(d, X_1, X_2) \mid d \in X, X_1 \cup X_2 \subseteq X\}$$
$$\mathbf{loc}(X, 0) = \{(d, \varnothing, \varnothing) \mid d \in X\}$$

Intuitively, we would like to define the interpretation of types so that $[\![\mathbf{ref}(t)]\!] = \mathbf{ref}([\![t]\!])$ and $[\![\mathbf{loc}(t, b)]\!] = \mathbf{loc}([\![t]\!], b)$. However, as it is the case for function spaces, for cardinality reasons it is not possible to have $\mathcal{D} \subseteq \mathcal{D} \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})$. Thus, we proceed as for function spaces and define a second interpretation $\mathbb{E}(.)$, called the extensional interpretation, and say that an interpretation is a model if and only if for all types t we have $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$ (see [Frisch et al. 2008] for details).

The extensional interpretations of types $\mathbf{loc}(t, b)$ and $\mathbf{ref}(t)$ are respectively defined as:

$$\mathbb{E}(\mathbf{ref}(t)) = \mathbf{ref}([\![t]\!])$$
$$\mathbb{E}(\mathbf{loc}(t, b)) = \mathbf{loc}([\![t]\!], b)$$

We use $\mathbb{1}_{\mathbf{loc}}$ to denote the type of all locations, that is the type whose interpretation is $[\![\mathbb{1}_{\mathbf{loc}}]\!] = \mathcal{D} \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D})$. As usual, every type that contains only locations—i.e., every subtype of $\mathbb{1}_{\mathbf{loc}}$—can be expressed as a disjunctive normal form, that is, a union of intersections of $\mathbf{loc}(.,.)$ atoms (respectively, $\mathbf{ref}(.)$ atoms) and of their negations. Determining subtyping then is equivalent to determining the emptiness of the intersections that form the union of a disjunctive normal form. The way to compute this is given by the two following lemmas:

Lemma F.1 ([Frisch et al. 2008]). *Let $(X_i)_{i \in P}$ and $(Y_j)_{j \in N}$ be two families of subsets of $\mathcal{D}$. Then:*

$$\bigcap_{i \in P} \mathbf{ref}(X_i) \subseteq \bigcup_{j \in N} \mathbf{ref}(Y_j)$$
$$\iff$$
$$\left(\bigcap_{i \in P} X_i = \emptyset\right) \text{ or } \left(\exists j \in N.\ \bigcap_{i \in P} X_i \subseteq Y_j \subseteq \bigcup_{i \in P} X_i\right)$$

Proof. The $\Leftarrow$ implication is straightforward. For the opposite direction, we assume that $\bigcap_{i \in P} \mathbf{ref}(X_i) \subseteq \bigcup_{j \in N} \mathbf{ref}(Y_j)$ and $\bigcap_{i \in P} X_i \neq \emptyset$. We define Z_1 as $\bigcap_{i \in P} X_i$ and Z_2 as $\bigcup_{i \in P} X_i$. We pick an element d from Z_1, which is not empty by hypothesis. The triple (d, Z_1, Z_2) is in $\bigcap_{i \in P} \mathbf{ref}(X_i)$, and thus, by hypothesis, also in $\bigcup_{j \in N} \mathbf{ref}(Y_j)$. This gives a j such that (d, Z_1, Z_2) is in $\mathbf{ref}(Y_j)$ and the rest of the proof follows easily. □

It is easy to see that for the above interpretation we have $\mathbf{ref}(t_1) \leq \mathbf{ref}(t_2) \Leftrightarrow (t_1 \leq \mathbb{0})$ or $(t_1 \simeq t_2)$: the interpretation generalizes the classic invariant subtyping for reference types to type system with empty type.[18]

Lemma F.2. *$(X_i)_{i \in P}$ and $(b_i)_{i \in P}$ be two families of subsets of $\mathcal{D}$ and of 0 or 1, respectively. Then:*

$$\bigcap_{i \in P} \mathbf{loc}(X_i, b_i) \simeq \mathbf{loc}(\textstyle\bigcap_{i \in P} X_i, \min_{i \in P} b_i)$$

[18] Formally, $\mathbf{ref}(t_1) \leq \mathbf{ref}(t_2) \Leftrightarrow [\![\mathbf{ref}(t_1)]\!] \subseteq [\![\mathbf{ref}(t_2)]\!] \Leftrightarrow [\![\mathbf{ref}(t_1) \wedge \neg \mathbf{ref}(t_2)]\!] = \varnothing \Leftrightarrow \mathbb{E}(\mathbf{ref}(t_1) \wedge \neg \mathbf{ref}(t_2)) = \varnothing \Leftrightarrow \mathbb{E}(\mathbf{ref}(t_1)) \cap (\mathcal{D} \setminus \mathbb{E}(\mathbf{ref}(t_2))) = \varnothing \Leftrightarrow \mathbf{ref}([\![t_1]\!]) \cap (\mathcal{D} \setminus \mathbf{ref}([\![t_2]\!])) = \varnothing \Leftrightarrow \mathbf{ref}([\![t_1]\!]) \subseteq \mathbf{ref}([\![t_2]\!]) \Leftrightarrow ([\![t_1]\!] = \varnothing)$ or $([\![t_1]\!] = [\![t_2]\!]) \Leftrightarrow (t_1 \leq \mathbb{0})$ or $(t_1 \simeq t_2)$.

Proof. Straightforward.

Lemma F.3. *Let $(X_i)_{i\in P}$ and $(Y_j)_{j\in N}$ be two families of subsets of $\mathcal{D}$ and $(b_i)_{i\in P}$ and $(b'_j)_{j\in N}$ two families of flags with value in $\{0,1\}$. Then:*

$$\bigcap_{i\in P} \mathbf{loc}(X_i, b_i) \subseteq \bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$$
$$\Longleftrightarrow$$
$$\left(\bigcap_{i\in P} X_i = \emptyset\right) \text{ or } \left(\min_{i\in P} b_i = 0 \text{ and } \bigcap_{i\in P} X_i \subseteq \bigcup_{j\in N} Y_j\right) \text{ or } \left(\exists j \in N.\ \bigcap_{i\in P} X_i \subseteq Y_j \text{ and } b'_j = 1\right)$$

Proof. The $\Leftarrow$ implication is straightforward.

For the opposite direction, it is immediate to deduce from Lemma F.2 that $\bigcap_{i\in P} X_i = \emptyset$ implies $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i) = \emptyset$ and, thus, the result. Therefore, we assume that $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i) \subseteq \bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$ and $\bigcap_{i\in P} X_i \neq \emptyset$. By Lemma F.2 we can distinguish two cases.

Either $\min_{i\in P} b_i = 0$, in which case we know that all elements of $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i)$ have the form $(d, \varnothing, \varnothing)$ with $d \in \bigcap_{i\in P} X_i$. By hypothesis $(d, \varnothing, \varnothing) \in \bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$. Now it is easy to see that an element of the form $(d, \varnothing, \varnothing)$ belongs to $\mathbf{loc}(Y_j, b'_j)$ if and only if $d \in Y_j$. Therefore, we deduce that for every $d \in \bigcap_{i\in P} X_i$ there exists $j \in N$ such that $d \in Y_j$ and we conclude that $\bigcap_{i\in P} X_i \subseteq \bigcup_{j\in N} Y_j$.

Or $\min_{i\in P} b_i = 1$. In that case we define Z as $\bigcap_{i\in P} X_i$ and pick an element d from Z which is not empty by hypothesis. Then the triple (d, Z, Z) is in $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i)$, and thus, by hypothesis, also in $\bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$ which gives a j such that (d, Z, Z) is in $\mathbf{loc}(Y_j, b'_j)$, with $b'_j = 1$ since Z is not empty.

□

As before, it is easy to see that by the lemma above we have $t_1 \leq t_2 \Leftrightarrow \mathbf{loc}(t_1, 0) \leq \mathbf{loc}(t_2, 0) \Leftrightarrow \mathbf{loc}(t_1, 0) \leq \mathbf{loc}(t_2, 1) \Leftrightarrow \mathbf{loc}(t_1, 1) \leq \mathbf{loc}(t_2, 1)$. Therefore, in Ballerina, locations are covariant and readonly locations can be used where writable ones are expected. Both facts may seem at first surprising, but we explain their rationale in Section F.1.1. More generally we have, $\mathbf{loc}(t_1, b_1) \leq \mathbf{loc}(t_2, b_2) \Leftrightarrow (t_1 \leq \mathbb{0})$ or $(t_1 \leq t_2$ and $b_1 \leq b_2)$.

To handle locations we extend our calculus with the following constructions:

$$e ::= \ \ldots \mid !e \mid (e := e) \mid \mathtt{cell}_{t_1,t_2} e$$
$$v ::= \ \ldots \mid \mathtt{cell}_{t_1,t_2} v$$

The expressions $!e$ and $(e := e)$ are for dereferencing and assignment, while $\mathtt{cell}_{t_1,t_2} e$ is an expression that defines a cell that contains the value resulting from the evaluation of e and that declares to return values of type t_1 for reading and to accept values of type t_2 for writing. The intuition for the corresponding cell *value* is the one that we explained before, that is, the value $\mathtt{cell}_{t_1,t_2} v$ is a cell in that contains a value v, which refuses write operations of values not in t_2, and which silently discards writes of values not in t_1. The expression $!e$ returns the value stored in the cell returned by e; the expression $e_1 := e_2$ stores the value returned by e_2 in the cell $\mathtt{cell}_{t_1,t_2} v$ returned by e_2 provided that this value is both of type t_1 (otherwise it silently discards it), and of type t_2 (otherwise its semantics is undefined). This is formalized by the following definition of the operational semantics, which rewrites a pair formed by an expression e and a store ς (i.e., a mapping from memory locations—ranged over by μ— to values) into a new pair and where we use

() as result for assignments:

$$\begin{array}{rcll}
\mathtt{cell}_{t_1,t_2}v;\varsigma & \rightsquigarrow & \mu;\varsigma\{\mu \mapsto \mathtt{cell}_{t_1,t_2}v\} & \text{with } \mu \notin \mathrm{dom}(\varsigma) \\
!\mu;\varsigma & \rightsquigarrow & v;\varsigma & \text{if } \varsigma(\mu) = \mathtt{cell}_{t_1,t_2}v \\
\mu := v;\varsigma & \rightsquigarrow & ();\varsigma\{\mu \mapsto \mathtt{cell}_{t_1,t_2}v\} & \text{if } \varsigma(\mu) = \mathtt{cell}_{t_1,t_2}v' \text{ and } v : t_1 \vee t_2 \\
\mu := v;\varsigma & \rightsquigarrow & ();\varsigma & \text{if } \varsigma(\mu) = \mathtt{cell}_{t_1,t_2}v' \text{ and } v : t_2 \setminus t_1
\end{array}$$

Notice that the semantics of writing a value v into a cell $\mathtt{cell}_{t_1,t_2}v'$ is undefined when v is not of type t_2. In general, it should be the task of the type system to statically ensure that such a situation cannot happen, but in Ballerina this is accepted: the validity of each write is dynamically checked, and any violation generates an error, that is, roughly:

$$\mu := v;\varsigma \quad \rightsquigarrow \quad \mathsf{error};\varsigma \qquad \text{if } \varsigma(\mu) = \mathtt{cell}_{t_1,t_2}v' \text{ and } v : \neg t_2$$

The semantic intuition underlying the definition of the new values is that, if v denotes some element d, then $\mathtt{cell}_{t_1,t_2}v$ denotes the element $(d, [\![t_1]\!], [\![t_2]\!])$.

The definition of the typing rules depends on the kind of locations we are working with.

Let us start with the classic reference types. The first problem to solve is to define the typing of the cell expressions. Intuitively, we would like to have the following rule:

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t \leq t_2}{\Gamma \vdash \mathtt{cell}_{t_1,t_2}e : \mathbf{ref}(t)}$$

that is, the expression e must return a value that is in the "read-type" t_1, in which case the expression can be considered to be of type $\mathbf{ref}(t)$ as long as we have $t_1 \leq t \leq t_2$. This rule allows the system to deduce for the same expression different reference types (i.e., all reference types whose type is included between the type for reads and the type for writes of the cell expression). The intersection rule, then deduces for the expression the intersection of the different types. In semantic subtyping the intersection rule is not included in the system: it is an admissible rule, thanks to the fact that intersections are accounted for in introduction rules (notably, when typing λ-abstractions). Not only: the theoretical framework also wants that the expression $\mathtt{cell}_{t_1,t_2}e$ has type $\mathbf{ref}(t)$ if *and only if* $t_1 \leq t \leq t_2$; otherwise, following the semantic subtyping approach with function types, it should have type $\neg\mathbf{ref}(t)$. As a consequence, in order to preserve the admissibility of the intersection rule, the rule for cells with classic reference types is:

$$\frac{\Gamma \vdash e : t_1 \quad \forall i = 1..n.\ t_1 \leq s_i \leq t_2 \quad \forall j = 1..m.\ \neg(t_1 \leq s'_j \leq t_2)}{\Gamma \vdash \mathtt{cell}_{t_1,t_2}e : \wedge_{i=1..n}\mathbf{ref}(s_i) \wedge \wedge_{j=1..m}\neg\mathbf{ref}(s'_j)}$$

The interest of this rule is mainly theoretical (i.e., it is needed to ensure that the model of values is a model, that is, that for every value $\vdash v : t \iff \nvdash v : \neg t$: see [Frisch et al. 2008, Theorem 5.5]). We give its practical version later in the section.

The rules for dereferencing and assignment are, instead, much simpler. They are the classic rules that are used in practice, that is:

$$\frac{\Gamma \vdash e : \mathbf{ref}(t)}{\Gamma \vdash !e : t} \qquad \frac{\Gamma \vdash e_1 : \mathbf{ref}(t) \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1 := e_2) : ()}$$

For the case of Ballerina's location types, we proceed as we did for the classic reference types and follow semantic intuition to define *two* rules for cells that take into account whether the location is readonly or not

$$\frac{\Gamma \vdash e : t \qquad t_1 \vee t_2 \leq t}{\Gamma \vdash \mathtt{cell}_{t_1,t_2}e : \mathbf{loc}(t,1)} \qquad \frac{\Gamma \vdash e : t \qquad t_1 \vee t_2 \leq \mathbb{0}}{\Gamma \vdash \mathtt{cell}_{t_1,t_2}e : \mathbf{loc}(t,0)}$$

Once more it is possible to deduce different types for the same expression and therefore the two rules above should be synthesized into a unique rule that preserves the admissibility of the intersection rule and allows the system to deduce negated location types. Unfortunately, contrary to the case with **ref**() types, it is not possible to derive the negative part of such a rule by negating the two rules above: this would correspond to deduce that an expression e does not have a given type t. The consequence of this fact is only of theoretical interest: we do not know whether the model of values is a set-theoretic model. In practice, however, the two rules above are not only sufficient, but even too general, and they should be restricted as we show below.

The rules for dereferencing and for assignment are almost the same as for classic reference types, the only difference being that assignments are permitted only for writable locations:

$$\frac{\Gamma \vdash e : \mathbf{loc}(t, b)}{\Gamma \vdash !e : t} \qquad \frac{\Gamma \vdash e_1 : \mathbf{loc}(t, 1) \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1 := e_2) : ()}$$

However, if working in the language Ballerina there will not be explicit deref and update operations as the above, since deref and updates are embedded directly in the operations of record read and update:

$$\frac{\Gamma \vdash e : \mathbf{\{}\ell = \mathbf{loc}(t, b)\mathbf{\}}}{\Gamma \vdash e.\ell : t} \qquad \frac{\Gamma \vdash e_1 : \mathbf{\{}\ell = \mathbf{loc}(t, 1)\mathbf{\}} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1.\ell = e_2) : ()}$$

that is, reading the field ℓ of a record requires that the record contains *at least* a field for it (the expression is typed by an open record), and returns the corresponding type; likewise for update, with the further constraint that the corresponding field must be a writable location. Similar rules must be defined for tuples, arrays, maps, and lists, since in Ballerina their elements are locations.

Even though in Ballerina locations appear as subcomponents of particular structures, the advantage of defining a type system by using explicit location types instead of embedding them directly in the type of the structures at issue such as record types, is that one can use for subtyping record types directly the formula (19) as is. The only difference will be that records (and tuples and array ...) map labels into *location* types, that is, into unions of intersections of $\mathbf{loc}(.,.)$ atoms and their negations.

We just established the theoretical framework to subtype and type-check location expressions. To avoid the paradoxes pointed out by Frisch et al. [2008, Appendix A] these come equipped with two distinct read and write types which are checked at run-time to silently discard values used in assignments or even to fail. Now, it is clear that in a practical setting we do not want assignments to silently discard the passed values, therefore in practice we only allow cells in which both types coincide, so that writing a forbidden value or writing a silently discarded one will have exactly the same effect. This corresponds to restricting the syntax of a programming language so that only cell expressions of the form $\texttt{cell}_{t,t}e$ are permitted. Under such a restriction the typing rules for cells specialize as follows:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{cell}_{t,t}e : \mathbf{ref}(t)} \qquad \frac{\Gamma \vdash e : t \qquad s \leq t}{\Gamma \vdash \texttt{cell}_{s,s}e : \mathbf{loc}(t, 1)} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{cell}_{\mathbb{0},\mathbb{0}}e : \mathbf{loc}(t, 0)}$$

Implementation-wise the duplication of the types in a cell expression becomes useless, and therefore in practice cell expressions will be a pair formed by an expression and just the type of allowed writes, e.g., $\texttt{cell}_t e$ rather than $\texttt{cell}_{t,t}e$, with $\texttt{cell}_{\mathbb{0}}e$ denoting readonly cells.

F.1.1 Rationale. The rationale of the two kinds of types is easy to explain. The use of **ref**(.) types corresponds to the one that can be found in a statically-typed functional languages, where the type system must ensure that there will be no stuck expression and, therefore, that all read and write

operations will be well-typed. This yields to a generalization of the classic invariant subtyping rule for reference types, since, as we already noticed: $\mathbf{ref}(t_1) \leq \mathbf{ref}(t_2) \Leftrightarrow (t_1 \leq \mathbb{0})$ or $(t_1 \simeq t_2)$.

The aims of the types of Ballerina are different and essentially twofold. The first point is that, for its use-cases Ballerina needs covariant subtyping for mutable types, even though this may cause run-time exceptions for write operations. But this is ok since in Ballerina philosophy, only reads are safe—the type system statically ensures that every read operation returns only values of the expected type—while writes are checked at runtime and can raise exceptions. As we have already seen, our interpretation of $\mathbf{loc}(.,.)$ is covariant on the type component, since $\mathbf{loc}(t_1, b_1) \leq \mathbf{loc}(t_2, b_2) \Leftrightarrow (t_1 \leq \mathbb{0})$ or $(t_1 \leq t_2$ and $b_1 \leq b_2)$. The second reason is that Ballerina uses concurrency and for that it is useful to guarantee that some data structures shared by different threads cannot be mutated, so as to avoid the use of locks to access them. For that Ballerina relies on a specific basic type `readonly` that is used to intersect other types to ensure that the values of the intersection cannot be mutated. If a function parameter is declared as readonly, that means that the caller is guaranteeing that the value it passed as an argument can never be mutated: thus we want to disallow read-and-write data to be used where readonly data is expected, or this would disrupt this guarantee. Ballerina requires locations to satisfy two further properties. First it must be possible to use readonly data structures where their read-and-write counterpart is expected: as a matter of fact in both cases write may fail at run-time, either because the value to be written is not compatible with the type expected by the cell (write of a read-and-write cell) or because *no* value can be written in the cell (write of a read-only cell). Thus, in this framework a read-only cell is exactly a read-and-write cell writable with values of the empty type.[19] This is the meaning of the covariance of the writable bits b_1 and b_2 in the example above. Second, from an operational point of view read-only cells are indistinguishable from their unboxed counterpart: it is impossible to write a program that distinguishes whether the values it is using are stored in a readonly cell or not. Therefore, we want that read-only types satisfy the same subtyping relations as the types they box. In particular, we want the following equality to hold:

$$\mathbf{loc}(t_1, 0) \vee \mathbf{loc}(t_2, 0) \quad \simeq \quad \mathbf{loc}(t_1 \vee t_2, 0)$$

so that it is not possible to distinguish the union of the values of two types from the union of the readonly locations containing the values of these two types. This is guaranteed by our interpretation. Notice instead that for read-and-write value while

$$\mathbf{loc}(t_1, 1) \vee \mathbf{loc}(t_2, 1) \quad \leq \quad \mathbf{loc}(t_1 \vee t_2, 1)$$

holds, the converse in general does not: for instance, $v = \mathtt{cell}_{\mathsf{Int}\vee\mathsf{Bool},\mathsf{Int}\vee\mathsf{Bool}}42$ is in $\mathbf{loc}(\mathsf{Int}\vee\mathsf{Bool}, 1)$ but not in $\mathbf{loc}(\mathsf{Int}, 1)\vee\mathbf{loc}(\mathsf{Bool}, 1)$; if we in v first an integer value then a Boolean one, both operations will be successful while for values in $\mathbf{loc}(\mathsf{Int}\vee\mathsf{Bool}, 1)$ one of the two must fail.

In practice, in our system we do not need to add the `readonly` type of Ballerina, since this can be encoded as the recursive type which is the union of all basic types, all functions, and all tuples and record with readonly subcomponents.

$$\mathbf{type}\ X = \mathbb{1}_{\mathbf{basic}} \vee (\mathbf{loc}(X, 0) \times \mathbf{loc}(X, 0)) \vee \mathbf{\{\_ \Rightarrow loc}(X, 0)\mathbf{\}} \vee \mathbb{0} \to \mathbb{1}$$

In Ballerina syntax this roughly corresponds to:[20]

[19] More pragmatically, mutability is the default behavior in Ballerina, and the language designers did not want to require the programmer to write specific functions for data that may happen to be read-only so that, unless this is necessary to ensure specific properties (such as immutability of shared data), all functions are written for mutable data.

[20] Keywords are in boldface. In particular **`readonly`** is the keyword Ballerina uses to declare a subcomponent of a structure (e.g., a record field) to be readonly, while `readonly` is the name of the type that is being defined.

```
type X = () | int | boolean | string | float | decimal
       | empty -> any
       | [ (readonly X...) ]
       | { (readonly X...) }

type readonly = X
```

(keywords are in boldface) the last summand of the definition of `X` meaning records with only readonly optional fields that include readonly values (Ballerina uses the syntax "...." to denote open record types). In other terms, all fields are declared readonly and are optional fields whose content is the recursion variable for readonly types. Likewise for tuples.

F.1.2 Universal model. To define a universal model it suffices to use the same technique as for function spaces and consider finite powersets for Ballerina locations and cofinite powersets for classic invariant references. That is.

$$\begin{aligned}
\mathbf{loc}(X,1) &= \{(d,X_1,X_2) \mid d \in X, X_1 \cup X_2 \subseteq X\} &\subseteq\quad & \mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{fin}}(\mathcal{D})\\
\mathbf{loc}(X,0) &= \{(d,\varnothing,\varnothing) \mid d \in X\} &\subseteq\quad & \mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{fin}}(\mathcal{D})\\
\mathbf{ref}(X) &= \{(d,X_1,X_2) \mid d \in X_1 \subseteq X \subseteq X_2\} &\subseteq\quad & \mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{cofin}}(\mathcal{D})
\end{aligned}$$

As before the elements of $\mathcal{D}$—which now is defined as $\mathcal{D} = \mathcal{B} + \mathcal{D}\times\mathcal{D} + \mathcal{P}_{\text{fin}}(\mathcal{D}\times\mathcal{D}_\Omega) + (\mathcal{L} \rightharpoonup \mathcal{D}_\perp) + (\mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times(\mathcal{P}_{\text{fin}}(\mathcal{D})\cup\mathcal{P}_{\text{cofin}}(\mathcal{D})))$—can be defined inductively, by adding the production $(d,\{d_1,...,d_m\},\{d_1,....,d_n\})$ to model Ballerina locations and $(d,\{d_1,...,d_m\},\mathcal{D}\setminus\{d_1,....,d_n\})$ for invariant reference types.

The definition of the predicate $(d : t)$ is extended as follows:

$$\begin{aligned}
(\ (d,\{d_1,...,d_m\},\mathcal{D}\setminus\{d'_1,....,d'_n\}) : \mathbf{ref}(t)\ ) &= (d:t) \wedge \forall i.(d_i:t) \wedge \forall j.\neg(d'_j:t)\\
(\ (d,\{d_1,...,d_m\},\{d_{m+1},....,d_{m+n}\}) : \mathbf{loc}(t,1)\ ) &= (d:t) \wedge \forall i.(d_i:t)\\
(\ (d,\{\},\{\}) : \mathbf{loc}(t,0)\ ) &= (d:t)
\end{aligned}$$

From a formal point of view this means that in this model we have two distinct kind of locations, those typed by $\mathbf{ref}(.)$ and those typed by $\mathbf{loc}(.,.)$. This however is needed just for defining the subtyping relation and thus is it not a problem to use the same values for both types, which makes it possible to use in a same language both kinds of location types used to classify the same set of values.

Although in the formal system we just presented, cells with loc-types are first class values, in a practical setting it is better to avoid such a scenario, for the simple reason that, as we already pointed out, it is not possible to observationally distinguish (i.e, to find a context that tells apart) a readonly location containing some value v from the value v itself. So in a practical setting one should avoid this confusion, either by boxing all values into locations (i.e., all the values used in the language are stored in locations), or to limit the use of locations as subcomponents of particular structures (i.e., all elements of records, tuples, list, ... are locations). The latter approach corresponds to considering the following model (without the cofinite part): $\mathcal{D} = \mathcal{B}+\mathcal{D}\times\mathcal{D}+\mathcal{P}_{\text{fin}}(\mathcal{D}\times\mathcal{D}_\Omega)+(\mathcal{L} \rightharpoonup (\mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{fin}}(\mathcal{D}))_\perp)$. In words, records map labels either to locations or to the undefined value, and locations are used only for records. This is the approach followed by the language Ballerina with a small caveat that we are going to explain in the next subsection.

F.2 Records with erasable fields

The theory of locations we presented so far, works for records values whose domain is fixed: if we have a record value with a mutable location field, then all we can do is to modify the content of

the field according to its static type, but we cannot erase that field, even if at the moment of the creation of that value the field was declared to be optional. Likewise, we cannot add to a record value an absent field even if that field was declared as optional. These two operations would require the system to allow the program to write a location in a field that is $\bot$ and to write $\bot$ in a filed that contains a location. But all a program can do is to write in a location a value that is in the "write-type" associated to the location.

We want to modify the interpretation of record types and record values so that this becomes possible. That is, whenever a record field is declared mutable (read and write) *and optional*, then it means that this field can be not only mutated, but also erased if present and added if absent.

Recall that we write $X_\bot$ for $X \cup \{\bot\}$ with $\bot \notin X$. Without mutable types, record values are interpreted as quasi-constant functions that map labels into either an element of the domain or into $\bot$, that is $(\mathcal{L} \to \mathcal{D}_\bot)$.

Simply adding locations to $\mathcal{D}$, that is defining it as $\mathcal{D} = \mathcal{B} + \mathcal{D}\times\mathcal{D} + \mathcal{P}_{\text{fin}}(\mathcal{D}\times\mathcal{D}_\Omega) + (\mathcal{L} \to \mathcal{D}_\bot) + (\mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{fin}}(\mathcal{D}))$ is not a solution good enough, since we would have a system that would mix values of different types that are in practice indistinguishable one for the other, as it is the case for values of type $\mathbf{loc}(t, 0)$ and those of type t. This is why in the previous subsection we suggested limiting the use of locations as the codomain of quasi-finite functions. This corresponds to interpreting record values as quasi constant functions in $(\mathcal{L} \to (\mathcal{D}\times\mathcal{P}_{\text{fin}}(\mathcal{D})\times\mathcal{P}_{\text{fin}}(\mathcal{D}))_\bot)$. In words, a record value is a map that associates each label either to a location or to $\bot$.

If we want to allow the possibility for records to add and delete mutable optional fields, then the required modification is straightforward, since it suffices to interpret records into locations that contain either a value or $\bot$, that is a function in $(\mathcal{L} \to (\mathcal{D}_\bot\times\mathcal{P}_{\text{fin}}(\mathcal{D}_\bot)\times\mathcal{P}_{\text{fin}}(\mathcal{D}_\bot)))$. The difference is that while in the previous interpretation a label was mapped either to $\bot$ or to a location containing some value, here a label always mapped to a location, that either contains $\bot$ or it contains some value. In this setting a record value in which the key k is undefined corresponds to a quasi-constant function that maps the key k into a location containing $\bot$. In practice this means that the empty open record type $\{\}$ is now interpreted as $\{\!\{\_ = \mathbf{loc}(\mathbb{1}\vee\bot, 1)\}\!\}$, while the closed one $\{|\,|\}$ is $\{\!\{\_ = \mathbf{loc}(\bot, 0)\}\!\}$. The interesting part is the interpretation of records with optional fields: for instance $\{|\,\ell \Rightarrow \mathsf{Int}\,|\}$ is interpreted as $\{\!\{\ell = \mathbf{loc}(\mathsf{Int}\vee\bot, 1), \_ = \mathbf{loc}(\bot, 0)\}\!\}$, and a record in which the location associated to ℓ contains the value $\bot$ means that the field ℓ is absent. It now becomes possible for record values of type $\{|\,\ell \Rightarrow \mathsf{Int}\,|\}$ to add the field ℓ even if it is absent (just, write an integer in its location), and to delete the field ℓ even when it is present (just, write $\bot$ in it). When computing the various set-theoretic operations (in particular complement) on the content of locations, one must take into account that the interpretation domain is $\mathbb{1} \vee \bot$ rather than just $\mathbb{1}$.

From the language point of view there are some modifications, too, because in this setting the addition and removal of fields are now side effects and must be typed accordingly: namely, we must check that whenever we add or remove a field from a record expression, the field is declared as optional in the type of the record expression.

F.3 Matching values

The last modification we are going to describe is the addition of a new type to allow precise typing of pattern matching expressions as they are implemented in Ballerina.

In Ballerina the matching expression looks only at the content of the locations that form a record value, but at not at their mutability. A pattern such as `{ x = int ...}` (written in Ballerina's syntax) matches every record value in which the field `x` is defined and contains an integer value, whatever its mutability status is. In other terms, this pattern matches a record value $\{x = v\}$ for some very different definitions of v, e.g. $v = \texttt{cell}_{\mathbb{0},\mathbb{0}}42$ (the location contains an integer and it is readonly) $v = \texttt{cell}_{\mathsf{Int},\mathsf{Int}}42$ (the location contains an integer, now and after any possible mutation),

and $v = \texttt{cell}_{\mathbb{1},\mathbb{1}}\,42$ (the location contains an integer now but this may not be true after a mutation, since its read-write type permits to assign any value to the location). Therefore, in the case where the field of the value contains a writable mutation, the fact that the match succeeded *does not* imply that the matched value is of *type* `{ x = int ...}`: values of this type are values for which the type system statically ensures that a selection of the field `x` always returns an integer for every possible mutations (e.g., the first two definitions of v above); instead the *pattern* `{ x = int ...}` matches also values in which the field `x` is a location that currently contains an integer but for which the type system *does not* ensure that it will always contain integers, since by some later mutation the field `x` could be updated with a value different from an integer (e.g., the third definition of v).

This is a design choice of Ballerina justified by the fact that when a program is processing untyped input (e.g., from an HTTP request with a body of application/json), the program will receive the input as a value of type `json`. This kind of input is processed by the matching expression. We do not want to make that input to be immutable, since we want to allow the program to transform the JSON code by directly mutating it instead of generating a local copy (particularly, in a gateway scenario). Even if it was immutable, we would not want users to have to declare the type as readonly in order to get the right result: unsophisticated users are not expected to know about readonly types. Processing the input—i.e., using a match expression on that input—needs just to consider the value part of the mutable cells, which explains the semantics of the pattern `{ x = int ...}` we described above. In order to take into account mutability—i.e., the type of the input—a different operator called "`is`" is provided: the programmer has the possibility to match only values whose type "`is`" `{ x = int ...}`.

In order to capture in a type the set of values accepted by record patterns as the above, we extend the types for location with a new "∞" flag:

$$\begin{aligned}\mathbf{loc}(X,0) &= \{(d,\varnothing,\varnothing) \mid d \in X\}\\ \mathbf{loc}(X,1) &= \{(d,X_1,X_2) \mid d \in X, X_1 \cup X_2 \subseteq X\}\\ \mathbf{loc}(X,\infty) &= \{(d,X_1,X_2) \mid d \in X, X_1 \cup X_2 \subseteq \mathcal{D}\}\end{aligned}$$

As before $\mathbb{E}(\mathbf{loc}(t,b)) = \mathbf{loc}(\llbracket t \rrbracket, b)$, so that our interpretation will validate the equation $\llbracket \mathbf{loc}(t,b) \rrbracket = \mathbf{loc}(\llbracket t \rrbracket, b)$. Thus, a value of type $\mathbf{loc}(t,\infty)$ is a location that contains a value of type t whatever its write and read types are.

The modifications to make to the theory in order to include this new type are really minimal. The typing rules must be modified to account for the existence of this new type in particular we have:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{cell}_{t_1,t_2}\, e : \mathbf{loc}(t,\infty)} \qquad \frac{\Gamma \vdash e : t \qquad t_1 \vee t_2 \leq t}{\Gamma \vdash \texttt{cell}_{t_1,t_2}\, e : \mathbf{loc}(t,1)} \qquad \frac{\Gamma \vdash e : t \qquad t_1 \vee t_2 \leq \mathbb{0}}{\Gamma \vdash \texttt{cell}_{t_1,t_2}\, e : \mathbf{loc}(t,0)}$$

while for reads and writes we have

$$\frac{\Gamma \vdash e : \{\ell = \mathbf{loc}(t,b)\} \quad b \leq 1}{\Gamma \vdash e.\ell : t} \qquad \frac{\Gamma \vdash e : \{\ell = \mathbf{loc}(t,\infty)\}}{\Gamma \vdash e.\ell : \mathbb{1}}$$

$$\frac{\Gamma \vdash e_1 : \{\ell = \mathbf{loc}(t,1)\} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1.\ell = e_2) : ()} \qquad \frac{\Gamma \vdash e_1 : \{\ell = \mathbf{loc}(t,\infty)\} \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash (e_1.\ell = e_2) : ()}$$

where the order on the read/write flags is $0 \leq 1 \leq \infty$. The rules for locations with the ∞ flag state that reading such a location will return some value (of generic type $\mathbb{1}$) and that values of any type can be assigned to them.

Lemma F.2 holds as it is, without any modification to its statement.

Regarding Lemma F.3, we have to extend it to take into account the types of the form $\mathbf{loc}(t, \infty)$. The extension is straightforward and it consists of adding two more clauses to cover the case when $\min_{i\in P} b_i = \infty$, and slightly modifying the third clause in the statement, that is.

Lemma F.4. *Let $(X_i)_{i\in P}$ and $(Y_j)_{j\in N}$ be two families of subsets of $\mathcal{D}$ and $(b_i)_{i\in P}$ and $(b'_j)_{j\in N}$ two families of flags with value in $\{0, 1, \infty\}$. Then:*

$$\bigcap_{i\in P} \mathbf{loc}(X_i, b_i) \subseteq \bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$$

$$\Longleftrightarrow$$

$$\left(\bigcap_{i\in P} X_i = \emptyset\right) \text{ or } \left(\min_{i\in P} b_i = 0 \text{ and } \bigcap_{i\in P} X_i \subseteq \bigcup_{j\in N} Y_j\right) \text{ or } \left(\exists j \in N.\ \bigcap_{i\in P} X_i \subseteq Y_j \text{ and } b'_j \geq 1\right) \text{ or}$$

$$\left(\exists j \in N.\ Y_j = \mathcal{D} \text{ and } b'_j = 1\right) \text{ or } \left(\bigcap_{i\in P} X_i \subseteq \bigcup_{\{j\in N \mid b'_j = \infty\}} Y_j\right)$$

Proof. The proof for the cases when $\min_{i\in P} b_i$ is either 0 or 1 are essentially the same as in the proof of Lemma F.3. In this proof there remains a further case to examine, that is when $\min_{i\in P} b_i$ is ∞. The last two clauses in the statement of the lemma cover the case when $\min_{i\in P} b_i = \infty$, that is when $b_i = \infty$ for all $i \in P$. In that case it is easy to see that $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i)$ is the set of all triplets (d, X_1, X_2) such that $d \in \bigcap_{i\in P} X_i$ where X_1 and X_2 are any subsets of $\mathcal{D}$. If we assume that $\bigcap_{i\in P} \mathbf{loc}(X_i, b_i) \subseteq \bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$, then this in particular implies that for all $d \in \bigcap_{i\in P} X_i$ we have that $(d, \mathcal{D}, \mathcal{D})$ must belong to $\bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$. Now, given a set of triplets $\mathbf{loc}(X, b)$ there are only two possible cases for $(d, \mathcal{D}, \mathcal{D})$ to belong to $\mathbf{loc}(X, b)$: either (i) $X = \mathcal{D}$ and $b \geq 1$, since then for both $b = 1$ and $b = \infty$ we have that $(d, \mathcal{D}, \mathcal{D})$ belongs to $\mathbf{loc}(X, b)$ or (ii) we have that $d \in X$ and $b = \infty$. In the first case $(d, \mathcal{D}, \mathcal{D})$ belonging to $\bigcup_{j\in N} \mathbf{loc}(Y_j, b'_j)$ implies that there exists $j \in N$ such that $Y_j = \mathcal{D}$ and $b'_j = 1$ (which is covered by the fourth clause) or $b'_j = \infty$ (which is covered by the last clause). In the second case the same hypothesis yields that for every $d \in \bigcap_{i\in P} X_i$ there exists a $j \in N$ such that $d \in Y_j$ and $b'_j = \infty$, which implies that $\bigcap_{i\in P} X_i \subseteq \bigcup_{\{j\in N \mid b'_j = \infty\}} Y_j$ (covered by the last clause). We have obtained the result. □

In summary the proof of the lemma above tells us that to check the containment given in the statement we need first to compute the intersection $\bigcap_{i\in P} X_i$ and check whether it is empty. If such is not the case, then we compute $\min_{i\in P} b_i$ and according to the result we apply one of the following cases:

(1) if $\min_{i\in P} b_i = 0$, then we check whether $\bigcap_{i\in P} X_i \subseteq \bigcup_{j\in N} Y_j$;
(2) if $\min_{i\in P} b_i = 1$, then we check whether there exists $j \in N$ such that $b'_j \geq 1$ and $\bigcap_{i\in P} X_i \subseteq Y_j$
(3) if $\min_{i\in P} b_i = \infty$, then we check whether there exists a $j \in N$ such that $b'_j = 1$ and $Y_j = \mathcal{D}$ and, if not, we check whether $\bigcap_{i\in P} X_i$ is contained in the union of all Y_j such that $b'_j = \infty$.

Finally, it is straightforward to transpose this to deciding subtyping for locations: it suffices to replace the sets X's and Y's with (the interpretation of) the types and substitute for each use of $\mathcal{D}$ either $\mathbb{1}$ or, if we are in the case of erasable field discussed in Section F.2, $\mathbb{1} \vee \bot$.

Thanks to this extension, we can reuse the technique to type pattern matching that we described in Section 3.3 with minimal changes. Accounting for the Ballerina's semantics of record patterns does not require any significant modification to the definitions given in Section 3.3: the only definition that needs to be modified is the one of accepted type of a pattern, so that it reflects Ballerina's semantics of pattern matching. In particular, we want the accepted type of the pattern `{ x = int ...}` to be the type $\mathbf{\{}x = \mathbf{loc}(\mathsf{Int}, \infty)\mathbf{\}}$. More generally, we will have $\wr\{\ell = p\}\int = \mathbf{\{}\ell = \mathbf{loc}(\wr p\int, \infty)\mathbf{\}}$.

Received 2023-03-01; accepted 2023-06-27