Guard Analysis and Safe Erasure Gradual Typing: a Type System for Elixir

GIUSEPPE CASTAGNA, CNRS, Université Paris Cité, France
GUILLAUME DUBOC, Université Paris Cité and Remote Technology, France

We define several techniques to extend gradual typing with semantic subtyping, specifically targeting dynamic languages. Focusing on the Elixir programming language, we provide the theoretical foundations for its type system. Our approach demonstrates how to achieve type soundness for gradual typing in existing dynamic languages without modifying their compilation, while still maintaining high precision. This is accomplished through the static detection of *strong functions*, which leverage runtime checks inserted by the programmer or performed by the virtual machine, and through a fine-grained type analysis of pattern-matching expressions with guards.

1 Introduction

Elixir is an open-source dynamic functional programming language that runs on the Beam, the Erlang Virtual Machine [36]. It was designed for building scalable and maintainable applications with a high degree of concurrency and seamless distribution. Its characteristics have earned it a surging adoption by hundreds of industrial actors and tens of thousands of developers. Despite being a dynamically-typed language, there exist tools to perform static analysis on Elixir programs, such as Dialyzer [31] or Gradualizer [37], and attempts have been made at forming a theoretic basis to type it, but with clear limitations [5]. To answer the demand for a stricter, more expressive and informative type system for Elixir, Castagna et al. [9] describe how set-theoretic types augmented with the dynamic type of gradual typing can be used to introduce static typing into Elixir. The approach assumes that Elixir programmers would start annotating their code with types and gradually add more and more of them, until they reach a fully statically-typed program. In this work, we describe the technical difficulties in fitting the type system outlined in [9] onto a language like Elixir, with all its quirks, and our solutions to those problems. To do so, we define Core Elixir, a subset of Elixir on which we define a gradual type system with semantic subtyping before proving the usual type safety results on the type system.

The main novel idea presented in this work may be the one of *strong functions*, which is described in Section 2. In the theory and application of gradual typing, there is a clear rift between two kinds of gradually typed languages. Firstly, those that treat the dynamic type as a liability that needs to be *checked*; to preserve their soundness, these languages (e.g., Reticulated Python [39]) use different strategies to insert type-checks into their runtime to protect statically typed parts of their code from dynamically typed ones. But for some languages, such as Typescript [4], this is not an option as the runtime does not check types by default. Hence, a second way for gradual systems is to resort to *full erasure*: the types of a TypeScript program leave no trace in the JavaScript emitted by the compiler, thus, every type property comes from static analysis. This can lead type systems towards *unsoundness*, and thus the program can crash due to unsafe uses of dynamic.

However, our situation is more nuanced. Indeed, although Elixir is dynamically typed, it is compiled and then executed on the Erlang VM which, itself, is *type-safe* through explicit type-checks. Furthermore, such checks can be executed at the guise of the programmer, who writes those into guards. Hence, we had the opportunity to quantify how much checking the VM actually does, and integrate that into our plans for a gradual type system. The concept of strong functions directly

Authors' Contact Information: Giuseppe Castagna, Institut de Recherche en Informatique Fondamentale (IRIF), CNRS, Université Paris Cité, France; Guillaume Duboc, Institut de Recherche en Informatique Fondamentale (IRIF), Université Paris Cité and and Remote Technology, France.

comes from that: these are functions whose input and output types are entirely or partially *checked* by the VM either because of checks inserted by the programmer or by standard checks performed by the VM; hence, it is possible even when applied in uncertain conditions (when dynamic code is involved) to give their result a static type. This approach, that we call *safe-erasure gradual typing*, refers to the fact that, although no checks are inserted into the language related to the types asserted by the typechecker, this "type erasure" is safe, because the type-checker knows which parts of the code are checked by the VM or the programmer, and which are not. Practically, this means that the typechecker can be more precise in its analysis, and infer a static type where a dynamic type was used, because it knows that the VM will check the type of the value at runtime. A requirement for this approach to work is to be able to extract as much (necessarily, static) type information from Elixir guards as possible, which is the subject of the technical analysis developed in Section 3.

A key of our approach is the use of semantic subtyping, that allows us the use of set-theoretic type operators (union, intersection, difference), but also provides us with a decidable subtyping relation [17, 26], which appears crucial especially in the analysis of guards. Indeed, this analysis constantly mixes very precise conditions on types (including singleton types), and uses intersections, differences, and unions to refine these results. It would not otherwise be possible to guarantee such a level of precision, and the pattern matching would end up being grossly approximated.

1.1 A Walkthrough of the Work

The type system of Elixir described by Castagna et al. [9] is a gradual polymorphic type system based on the polymorphic type system of CDuce [14, 15]. In this work, we describe the technical additions that are missing in the CDuce type system in order to type Elixir programs, presenting them one by one. These are the techniques of *strong function typing* and *propagation of the* `dynamic()` *type* necessary for safe-erasure gradual typing, both described in Section 2; the *guard analysis* described in Section 3; the typing (and subtyping) of *multi-arity functions* presented in Section 4; the *type inference for anonymous functions* described in Section 5. In this section, we are going to present them one after the other by giving some small examples that should help the reader understand the technical developments described in the following sections.

1.1.1 Soundness. The type system we present here satisfies the following soundness property

> If an expression is of type t, then it either diverges, or produces a value of type t, or fails on a dynamic check either of the virtual machine or inserted by the programmer

whose formal statement is given in Theorem 2.3. The system is gradual since the type syntax includes a `dynamic()` type used to type expressions whose type is unknown at compile time. The soundness guarantee above is typical of the so-called *sound gradual typing* approaches. These approaches ensure soundness by using typing derivations to insert some suitable dynamic checks at compile time. Our system, instead, does not modify Elixir standard compilation: types are not used for compilation and are erased after type-checking. Our system is, thus, a *type-sound (i.e., safe) erasure gradual typing system*, the first we are aware of. In particular, the compiler does not insert any dynamic check in the code apart from those explicitly written by the programmer. Therefore, our system must ensure soundness by considering only the checks written by the programmer or performed by the Beam machine.

Writing a sound gradual type system for Elixir is easy: since every Elixir computation that does not diverge or error returns a value (no stuck terms, thanks to the Beam), then a system that types every expression by `dynamic()` is trivially sound . . . but hardly useful. Therefore, we need a system that must fulfill two opposite requirements

(1) it must use `dynamic()` as little as possible so as to be useful, and

(2) it must use `dynamic()` enough so as not to hinder the versatility of gradual typing

The first requirement is fulfilled by the typing of strong functions, the second requirement by the propagation of `dynamic()`. We will demonstrate both of these aspects next.

1.1.2 Strong functions (Section 2). Consider the definition in Elixir of a function `second` that selects the second projection of its argument (`elem(`e, n`)` selects the (n+1)-th projection of the tuple e):

```
def second(x), do: elem(x,1)
```

If the argument of the function is not a tuple with at least two elements, then the Beam raises a runtime exception. The function definition above is untyped. We can declare its type by preceding it by a `$`-prefixed type declaration, as in

```
$ dynamic() -> dynamic()
def second(x), do: elem(x,1)
```

This is one of the simplest types we can declare for `second`, since it essentially states that `second` is a function, and nothing more: it expects an argument of an unknown type and returns a result of an unknown type. We can give `second` a type slightly more precise than (i.e., a subtype of) the type above, that is:

```
$ {dynamic(), dynamic(), ..} -> dynamic()
```

which states that the argument of a function of this type must be a tuple with *at least* two elements of unknown type (the trailing "`..`" indicates that the tuple may have further elements). With this declaration, the application of `second` to an argument not of this type will be statically rejected, thus statically avoiding the runtime raise by the Beam. We can give to the function also a non-gradual type—i.e., a type in which `dynamic()` does not occur: we call them *static types*—, as:

```
$ {term(), integer()} -> integer()
```

This type declaration states that `second` is a function that takes a pair whose second element is an integer (`term()` is Elixir's top type that types all values) and returns an integer. If this is the type declared for `second`, then the type deduced for the application `second({true,42})` is, as expected, `integer()`. If `dyn` is an expression of type `dynamic()`, then the type deduced for `second(dyn)` will be `dynamic()`: if `dyn` evaluates into a tuple with at least two elements, then the application will return a value that can be of any type, thus we cannot deduce for it a type more precise than `dynamic()`. This differs from current sound gradual typing approaches, which would deduce `integer()` for this application, but also insert a runtime check that verifies that the result is indeed an integer. However, this is not the way an Elixir programmer would have written this function. If the programmer intention is that `second` had type `{term(), integer()} -> integer()`, then the programmer would rather write it as follows:

```
$ {term(), integer()} -> integer()
def second_strong(x) when is_integer(elem(x,1)), do: elem(x,1)
```

This is defensive programming. The programmer inserts a *guard* (introduced by the keyword `when`) that checks that the argument is a tuple whose second element is an integer (the analysis of this kind of complex guards is the subject of Section 3). Thanks to this check (which makes up for the one inserted at compile time by other sound gradual typing approaches) we can safely deduce that `second_strong(dyn)` has type `integer()`. The above is called a *strong function*, because the programmer inserted a dynamic check that ensures that even if the function is applied to an argument not in its domain, it will always return a result in its codomain—i.e., `integer()`—or fail.

This allows the system to deduce for `second_strong(dyn)` the type `integer()` instead of `dynamic()`, thus fulfilling our first requirement. A function can be strong not only because it was defensively programmed, but also thanks to the checks performed at runtime by the Beam, as for:

```
$  {term(), integer()} -> integer()
def inc_second(x), do: elem(x,1) + 1
```

which is also strong because the Beam dynamically checks that both arguments of an addition are of type `integer()`. Therefore, also in this case, we know that if the function returns a value, then this is an integer. Thus, we can safely deduce the type `integer()` for `inc_second(dyn)` and, thus, for instance, that the addition `inc_second(dyn) + second_strong(dyn)` is well typed. To determine whether a function is strong, we define in Section 2 an auxiliary type system that checks whether the function, when applied to arguments *not* in its domain, returns results in its codomain or fails.

1.1.3 Propagation of `dynamic()` *(Section 2).* In fact, for both the above applications, `inc_second(dyn)` and `second_strong(dyn)`, our system deduces a type better than (i.e., a subtype of) `integer()`: it deduces `integer() and dynamic()`. This is an *intersection type*, meaning that its expressions have both type `integer()` and type `dynamic()`. What the system does is to propagate the type `dynamic()` of the argument `dyn` of the applications into the result. This is meant to preserve the versatility of the gradual typing that originated the application, thus fulfilling our second requirement: expressions of this type can be used wherever an integer is expected, but also wherever any strict subtype of `integer()` (e.g., natural numbers) is. To see the advantages of this propagation, consider the following example that we also use to introduce more set-theoretic type connectives:

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

The definition of `negate` is given by multiple clauses tested in the order in which they appear. When `negate` is applied, the runtime first checks whether the argument is an integer, and if so, it executes the body of the first clause, returning the opposite of `x`; otherwise, it checks whether it is a Boolean, and if so returns its negation; in any other case the application fails. Multi-clause definitions, thus, are equivalent to (type-)case expressions (and indeed, in Elixir they are compiled as such). The same static checks of *redundancy* and *exhaustiveness* that are standard for case expressions apply here, too. For instance, if we declare `negate` to be of type `integer() -> integer()`, then the type system warns that the second clause of `negate` is redundant; if we declare the type `term() -> term()` instead, then the function is not well-typed since the clauses are not exhaustive. To type the function above without any warning, we can use a union type, denoted by `or`:

```
$ integer() or boolean -> integer() or boolean()
```

which states that `negate` can be applied to either an integer or a Boolean argument and returns either an integer or a Boolean result. Next, let us consider the following definition

```
$ dynamic(), dynamic() -> integer()
def subtract(a, b), do: a + negate(b)
```

and see whether it type-checks. The type declaration states that `subtract` is a function that, when applied to two arguments of unknown type, returns an integer (or it diverges, or fails). Since the parameter `b` is declared of type `dynamic()`, then the system deduces that `negate(b)` is of type `(integer() or boolean) and dynamic()` (the `dynamic()` in the type of `b` is propagated into the type of the result). To fulfill local requirements, the static type system can assume `dynamic()` to become any type at run-time: following the terminology by Castagna et al. [11], we say that `dynamic()` can *materialize* into any other type. In the case at issue, addition expects integer arguments. Therefore, the function body is well typed only if we can deduce `integer()` for `negate(b)`. This is possible since the type of this expression is `(integer() or boolean) and dynamic()` and the system can materialize the `dynamic()` in there to `integer()` thus deriving (a type equivalent to) `integer()`.

Notice the key role played in this deduction by the propagation of `dynamic()`: had the system deduced for `negate(b)` just the type `integer() or boolean()`, then the body would have been rejected by the type system since additions expect `integer()`, and not `integer() or boolean()`.

A similar problem would happen had we declared `subtract` to be of type

```
15 $ integer(), integer() -> integer()
```

In that case, the type `integer() or boolean() -> integer() or boolean()` is not good enough for `negate`: since we assume `b` be to be of type `integer()`, then the type deduced for `negate(b)` is again `(integer() or boolean())` which is not accepted for additions. The solution is to give `negate` a better type by using the intersection type

```
16 $ (integer() -> integer()) and (boolean() -> boolean())
```

which is a subtype of the previous type in line 12, and states that `negation` is a function that returns an integer when applied to an integer and a Boolean when applied to a Boolean. This type allows the type system to deduce the type `integer()` for `negate(b)` whenever `b` is an integer. This example shows why it is important to specify (or infer) precise intersection types for functions. The inference system we present in Section 5 will infer for an untyped definition of `negate` the intersection of arrows in line 16 rather than the less precise arrow with unions of line 12.

Finally, we want to signal that the new typing of `negate` in line 16 does not modify the propagation of `dynamic()`: the type deduced for `negate(dyn)` is still `(integer() or boolean) and dynamic()`.

1.1.4 Guard Analysis (Section 3). Until now, the guards employed in our examples primarily involve straightforward type checks on function parameters (e.g., `is_integer(a)`, `is_boolean(x)`). The system we investigate for safe-erasure gradual typing in Section 2 exclusively focuses on these kinds of tests. There is a single exception in our examples with a more intricate guard, specifically `is_integer(elem(x,1))` used in line 7. In Elixir, guards can encompass complex conditions, utilizing equality and order relations, selection operations, and Boolean operators. To illustrate, consider the following (albeit artificial) definition:

```
17 def test(x) when is_integer(elem(x,1)) or elem(x,0) == :int, do: elem(x,1)
18 def test(x) when is_boolean(elem(x,0)) or elem(x,0) == elem(x,1), do: elem(x,0)
```

The first clause of the `test` definition executes when the argument is a tuple where either the second element is an integer *or* the first element is the atom `:int` (in Elixir, atoms are user-defined constants prefixed by a colon). The second clause requires its argument to be a tuple in which the first element is either equal to the second element or is a Boolean.

To type this kind of definitions, the system needs to conduct an analysis characterizing the set of values for which a guard succeeds. Section 3 presents an analysis that characterizes this set in terms of types. In some cases, it is possible to precisely represent this set with just one type. For example, the set of values satisfying the guard `is_integer(elem(x,1))` in line 7 corresponds exactly to the values of type `{term(), integer(), ..}`. Likewise, the arguments that satisfy the guard of the first clause of `test` in line 17 are precisely those of the union type `{term(), integer(), ..} or {:int, term(), ..}`, where `:int` denotes the singleton type for the value `:int`. However, such a precision is not always achievable, as demonstrated by the guard in the second clause of `test` (line 18). Since it is impossible to characterize by a type all and only the tuples where the first two elements are equal, we have to approximate this set. To represent the set of values that satisfy such guards, we use two types—an underapproximation and an overapproximation—referred to as the *surely accepted type* (since it contains only values for which the guard succeeds) and the *possibly accepted type* (since it contains all the values that have a chance to satisfy the guard).[1] For the guard in line 18, the surely accepted type is `{boolean(), ..}` since all tuples whose first element is a Boolean satisfy the guard; the possibly accepted type, instead, is `{term(), term(), ..} or {boolean()}` since the only values that may satisfy the guard are those with at least two elements or those with just one element of type `boolean()`. When the possibly accepted type and the surely accepted type coincide, they provide a precise characterization of the guard, as demonstrated in the two previous examples of guards (lines 7 and 17).

The type system uses these types to type case-expressions and multi-clause function definitions. In particular, to type a clause, the system computes all the values that are *possibly* accepted by its guard, minus all those that are *surely* accepted by a previous clause, and use this set of values to type the clause's body. For example, when declaring `test` to be of type `{term(), term(), ..} -> term()`, the system deduces that the argument of the first clause has type `{term(), integer(), ..} or {:int, term(), ..}`. For the second clause, the system subtracts the type above from the possibly accepted type of the second clause's guard (intersected with the input type, i.e., `{term(), term(), ..}`), yielding for `x` the type `{not(:int), not(integer()), ..}`, that is, all the tuples with at least two elements where the first is not `:int` (`not` *t* denotes a *negation type*, which types all the values that are not of type *t*) and the second is not an integer.

If the difference computed for some clause is empty, then the clause is redundant and a warning is issued. This happens, for instance, for the second clause of `test`, if we declare for the function `test` the type `{:int, term(), ..} -> term()`: all arguments will be captured by the first clause.

If the domain of the function (or, for case expressions, the type of the matched expression) is contained in the union of the *surely* accepted types of all the clauses, then the definition is exhaustive. For instance, this is the case if we declare for `test` the type `{term(), boolean()} -> term()`. If, instead, it is contained only in the union of the *possibly* accepted types, then the definition *may* not be exhaustive, and a warning is emitted as for declaring `{term(), term(), ..} or {boolean()} -> term()`. In all the other cases, the definition is considered ill-typed, as for a declared type `tuple() -> term()` (where `tuple()` is the type of all tuples), since a tuple with a single element that is not a Boolean is an argument in the domain that cannot be handled by any clause.

Finally, the guard analysis of Section 3 produces for each guard a result that is more refined than just the possibly and surely accepted types for the guards. For each guard, the analysis partitions these types into smaller types that will then be used by the inference of Section 5 to produce a typing for non-annotated functions. For instance, for the untyped version of `test` given in lines 17–18 the analysis will produce four different input types that the inference of Section 5 will use to deduce the following intersection type for `test`:

```
$ ({term(), integer(), ..} -> integer()) and
  ({:int, term(), ..} -> term()) and
  ({boolean()} or {boolean(), not(integer()), ..} -> boolean()) and
  ({not(boolean() or :int), not(integer()), ..} -> not(boolean() or :int)
```

Splitting the domain of `test` as in the code above is not so difficult since its guards use the connective `or` and, as we will see, to compute the split, the system in Section 3 normalizes guards into Boolean disjunctions. Notice, however, that the analysis must take into account the order in which the guards

[1] Formally, the *surely accepted type* is the largest type contained in all types containing only values that satisfy the guard, and the *possibly accepted type* is the smallest type containing all types that contain only values that satisfy the guard.

are written. If in line 18 we use the guard `elem(x,0) == elem(x,1) or is_boolean(elem(x,0))`, that is, we swap the order of the operands of the guard, then the arrow type in line 21 is no longer correct, since the application `test({true})` would fail and, therefore, the type `{boolean()}` must not be included in the domain of the arrow in line 21.

1.1.5 Multi-arity Functions (Section 4). At lines 13-14 we defined the function `subtract` which has two parameters. This arity is reflected in its type, where its domain consists of two comma-separated types. All the other functions given as example are unary. While the distinction between unary and binary functions may seem trivial to a programmer, it holds significant implications for the type system. The CDuce type system can only handle unary functions, and simulates n-ary functions as unary functions on n-tuples. But this is not sufficient in Elixir. First, applying a function to two arguments or to a pair involves different syntaxes, e.g., `subtract(42,42)` and `test({42,42})`. Second, a programmer can explicitly test whether a function f has arity n using `is_function(`f`,`n`)`. Consequently, we need a type system in which it is possible to express the type of all functions of a given arity. For instance, we may want to give a type to:

```
23 def curry(f) when is_function(f, 2), do: fn a -> fn b -> f.(a, b) end end
24 def curry(f) when is_function(f, 3), do: fn a -> fn b -> fn c -> f.(a, b, c) end end end
```

but in current systems with semantic subtyping, we can only express the type of *all* functions, that is, `none() -> term()`.[2] Simulating, say, binary functions with functions on pairs does not work since `{none(), none()} -> term()` is not the type of all binary functions: since the product with the empty set gives the empty set, this type is equivalent to `none() -> term()`, the type of *all* functions. This is the reason why we introduced the syntax $\texttt{(t}_1\texttt{,}\cdots\texttt{,t}_n\texttt{) -> t}$ that outlines the arity of the functions. Now the type of all binary functions can be written as `(none(), none()) -> term()`, and we can declare for the function `curry` the following type (though, type variables or even a gradual type would be more useful than this type).

```
25 $ (((none(), none()) -> term()) -> none() -> none() -> term()) and
26   (((none(), none(), none()) -> term()) -> none() -> none() -> none() -> term())
```

All this requires modifications, both in the interpretation of types and in the algorithm that decides subtyping, that we describe in Section 4.

1.1.6 Inference (Section 5). In a couple of examples we highlighted our system's ability to deduce the function type even in the absence of explicit type declarations. For instance, we said that our type system can infer for `negate` (lines 10–11) the intersection type in line 16, and for `test` (lines 17–18) the type in lines 19–22. This kind of inference is different from that performed for parametric polymorphism by languages of the ML family. Instead, it leverages the guard analysis of Section 3 to derive the type of guarded functions: it simply considers the guards of the different clauses of a function definition as implicit type declarations for the function parameters, and use them for type inference.

This kind of inference is used when explicit type declarations are omitted. This is particularly valuable for anonymous functions of which we saw a couple examples in the definition of `curry` (lines 23–24) where the body of the two clauses consists of anonymous functions. We aim to avoid imposing an obligation on programmers to explicitly annotate anonymous functions as in:

[2] A value is of type s `->` t iff it is a function that when applied to an argument of type s, it returns only results of type t; thus, every function vacuously satisfies the constraint `none() -> term()`, as there is no value of type `none()`.

```
27 $ list(integer()) -> list(integer())
28 def bump(lst), do: List.map(fn x when is_integer(x) -> x + 1 end, lst)
```

Here, the guard already provides the necessary information, making explicit annotations superfluous. Additionally, we view the use of an untyped or anonymous function as an implicit application of gradual typing. We have seen in §1.1.3, that whenever gradual typing was explicitly introduced by an annotation, the system propagated `dynamic()` in all intermediate results so as to preserve the versatility of the initial gradual typing. We do the same here and propagate the (implicit use of) `dynamic()` in the results of the anonymous/untyped functions by intersecting their inferred type with an extra arrow of the form t `-> dynamic()`, where t is the domain inferred for the function. For example, the type inferred for `negate` will be the type in line 16 intersected with the type `integer() or boolean() -> dynamic()`, while the intersection in lines 19–22 inferred for `test` will have an extra arrow `{term(), term(), ..} or {boolean()} -> dynamic()`. Likewise, the type `(integer() -> integer()) and (integer() -> dynamic())` will be given to the anonymous function in the body of `bump` (line 28); this type is equivalent to the simpler type `integer() -> integer() and dynamic()`. All these concepts are formalized in Section 5.

1.1.7 Implementation in Elixir (Section 6). All the features and algorithms presented here are included in Elixir, since the 1.17 release of the language [21]: the front-end of the Elixir's compiler types (multi-arity) functions using safe erasure gradual typing, with strong functions, `dynamic()` propagation, and guard analysis. The latter is used to perform inference as described in §1.1.6. The current 1.18 (beta) implementation covers basic, atom, tuple, and map types, and emits warnings when it fails to type. Although the missing types and the type annotations for functions are planned only for future releases (see [20]), the data-structures and algorithms for the types described in this work are already part of the official compiler. Therefore, we wrap up by presenting in Section 6 the type-checker performance on large codebases and the user feedback we received so far.

1.2 Contributions and Limitations

Our primary contribution is the establishment of the theoretical foundations of the Elixir type system, whose general lines were outlined by [8]. Notably, we define what we believe to be the first safe-erasure gradual type system. The technical contributions can be succinctly outlined as follows:

(1) **Gradual Typing:** Formalization of techniques for strong functions and `dynamic()` propagation.
(2) **New Typing Technique:** Definition of a typing technique for guards and patterns based on the concepts of possibly accepted types and surely accepted types.
(3) **Semantic Subtyping Extension:** Extension of semantic subtyping to incorporate multi-arity function spaces.
(4) **Type Inference Techniques:** Development of type inference techniques for anonymous functions, leveraging pattern and guard analysis.
(5) **Properties:** Definition of three different characterizations of type safety and the proofs of these properties for a language equipped with safe-erasure gradual typing.

The characterization of the gradual safety in the last point was the big technical challenge of this work, since it needs to be stated in terms of a new relation $v \mathrel{\overset{\circ}{\circ}} t$, laxer than the typing relation $v : t$.

Concerning limitations, some are intentional omissions, like typing of records and maps (defined by Castagna [7]) and parametric polymorphism (defined by Castagna et al. [14, 15], orthogonal to features introduced in this work). Others are genuine limitations, with the two most prominent being constrained application of type narrowing and absence of type reconstruction à la ML.

Regarding type narrowing, our system incorporates a simplistic form of it, allowing specialization in the branches of a case-expression of the type of variables occurring in the matched expression under specific conditions (see Section 3). However, it does not achieve the level of granularity seen in the analyses by Tobin-Hochstadt and Felleisen [38] and Castagna et al. [13]. Concerning type reconstruction, although recognized as valuable, in particular for anonymous functions, it was not explored in this work, with the example of the `curry` function highlighting its potential significance. While a theoretical solution for addressing both limitations exists, as defined by Castagna et al. [12], its current computational cost remains too high for practical integration into Elixir.

2 Safe Erasure Gradual Typing

We define Core Elixir in Figure 1, a typed λ-calculus with constants c (including tuples of constants $\{0, 1\}$, etc.); variables x; λ-abstractions $\lambda^{\mathbb{I}}x.e$ annotated by interfaces (ranged over by $\mathbb{I}$), that is, finite sets of arrows whose intersection is the type of the λ-abstraction; tuples $\{\overline{e}\}$ (we use the overbar to denote sequences, e.g., $\overline{e}$ stands for $e_1, ..., e_n$) and projections $\pi_e\, e$; type-case expressions case $e\ (\tau_i \to e_i)_{i\in I}$; and, for illustrating the typing of Beam-checked operators, the sum $+$.

The language has strict weak-reduction semantics defined by the reduction rules in Figure 2. The semantics is defined in terms of values v and evaluation contexts $\mathcal{E}$:

Values $v ::= c \mid \lambda^{\mathbb{I}}x.e \mid \{\overline{v}\}$
Contexts $\mathcal{E} ::= \square \mid \mathcal{E}(e) \mid v(\mathcal{E}) \mid \{\overline{v}, \mathcal{E}, \overline{e}\} \mid \pi_{\mathcal{E}}\, e \mid \pi_v\, \mathcal{E} \mid \texttt{case}\ \mathcal{E}\ (\tau_i \to e_i)_{i\in I} \mid \mathcal{E} + e \mid v + \mathcal{E}$

The reduction rules are standard: call-by-value beta-reduction where $e[v/x]$ denotes the capture-free substitution of x with v in e, tuple projection, and a first-match type-case that reduces to the first branch that matches the value. Given a value v and a test type τ, we denote by $v \in \tau$ the fact that v belongs to the set represented by τ (e.g., $0 \in \texttt{int}$ and $\{0, 1\} \in \texttt{tuple}$), and we write $v \notin \tau$ if not (e.g., $0 \notin \texttt{bool}$). The failure reductions correspond to explicit runtime errors raised by the Erlang VM, and they will be used to make the type safety results more precise, by explicitly identifying which failure states are prevented in a typed program. Failures are denoted as a labeled symbol ω_p, where the label p informs of the type of exception (e.g., $\omega_{\textsc{ArithError}}$ for trying to sum non-integer values).

The types are defined in Figure 1. Base types include integers, Booleans, atoms, functions, tuples, and the dynamic type '?'. Also, we have open tuple types: $\{\overline{t}, ..\}$ denotes any tuple starting with a sequence of elements of types $\overline{t}$. The types are set-theoretic, with connectives union $\vee$ and negation $\neg$, with intersection defined as $t_1 \wedge t_2 = \neg(\neg t_1 \vee \neg t_2)$, and difference defined as $t_1 \setminus t_2 = t_1 \wedge \neg t_2$. The top type $\mathbb{1}$, the type of all values, is defined as $\mathbb{1} = \texttt{int} \vee \texttt{atom} \vee \texttt{function} \vee \texttt{tuple}$, while the bottom type $\mathbb{0}$ is defined as $\mathbb{0} = \neg\mathbb{1}$. Note that, since constants are included in types, every value that does not contain λ-abstractions exists as a singleton type. Types are defined coinductively (for type recursion) and, as customary in semantic subtyping, they are contractive (no infinite unions or negations) and regular (necessary for a decidable subtyping relation): see, e.g., [26] for details.

Figure 3 presents the complete non-gradual typing rules for Core Elixir. This system uses a presentation in which only the relevant part of the type environments is presented (i.e., the part $\Gamma \vdash$ is omitted). Furthermore, the notation $\dashv x : t$ means that the (implicit) context ascribes x to t and $x : t' \vdash e : t$ denotes a local context extension that proves $e : t$. Rules marked by a "ω" correspond to cases in which the type-checker emits a warning since it cannot ensure type safety. More precisely, whenever rule (proj_{ω}) or ($\text{proj}^{\mathbb{1}}_{\omega}$) are used, the type-checker warns that the expression may generate an "index out of range" exception. The rules are standard for such a simple calculus, with small changes due to the use of set-theoretic operators: the rule (λ) for lambda abstractions checks intersections of function types by checking that a function is well-typed for every arrow type given in its annotation. Rule (proj) uses the fact that the index can have a union of integer types (since

$$\begin{array}{llll}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda^{\mathbb{I}}x.e \mid e(e) \mid \{\overline{e}\} \mid \pi_e\, e \mid \texttt{case}\ e\ \overline{\tau \to e} \mid e + e \\
\textbf{Test types} & \tau & ::= & b \mid \{\overline{\tau}\} \mid \{\overline{\tau}, ..\} \\
\textbf{Base types} & b & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{atom} \mid \texttt{function} \mid \texttt{tuple} \\
\textbf{Types} & t & ::= & b \mid c \mid t \to t \mid \{\overline{t}\} \mid \{\overline{t}, ..\} \mid t \vee t \mid \neg\, t \mid ? \\
\textbf{Interfaces} & \mathbb{I} & ::= & \{t_i \to t'_i\}_{i=1..n}
\end{array}$$

Fig. 1. Expressions and Types Syntax

$$\begin{array}{llcll}
[\textsc{App}] & (\lambda^{\mathbb{I}}x.e)(v) & \hookrightarrow & e[v/x] & \\
[\textsc{Proj}] & \pi_i\,\{v_0, .., v_n\} & \hookrightarrow & v_i & \text{if } i \in [0\,..\,n] \\
[\textsc{Match}] & \texttt{case}\ v\ (\tau_i \to e_i)_{i\in I} & \hookrightarrow & e_j & \text{if } v \in \tau_j \text{ and } v \notin \bigvee_{i<j} \tau_i \\
[\textsc{Plus}] & v + v' & \hookrightarrow & v'' & \text{where } v'' = v + v' \text{ and } v, v' \text{ are integers} \\
[\textsc{Context}] & \mathcal{E}[e] & \hookrightarrow & \mathcal{E}[e'] & \text{if } e \hookrightarrow e' \text{ without using } [\textsc{Context}] \\
 & & & & \\
[\textsc{App}_\omega] & v(v') & \hookrightarrow & \omega_{\textsc{BadFunction}} & \text{if } v \neq \lambda^{\mathbb{I}}x.e \\
[\textsc{Proj}_{\omega,\textsc{range}}] & \pi_v\,\{v_0, .., v_n\} & \hookrightarrow & \omega_{\textsc{OutOfRange}} & \text{if } v \neq i \text{ for } i = 0\,..\,n \\
[\textsc{Proj}_{\omega,\textsc{notTuple}}] & \pi_{v'}\, v & \hookrightarrow & \omega_{\textsc{NotTuple}} & \text{if } v \neq \{\overline{v}\} \\
[\textsc{Match}_\omega] & \texttt{case}\ v\ (\tau_i \to e_i)_{i\in I} & \hookrightarrow & \omega_{\textsc{CaseEscape}} & \text{if } v \notin \bigvee_{i\in I} \tau_i \\
[\textsc{Plus}_\omega] & v + v' & \hookrightarrow & \omega_{\textsc{ArithError}} & \text{if } v \text{ or } v' \text{ not integers} \\
[\textsc{Context}_\omega] & \mathcal{E}[e] & \hookrightarrow & \omega_p & \text{if } e \hookrightarrow \omega_p \text{ without using } [\textsc{Context}_\omega]
\end{array}$$

Fig. 2. Standard and Failure Reductions

we have union types and integer singleton types), thus it types the projection with the union of all the fields that can be selected. Rule (case) types only the branches that are attainable; for a branch $\tau_i \to e_i$ being attainable means that the type of values produced by e (i.e., in type t), intersected with the test type τ_i, minus all the types of values captured by previous branches (i.e., all values in τ_j for $j < i$) is non-empty. The side condition $t \leq \bigvee_{i\in I} \tau_i$ checks for exhaustiveness.

Remark 1. *The reader may wonder why the presence of a (statically detected) non-attainable branch does not yield a type error. The reason is that this attainability property cannot be decided locally. For instance, to deduce the intersection type* $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$ *for the function* $\lambda^{\{\texttt{int}\to\texttt{int},\, \texttt{bool}\to\texttt{bool}\}}x.\texttt{case}\ x\ (\texttt{int} \to x{+}1, \texttt{bool} \to -x)$*, the system types the case-expression twice: once under the assumption* $x{:}\texttt{int}$*, making the* `bool` *branch unattainable, and once under the assumption* $x{:}\texttt{bool}$ *making the* `int` *branch unattainable. Thus, each branch is attainable at some point, though not at the same time. The property of being statically attainable is, thus, a global property, not expressible in a compositional system. The type-checker will check that every branch of every case is typed at least once, and emit a "unused branch" warning when this condition is not met. We will assume this property, so as to refine the type system used to prove the soundness of our approach (cf. Appendix B).*

The type system of Figure 3 is sufficient to type *non-gradual* Core Elixir programs (i.e., those with interfaces without '?'). After introducing the dynamic type '?', we handle it by a technique we dubbed *safe erasure gradual typing*, which implies the use of additional rules (Figure 4) and an auxiliary system to infer *strong function types* (Figure 5), which are key aspects of our approach.

Our type relations are subtyping ($\leq$), precision ($\preccurlyeq$), and consistent subtyping ($\lessapprox$). The theory of semantic subtyping for gradual types makes it possible to define all these relations just in terms of the semantic subtyping relation on static types, as it is defined by Frisch et al. [26]. Indeed (see Theorem 6.10 in [29]) every gradual type τ is equivalent to (i.e., it is both a subtype and a supertype of) $(? \wedge \tau^{\Uparrow}) \vee \tau^{\Downarrow}$ where $\tau^{\Uparrow}$ (resp. $\tau^{\Downarrow}$) is the *static* type obtained from τ by replacing the covariant occurrences of ? in it with $\mathbb{1}$ (resp. $\mathbb{0}$), and the contravariant occurrences of ? in it with $\mathbb{0}$ (resp. $\mathbb{1}$). For instance, $\{?, ?\}$ (the type of 2-tuples whose two elements can be anything at runtime) is

$$\text{(cst)}\ \frac{}{c : c} \qquad \text{(var)}\ \frac{\dashv x : t}{x : t} \qquad \text{(tuple)}\ \frac{\overline{e : t}}{\{\bar{e}\} : \{\bar{t}\}} \qquad (\lambda)\ \frac{\forall (t_i \to s_i) \in \mathbb{I}\ \ (x : t_i \vdash e : s_i)}{\lambda^{\mathbb{I}}(x).e : \bigwedge_i (t_i \to s_i)}$$

$$\text{(app)}\ \frac{e : t_1 \to t_2 \quad e' : t_1}{e(e') : t_2} \qquad \text{(case)}\ \frac{e : t \quad \forall i \in I\, \big(t \wedge \tau_i \smallsetminus (\bigvee_{j<i} \tau_j) \not\leq \mathbb{0} \Rightarrow e_i : t'\big)}{\texttt{case}\ e\ (\tau_i \to e_i)_{i \in I} : t'}\ t \leq \bigvee_{i \in I} \tau_i$$

$$\text{(proj)}\ \frac{e' : \bigvee_{i \in K} i \quad e : \{t_0, \ldots, t_n, ..\}}{\pi_{e'}\, e : \bigvee_{i \in K} t_i}\ K \subseteq [0, n] \qquad (\text{proj}_\omega)\ \frac{e' : \texttt{int} \quad e : \{t_0, \ldots, t_n\}}{\pi_{e'}\, e : \bigvee_{i \leq n} t_i}$$

$$(\text{proj}_\omega^{\mathbb{1}})\ \frac{e' : \texttt{int} \quad e : \texttt{tuple}}{\pi_{e'}\, e : \mathbb{1}} \qquad (+)\ \frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}} \qquad (\leq)\ \frac{e : t_1 \quad t_1 \leq t_2}{e : t_2}$$

Fig. 3. Declarative static type system

equivalent to $? \wedge \{\mathbb{1}, \mathbb{1}\}$ (the type of values that can be of any subtype of 2-tuples). Following Lanvin [29], gradual subtyping is defined by two uses of static (semantic) subtyping

$$\textit{Subtyping}: \qquad \tau_1 \leq \tau_2 \qquad \iff \qquad (\tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow}) \quad \text{and} \quad (\tau_1{}^{\Uparrow} \leq \tau_2{}^{\Uparrow})$$

A type is *less precise* or *materializes* into another if the latter can be obtained by replacing some of the former's occurrences of ? by other types. This is defined as:

$$\textit{Precision}: \qquad \tau_1 \preccurlyeq \tau_2 \qquad \iff \qquad (\tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow}) \quad \text{and} \quad (\tau_2{}^{\Uparrow} \leq \tau_1{}^{\Uparrow})$$

Finally, consistent subtyping relates two types that materialize into two other types, in which the former is a subtype of the latter. For example, $(? \vee \texttt{bool} \to \texttt{int})$ is a consistent subtype of $(\texttt{int} \to ?)$ since by materializing both ?'s to `int` we obtain that the former type is a subtype of the latter.

$$\textit{Consistent subtyping}: \qquad \tau_1 \mathrel{\tilde{\leqq}} \tau_2 \qquad \iff \qquad \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Uparrow}$$

Consistent subtyping captures the fact that, in gradual mode, the type-checker works differently: for example, confronted with a function call, it checks whether this application *could* succeed at runtime. Thus, instead of checking directly whether the argument is a subtype of the function's domain, it checks whether the (gradual) types of the argument and the function can materialize into two static types such that the materialized argument is a subtype of the materialized function's domain. In fact, Lanvin [29] proves that $\tau_1 \mathrel{\tilde{\leqq}} \tau_2$ if and only if there exists τ_1' and τ_2' such that $\tau_1 \preccurlyeq \tau_1'$, $\tau_2 \preccurlyeq \tau_2'$, and $\tau_1' \leq \tau_2'$. This leads to a less precise result type that contains the dynamic type. However, this can be mitigated through the use of strong functions, which we introduce next.

The rules that handle gradual programs are presented in Figure 4. *For projections*, rule $(\text{proj}_?)$ checks that types can materialize into correct ones (i.e., `int` for the index, and a tuple type for the tuple), in which case all we can deduce for the projection is the type '?'. However, in rule $(\text{proj}_\star)$, if we know that the expression has type $\{t_1, .., t_n\}$ and that the index materializes into an integer, then it can be typed with '?' intersected with the union of the tuple contents: $? \wedge \bigvee_{i=1..n} t_i$. *For applications*, rule $(\text{app}_?)$ gives what is generally the only sound result for applying a gradually typed function to a gradually typed argument, which is type '?'. For instance, it is unsound to type as `int` the result of applying function $\lambda^{\{\texttt{int} \to \texttt{int}\}}(x).x$ to a dynamic argument because although the function does return integers when given integers (hence, has type $\texttt{int} \to \texttt{int}$), if given a Boolean argument at runtime, it will return a Boolean result. That constitutes a large downside of adding '?' to a type system: it tends to escape and pollute the whole type system, and there is usually no way to statically determine a static return type for a function applied to a dynamic value, unless some runtime type checks are added to the code to enforce the function's signature: this is what current sound gradual typing systems do. This is not doable for us, as our system must only perform static type checking, and not modify the Elixir runtime. However, the VM of Elixir already performs type checks, both implicitly in strong operations such as +, and explicitly with the use of guards by programmers. Our idea is to take them both into account in the type system by endowing types with a new notion of functional type: strong arrows.

$$\textbf{Types} \qquad t \ ::= \ \cdots \mid (t \to t)^{\star}$$

A strong arrow denotes functions that work as usual on their domain, but when applied to an argument outside their domain they either *fail on an explicit runtime type check*, or *return a value of their codomain type*, or diverge. In Section 1.1.2 (line 7) we gave the example of the function `second_strong`, which in our calculus can be encoded as $\lambda^{\{\{\mathbb{1},\texttt{int}\}\to\texttt{int}\}}(x).\,\texttt{case}\,x\ (\{\mathbb{1},\texttt{int}\}\to\pi_1 x)$. It is a function that returns an integer if its argument is a tuple whose second element is an integer and otherwise "fails", that is, it reduces to $\omega_{\text{CaseEscape}}$. The notion of strong arrow is not relevant to a standard static type system, but to a gradual type system where uncertainty is both a problem (modules are not annotated, and the type-checker must infer types) and a feature (some programming idioms are inherently dynamic). The purpose of a strong arrow is then to guarantee that a function, when applied to a dynamic argument, will return a value of a specific type, as seen in rule ($\text{app}_\star$). This rule is only used when static type-checking fails, and it has to preserve the flexibility of the typing, as other functions would then struggle to type-check a fully static return type. Thus, rules annotated by $\star$ introduce '?' in their conclusion, in the form of an intersection. This property, which was described in §1.1.3 of the introduction, is called *dynamic propagation.* Alongside with '?', a static type is propagated to be used by the type-checker to detect type incompatibilities. If an argument of type $(? \wedge \texttt{int})$ is used where a Boolean is expected, a static type error will be raised. And if an argument of type $? \wedge (\texttt{int} \vee \texttt{bool})$ is used where a Boolean is expected, the type-checker in gradual mode will allow it, by considering that the argument could become a Boolean at runtime.

The introduction rule for strong arrows ($\lambda_\star$) requires an auxiliary type-checking judgment $\Gamma \vdash e \mathrel{\mathring{:}} t$ defined in Figure 5. This type system models the type checks performed by the Elixir runtime. Indeed, if $\Gamma \vdash e \mathrel{\mathring{:}} t$, then e either diverges, or fails on a runtime error, that we know of, or evaluates to a value of type t. Therefore, the system requires rules that accept typing programs with $\mathbb{0}$, such as $\texttt{case 42}\ (\texttt{bool} \to 5)$ which directly reduces to $\omega_{\text{CaseEscape}}$. This system is similar to the declarative one of Figure 3, but with additional "escape hatches" that make strong operations permissible no matter the type of their operands. For instance, since + is strong (the Elixir VM checks at runtime that the operands of an addition are both integers), then rule ($+^\circ$) only asks that its terms are well-typed. If the addition does not fail, then it returns an integer (typed as $\texttt{int}\wedge ?$ for dynamic propagation). Other such operations are tuple projection, pattern-matching, and also function application. Using this system, we infer strong function types with rule ($\lambda_\star$); if a function $\lambda^{\{t_1\to t_2\}}(x).\,e$ has type $t_1 \to t_2$, then this type is strong if, with x of type ?, the body e can be checked to have type t_2 (actually $t_2\wedge ?$ for dynamic propagation) using the rules of Figure 5. The rules explicitly allow expressions that are known to fail at compile time. As another example, consider rule (case°) in Figure 5, which does not have an exhaustiveness condition because an escaping expression will not return a value but fail at runtime. Note that, in this rule, if no pattern matches, then any type can be chosen for the result and, thus, the rule ($\text{case}^\circ_{\mathbb{0}}$) here above—which types a case-expression that always fails—is admissible.

$$(\text{case}^\circ_{\mathbb{0}})\ \frac{e \mathrel{\mathring{:}} t \qquad t \wedge \bigvee_i \tau_i \simeq \mathbb{0}}{\texttt{case}\ e\ (\tau_i \to e_i)_{i\in I} \mathrel{\mathring{:}} \mathbb{0}}$$

Remark 2. *It is not possible to deduce intersections of strong arrows, for instance for* $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$*. The reason is that strong arrows describe functions whose behavior is constant outside their domain: they necessarily error or return a value of their precise codomain type. A function of type* $(\texttt{int} \to \texttt{int})^\star$*, when given Booleans, can either error or* return integers*; thus it cannot also have type* $(\texttt{bool} \to \texttt{bool})^\star$.

$$(\mathrm{app}_?)\ \frac{e_1 : t_1 \quad e_2 : t_2}{e_1(e_2) : ?}\ \exists t.\begin{cases} t_1 \mathrel{\tilde{\leq}} t \to \mathbb{1} \\ t_2 \mathrel{\tilde{\leq}} t \end{cases} \qquad (\mathrm{proj}_?)\ \frac{e : t \quad e' : t'}{\pi_{e'}\, e : ?}\ \begin{cases} t \mathrel{\tilde{\leq}} \mathtt{tuple} \\ t' \mathrel{\tilde{\leq}} \mathtt{int} \end{cases}$$

$$(\mathrm{app}_\star)\ \frac{e_1 : (t_1 \to t)^\star \quad e_2 : t_2}{e_1(e_2) : ? \wedge t}\ t_2 \mathrel{\tilde{\leq}} t_1 \qquad (\mathrm{proj}_\star)\ \frac{e : \{t_0, \dots, t_n\} \quad e' : t}{\pi_{e'}\, e : ? \wedge \bigvee_{i \leq n} t_i}\ t \mathrel{\tilde{\leq}} \mathtt{int}$$

$$(\mathrm{case}_\star)\ \frac{e : t \qquad \forall i \in I\ \big((t \wedge \tau_i) \setminus (\bigvee_{j<i} \tau_j) \not\leq \mathbb{0} \Rightarrow e_i : t'\big)}{\mathtt{case}\ e\, (\tau_i \to e_i)_{i \in I} : ? \wedge t'}\ (t \mathrel{\tilde{\leq}} \vee_{i \in I} \tau_i)$$

$$(\mathrm{plus}_\star)\ \frac{e_1 : t_1 \quad e_2 : t_2}{e_1 + e_2 : \mathtt{int} \wedge ?}\begin{cases} t_1 \mathrel{\tilde{\leq}} \mathtt{int} \\ t_2 \mathrel{\tilde{\leq}} \mathtt{int} \end{cases} \qquad (\lambda_\star)\ \frac{\lambda^{\mathbb{I}} x.e : t_1 \to t_2 \qquad x : ? \vdash e \mathrel{\overset{\circ}{\circ}} t_2 \wedge ?}{\lambda^{\mathbb{I}} x.e : (t_1 \to t_2)^\star}$$

Fig. 4. Gradual rules

$$(\mathrm{cst}^\circ)\ \frac{}{c \mathrel{\overset{\circ}{\circ}} c \wedge ?} \qquad (\mathrm{var}^\circ)\ \frac{\dashv x : t}{x \mathrel{\overset{\circ}{\circ}} t} \qquad (\mathrm{tuple}^\circ)\ \frac{\forall i = 1..n.\ (e_i \mathrel{\overset{\circ}{\circ}} t_n)}{\{e_1, .., e_n\} \mathrel{\overset{\circ}{\circ}} \{t_1, .., t_n\}}$$

$$(\lambda^\circ)\ \frac{\forall (t_i \to s_i) \in \mathbb{I}.\, (x : t_i \vdash e \mathrel{\overset{\circ}{\circ}} s_i) \quad x : ? \vdash e \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{\lambda^{\mathbb{I}} x.e \mathrel{\overset{\circ}{\circ}} \bigwedge_{i \in I} (t_i \to s_i)} \qquad (\lambda^\circ_\star)\ \frac{\lambda^{\mathbb{I}} x.e \mathrel{\overset{\circ}{\circ}} t_1 \to t_2 \quad x : ? \vdash e \mathrel{\overset{\circ}{\circ}} t_2 \wedge ?}{\lambda^{\mathbb{I}} x.e \mathrel{\overset{\circ}{\circ}} (t_1 \to t_2)^\star}$$

$$(\mathrm{app}^\circ)\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} t_1 \to t_2 \quad e_2 \mathrel{\overset{\circ}{\circ}} t_1}{e_1(e_2) \mathrel{\overset{\circ}{\circ}} t_2} \qquad (\mathrm{app}^\circ_\star)\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} (t_1 \to t_2)^\star \quad e_2 \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{e_1(e_2) \mathrel{\overset{\circ}{\circ}} t_2 \wedge ?} \qquad (\mathrm{app}^\circ_?)\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} \mathbb{1} \quad e_2 \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{e_1(e_2) \mathrel{\overset{\circ}{\circ}} ?}$$

$$(\mathrm{proj}^\circ)\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} \{t_0, \dots, t_n, ..\} \quad e_2 \mathrel{\overset{\circ}{\circ}} \bigvee_{i \in K} i}{\pi_{e_2}\, e_1 \mathrel{\overset{\circ}{\circ}} \bigvee_{i \in K} t_i}\ K \subseteq [0, n] \qquad (\mathrm{proj}^\circ_{\mathtt{int}})\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} \{t_0, \dots, t_n\} \quad e_2 \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{\pi_{e_2}\, e_1 \mathrel{\overset{\circ}{\circ}} \bigvee_{i \leq n} t_i}$$

$$(\mathrm{proj}^\circ_{\mathbb{1}})\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} \mathbb{1} \quad e_2 \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{\pi_{e_2}\, e_1 \mathrel{\overset{\circ}{\circ}} ?} \qquad (\mathrm{case}^\circ)\ \frac{e \mathrel{\overset{\circ}{\circ}} t \quad \forall i \in I.\ \big((t \wedge \tau_i) \setminus \big(\bigvee_{j<i} \tau_j\big) \not\leq \mathbb{0} \Rightarrow e_i \mathrel{\overset{\circ}{\circ}} t'\big)}{\mathtt{case}\ e\, (\tau_i \to e_i)_{i \in I} \mathrel{\overset{\circ}{\circ}} t'}$$

$$(+^\circ)\ \frac{e_1 \mathrel{\overset{\circ}{\circ}} \mathbb{1} \quad e_2 \mathrel{\overset{\circ}{\circ}} \mathbb{1}}{e_1 + e_2 \mathrel{\overset{\circ}{\circ}} \mathtt{int} \wedge ?} \qquad (\leq^\circ)\ \frac{e \mathrel{\overset{\circ}{\circ}} t_1 \quad t_1 \leq t_2}{e \mathrel{\overset{\circ}{\circ}} t_2}$$

Fig. 5. Strong Type System

To summarize, we have presented in Figures 3, 4, and 5 three declarative systems that work together to model different typing disciplines over programs: Figure 3 presents a fully static discipline, where subtyping is used to check compatibility between types, and function type annotations are enforced. This is what is expected of a fully annotated program. Figure 4 only comes into play when the previous type-checking fails. It uses a more relaxed relation on types, consistent subtyping, to check programs whose types are gradual (i.e., where '?' occurs in them). Figure 5 serves as an auxiliary system to infer strong function types, but its elaboration mirrors the semantics of the Beam VM: every syntactically correct program typechecks with type $\mathbb{1}$, since it either diverges, returns a value (necessarily of type $\mathbb{1}$), or fails due to VM checks. However, some programs have more precise types which are passed around like information to be used later.

With this clear distinction, we formulate three type safety results that depend on whether the unsafe ω-rules or the gradual rules are used to type expressions.

Theorem 2.1 (Soundness). *For every expression e and type t such that $\varnothing \vdash e : t$ is derived without using any ω or gradual rules, either there exists a value $v : t$ such that $e \hookrightarrow^* v$, or e diverges.*

Theorem 2.2 (ω-Soundness)**.** For every expression e and type t such that $\varnothing \vdash e : t$ is derived using ω-rules but no gradual rules, either e diverges, or $e \hookrightarrow^* v$ with $v : t$, or $e \hookrightarrow^* \omega_{\textsc{OutOfRange}}$.

Theorem 2.3 (Gradual Soundness)**.** For every expression e and type t such that $\varnothing \vdash e : t$, either there is a value v such that $e \hookrightarrow^* v$ and $v \between t$, or there exists p such that $e \hookrightarrow^* \omega_p$, or e diverges.

The first theorem states that, in the absence of gradual typing, if no warning is emitted, then we are in a classic static typing system. If a warning is raised but gradual typing is still not used, then the second theorem states that the only possible runtime failure is the out-of-range selection of a tuple. If gradual typing is used, then Theorem 2.3 states that any resulting value will have the shape described by the inferred type. For instance, if the type t deduced for a given expression is '`int`' then any value the expression reduces to is necessarily an integer; if it is '?', then the value can be any value; if it is '$? \to ?$', then the value will be a λ-abstraction. Note that, while the first two theorems ensure that well-typed expressions produce only well-typed values of the same type, the third theorem ensures only that any value produced by the expression will satisfy $v \between t$, that is, that it will have the expected shape: because of weak-reduction, a gradually-typed expression can return a λ-abstraction whose body is not well-typed—though, it will be (type) safe in every context. Thus, in particular, if the expression is of type $t_1 \to t_2$, then Theorem 2.3 ensures that it can only return values that are λ-abstractions annotated by (a subtype of) of $t_1 \to t_2$.

3 Guard Analysis

Section 2 shows how to handle dynamic types in languages with explicit type tests. However, our focus is on exploring languages that use patterns and guards rather than relying solely on type tests. These are more general, as type cases can be encoded as guard type-tests on capture variables.

In Elixir, patterns are non-functional values containing capture variables, whereas guards consist of complex expressions formed by boolean combinations (`and`, `or`, `not`) from a limited set of expressions such as type tests (`is_integer`, `is_atom`, `is_tuple`, etc.), equality tests (`==`, `!=`), comparisons (`<`, `<=`, `>`, `>=`), data selection (`elem`, `hd`, `tl`, *map.key*), and size functions (`tuple_size`, `map_size`, `length`). The complete syntax for patterns and guards can be found in Appendix A, Figure 10. To define our typed guard analysis, we introduce a simplified syntax for patterns and guards in Figure 6. The revised syntax for case expressions is $\texttt{case}\ e\ \overline{pg \to e}$, where e represents the expression being matched and $\overline{pg \to e}$ denotes a list of branches. Each branch consists of a pattern-guard pair pg and the corresponding expression e to be executed. Additionally, we introduce a new expression $\texttt{size}\, e$ to calculate the size of a tuple, applicable in both expressions and guards, enhancing the complexity of guards to gauge the precision of our analysis.

Guards are constructed using three identifiers: guard atoms a, representing simplified expressions, type tests ($a \mathbin{?} \tau$), and comparisons ($a = a$, $a \neq a$), which are combined using boolean operators defined in g. The test types τ have been expanded to include union $\tau \vee \tau$ and negation $\neg\tau$, allowing for more expressive tests. For instance, it is now possible to verify whether a variable x is either an integer or a tuple ($x \mathbin{?} \texttt{int} \vee \texttt{tuple}$) or to ascertain that it is not a tuple ($x \mathbin{?} \neg\texttt{tuple}$).

This syntax closely resembles Elixir's concrete syntax; the main difference is that `not` is absent from guards, which is not restrictive: in Section A we detail a translation that eliminates it. To improve readability, we will sometimes use (p `when` g) to denote the pattern-guard pair pg.

The operational semantics is extended in Figure 8 to account for pattern-matching and the size operator. The updated evaluation contexts and a new guard evaluation contexts are defined as:

$$
\begin{array}{lll}
\textbf{Context} & \mathcal{E} & ::= \cdots \mid \texttt{size}\ \mathcal{E} \mid \texttt{case}\ \mathcal{E}\ \overline{pg \to e} \\
\textbf{Guard Context} & \mathcal{G} & ::= \square \mid \mathcal{G}\ \texttt{and}\ g \mid \mathcal{G}\ \texttt{or}\ g \mid \mathcal{G}\ ?\ t \mid \mathcal{G} = a \mid v = \mathcal{G} \mid \mathcal{G}\ \texttt{!=}\ a \mid v\ \texttt{!=}\ \mathcal{G}
\end{array}
$$

$$\begin{array}{llll}
\textbf{Exprs} & e & ::= & \ldots \mid \texttt{case}\ e\ \overline{pg \longrightarrow e} \mid \texttt{size}\, e \\
\textbf{Patterns} & p & ::= & c \mid x \mid \{\overline{p}\} \\
\textbf{Guards} & g & ::= & a \mathrel{?} \tau \mid a = a \mid a \mathrel{!=} a \\
 & & \mid & g \mathrel{\texttt{and}} g \mid g \mathrel{\texttt{or}} g \\
\textbf{Atoms} & a & ::= & c \mid x \mid \{\overline{a}\} \mid \pi_a\, a \mid \texttt{size}\, a \\
\textbf{Tests} & \tau & ::= & c \mid b \mid \{\overline{\tau}\} \mid \{\overline{\tau}, ..\} \\
 & & \mid & \tau \vee \tau \mid \neg\, \tau
\end{array}$$

(where variables occur at most once in each pattern)

Fig. 6. Pattern Matching Syntax

$$\begin{array}{rlr}
v/c = & \{\} & \text{if } v = c \\
v/x = & \{x \mapsto v\} & \\
\{v_1, \ldots, v_n\}/\{p_1, \ldots, p_n\} = & \bigcup_{i=1}^{n} \sigma_i & \text{if } v_i/p_i = \sigma_i \\
 & & \text{for all } i = 1..n \\
v/p = & \texttt{fail} & \text{otherwise} \\
v/(pg) = & \sigma & \text{if } v/p = \sigma \text{ and} \\
 & & g\,\sigma \hookrightarrow^{*} \text{true} \\
v/(pg) = & \texttt{fail} & \text{otherwise}
\end{array}$$

Fig. 7. Definitions of v/p and $v/(pg)$

$$\begin{array}{llcll}
[\textsc{Case}] & \texttt{case}\, v\ \texttt{do}\ (p_i g_i \longrightarrow e_i)_{i<n} & \hookrightarrow & e_j\, \sigma & \text{if } v/(p_j g_j) = \sigma \text{ and} \\
 & & & & \text{for all } i < j < n \quad v/(p_i g_i) = \texttt{fail} \\
[\textsc{Case}_\omega] & \texttt{case}\, v\ \texttt{do}\ (p_i g_i \longrightarrow e_i)_{i<n} & \hookrightarrow & \omega_{\textsc{CaseEscape}} & \text{if } v/p_i g_i = \texttt{fail} \text{ for all } i < n \\
[\textsc{Size}] & \texttt{size}\ \{v_1, .., v_n\} & \hookrightarrow & n & \\
[\textsc{Size}_\omega] & \texttt{size}\ v & \hookrightarrow & \omega_{\textsc{Size}} & \text{if } v \neq \{\overline{v}\}
\end{array}$$

$$\begin{array}{lrcll}
[\textsc{And}_\top] & \texttt{true and}\ g & \lhook\joinrel\longrightarrow & g & \\
[\textsc{And}_\bot] & v\ \texttt{and}\ g & \lhook\joinrel\longrightarrow & \texttt{false} & \text{if } v \neq \texttt{true} \\
[\textsc{Or}_\top] & \texttt{true or}\ g & \lhook\joinrel\longrightarrow & \texttt{true} & \\
[\textsc{Or}_\bot] & \texttt{false or}\ g & \lhook\joinrel\longrightarrow & g & \\
[\textsc{Eq}_\top] & v = v' & \lhook\joinrel\longrightarrow & \texttt{true} & \text{if } v = v' \\
[\textsc{Eq}_\bot] & v = v' & \lhook\joinrel\longrightarrow & \texttt{false} & \text{else}
\end{array}
\qquad
\begin{array}{lrcll}
[\textsc{NotEq}_\top] & v \mathrel{!=} v' & \lhook\joinrel\longrightarrow & \texttt{true} & \text{if } v \neq v' \\
[\textsc{NotEq}_\bot] & v \mathrel{!=} v' & \lhook\joinrel\longrightarrow & \texttt{false} & \text{else} \\
[\textsc{OfType}_\top] & v \mathrel{?} t & \lhook\joinrel\longrightarrow & \texttt{true} & \text{if } v \in t \\
[\textsc{OfType}_\bot] & v \mathrel{?} t & \lhook\joinrel\longrightarrow & \texttt{false} & \text{else} \\
[\textsc{Context}_g] & \mathcal{G}[g] & \lhook\joinrel\longrightarrow & \mathcal{G}[g'] & \text{if } g \lhook\joinrel\longrightarrow g' \\
[\textsc{Context}_a] & \mathcal{G}[a] & \lhook\joinrel\longrightarrow & \mathcal{G}[a'] & \text{if } a \hookrightarrow a' \\
[\textsc{Context}_\omega] & \mathcal{G}[a] & \lhook\joinrel\longrightarrow & \texttt{false} & \text{if } a \hookrightarrow \omega_p
\end{array}$$

Fig. 8. Pattern Matching and Guard Reductions

This semantics relies on matching values to patterns v/p and values to guarded patterns $v/(pg)$, as defined in Figure 7. When v is a value and p a pattern, v/p results in an environment σ that assigns capture variables in p to corresponding matching values occurring in v. For example, v/x returns the environment where x is bound to v, and $\{v_1, v_2\}/\{x, y\}$ yields $x \mapsto v_1, y \mapsto v_2$. Similarly, $v/(pg)$ creates such an environment but also verifies that the guard g evaluates to true within the created environment; if v does not match p or the guard condition fails, the operation returns the token fail. For instance, $v/(x \texttt{ when } x \mathrel{?} \texttt{int})$ results in $x \mapsto v$ if v is an integer and fail otherwise.

An important aspect of the semantics of pattern matching is that within a specific branch, if a reduction results in an error (i.e., reduces to an ω), the entire guard is considered to fail, and that branch is discarded. For instance, consider the guard ($\texttt{size}\, x = 2 \texttt{ or } x \mathrel{?} \texttt{int}$). If x is not a tuple, taking its size will lead to an error. Consequently, even if x is an integer, the guard will evaluate to false (as specified by rule [$\textsc{Context}_\omega$]) instead of true as could be expected from the disjunction.

It should also be noted that, in Elixir, both 'and' and 'or' operators exhibit short-circuit behavior. Specifically, if the left-hand side of an and-guard evaluates to false, the right-hand side is not evaluated (as specified by rule [$\textsc{And}_\bot$]); similarly, if the left-hand side of an or-guard evaluates to true, the right-hand side is not evaluated (as defined by rule [$\textsc{Or}_\bot$]).

3.1 Typing Pattern Matching

To type the expression case $e\ (p_i g_i \longrightarrow e_i)_{i \leq n}$ we aim to precisely type each branch's expression e_i by analyzing the set of values for which the pattern-guard pair $p_i g_i$ succeeds, that is, $\{v \in \textbf{Values} \mid v \text{ matches } p_i g_i\}$. However, not every such set of values corresponds perfectly to a type. For example, we have seen in §1.1.4 the guard in line 18—expressed in our formalism by the pattern-guard pair (x `when` ($\pi_0\, x$? `bool`) `or` ($\pi_0\, x = \pi_1\, x$))—, which matches tuples with either a Boolean as the first element, or whose first two elements are identical, yet no specific type denotes all such tuples. To address this, we define an approximation using two types, termed as the *potentially accepted type* $\llparenthesis pg \rrparenthesis$ (in our example it is $\{\texttt{bool}, ..\} \vee \{\mathbb{1}, \mathbb{1}, ..\}$ since any value accepted by the pattern will belong to this type) and the *surely accepted type* $\rrparenthesis pg \llparenthesis$ (here it is $\{\texttt{bool}, ..\}$ since all tuples starting with a Boolean are surely accepted) for pair pg. Through these types, we derive an approximating type t_i encompassing all values reaching e_i. When the matched expression is of type t, t_i is formulated as $(t \wedge \llparenthesis p_i g_i \rrparenthesis) \setminus \bigvee_{j<i} \rrparenthesis p_j g_j \llparenthesis$. In words, the values in t_i are those that may be produced by e (i.e., those in t), and may be captured by $p_i g_i$ (i.e., those $\llparenthesis p_i g_i \rrparenthesis$) and which are not surely captured by a preceding branch (i.e., minus those in $\rrparenthesis p_j g_j \llparenthesis$ for all $j < i$). This type t_i can be utilized to generate the type environment under which e_i is typed. This environment, denoted as ${}^{t_i}/_{p_i}$, assigns the deducible type of each capture variable of the pattern p_i, assuming the pattern matches a value in t_i. The definition of this environment is a standard concept in semantic subtyping and is detailed in Appendix E, Figure 28. A first approximation of the typing can be given by the following rule (given here only for presentation purposes but not included in the system):

$$(\text{case}_\omega(\text{coarse}))\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ \ (t_i \not\leq \mathbb{0} \ \ \Rightarrow\ \ \Gamma, {}^{t_i}/_{p_i} \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ (p_i g_i \rightarrow e_i)_{i\leq n}\ : s}\ \ \begin{array}{l} t_i = (t \wedge \llparenthesis p_i g_i \rrparenthesis) \setminus \bigvee_{j<i} \rrparenthesis p_j g_j \llparenthesis \\ t \leq \bigvee_{i\leq n} \llparenthesis p_i g_i \rrparenthesis \end{array}$$

The rule types a case-expression of n branches. For the i-th branch with pattern p_i and guard g_i, it computes t_i and produces the type environment ${}^{t_i}/_{p_i}$. This environment is used to type e_i only if $t_i \not\leq \mathbb{0}$, thus ensuring that some values may reach the branch and checking for case redundancy. The side condition $t \leq \bigvee_{i<n} \llparenthesis p_i g_i \rrparenthesis$ ensures that every value of type t may *potentially* be captured by some branch, addressing exhaustiveness. The rule is labeled with ω, indicating a potential warning, as the union $\bigvee_{i\leq n} \llparenthesis p_i g_i \rrparenthesis$ might over-approximate the set of captured values. However, if $t \leq \bigvee_{i\leq n} \rrparenthesis p_i g_i \llparenthesis$ holds, then there's no warning (cf. rule (case) in Figure 30, Appendix E), since all values in $\bigvee_{i\leq n} \rrparenthesis p_i g_i \llparenthesis$ are captured by some pattern-guard pair, and so are those in t.

The typing rule for case expressions is actually more complicated than the one above, since it performs a finer-grained analysis of Elixir guards that is also used to compute their surely/potentially accepted types. Let us look at it in detail:

$$(\text{case}_\omega)\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0} \ \ \Rightarrow\ \ \Gamma, {}^{t_{ij}}/_{p_i} \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ (p_i g_i \rightarrow e_i)_{i\leq n}\ : s}\ t \leq \bigvee_{i\leq n} \llparenthesis p_i g_i \rrparenthesis$$

$$\textit{where}\ \Gamma\,; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$$

In contrast to the prior rule, the system now computes a list of types $t_{i1}, \ldots, t_{im_i}$ for each $p_i g_i$ pair, which partitions the earlier t_i. The rule types each e_i expression m_i-times, each with a distinct environment ${}^{t_{ij}}/_{p_i}$. The t_{ij} values are derived from an auxiliary deduction system: $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$. This system, detailed in the rest of this section, inspects each g_i for OR-clauses, generating pairs $(t, \mathfrak{b})$ that indicate the clause's type t and a Boolean flag $\mathfrak{b}$ indicating its exactness. For example, the guard ($\pi_0\, x$? `bool` `or` $\pi_0\, x = \pi_1\, x$) of our example produces pairs ($\{\texttt{bool}, ..\}$, `true`) and ($\{\mathbb{1}, \mathbb{1}, ..\}$, `false`): the first flag is `true` since the type is exact; the second flag is `false` since the type is an approximation. The guard (x ? `int` `or` $\pi_0\, x$? `int`) instead will produce (`int`, `true`) and ($\{\texttt{int}, ..\}$, `true`). Guards are parsed in Elixir's evaluation order and potential clause failures. Analysis of guard g_i needs both Γ and p_i as it might use variables from either.

Given $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$, the potentially accepted type for $p_i g_i$ is the union of all t_{ij}'s, while the surely accepted type for $p_i g_i$ is the union of all t_{ij}'s for which $\mathfrak{b}_{ij}$ is true. Thus, we have $\llparenthesis p_i g_i \rrparenthesis = \bigvee_{j\leq m_i} t_{ij}$ and $\rrparenthesis p_i g_i \llparenthesis = \bigvee_{\{j\leq m_i \mid \mathfrak{b}_{ij}\}} t_{ij}$. In our example, if g is the guard

($\pi_0\, x \mathrel{?} \texttt{int} \texttt{ or } \pi_0\, x = \pi_1\, x$), then $⟅xg⟆ = \{\texttt{bool},..\} \vee \{\mathbb{1},\mathbb{1},..\}$ and $⟆xg⟅ = \{\texttt{bool},..\}$. Conversely, if g is the guard ($x \mathrel{?} \texttt{int} \texttt{ or } \pi_0\, x \mathrel{?} \texttt{int}$), then the potentially and surely accepted types of xg are the same, both being $\texttt{int} \vee \{\texttt{int},..\}$, indicating that the approximation is exact.

When using this analysis, type safety depends on the side conditions used. Rule (case) with $t \leq \bigvee_{i\leq n} ⟆p_i g_i⟅$ is safe for exhaustiveness, ensuring the same static guarantee as Theorem 2.1:

Theorem 3.1 (Static Soundness). *If $\varnothing \vdash e : t$ is derived with the ω-free rules of Figure 3 and the (case) rule with condition $t \leq \bigvee_{i\leq n} ⟆p_i g_i⟅$, then either $e \hookrightarrow^* v$ with $v : t$, or e diverges.*

Rule (case$_\omega$) above will be used whenever our guard analysis is too imprecise to type a correct program, raising a warning and adding $\omega_{\text{CaseEscape}}$ to the set of explicit runtime errors in Theorem 2.2. The gradual rule (case$_\star$) simply checks that the scrutinee may be covered by some patterns, and does not modify the type safety of Theorem 2.3. See Theorems E.4, E.5, and E.6 in Appendix E.1.

Remark 3 (Naive Type Narrowing). *In practice, if e is being matched, its skeleton $\texttt{sk}(e)$ (which is a pattern that matches the structure and variables of that expression while leaving out any functional or constant parts – see Definition E.1) is added to patterns. Thus, any type narrowing that occurs in the guard analysis is also applied to the variables of e. This is possible by adding intersections—noted &—to patterns: a value matches the intersection pattern $p_1\&p_2$ iff it matches both p_1 and p_2; now every pattern can be compiled as $\texttt{sk}(e)\&p$. Then, we handle dependencies between variables (e.g., if pattern $x\&\{y,z\}$ is followed by a guard $y \mathrel{?} \texttt{int}$, then the type of x is refined to $\{\texttt{int},\mathbb{1}\}$), using an environment update $\Gamma[x \mathrel{\hat{=}} t]_p$ (see next section) that narrows the type of x in Γ to t, and uses pattern p to properly refine the type of other variables in Γ that depend on x. The pattern p can then simply be passed around alongside Γ in the guard analysis judgments. Since it is a global dependency (no change is ever made to p), it is not necessary to propagate it in the typing rules, and we omitted it for clarity.*

3.2 An Overview of Guard Analysis.

In the rest this section we illustrate how to derive $\Gamma; t \vdash (p_i g_i)_{i\leq n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\leq n, j\leq m_i}$. The analysis of a single guard is expressed via a judgment of the form $\Gamma \vdash g \mapsto \mathcal{R}$, where $\mathcal{R}$ is defined as follows:

Results	$\mathcal{R}$	$::= \overline{T} \mid \omega$	where $T ::= \{\mathscr{S}\,;\mathscr{T}\} \mid \{\mathscr{S}\,;\texttt{false}\}$
Environments	$\mathscr{S},\mathscr{T}$	$::= (\Gamma,\mathfrak{b})$	
Failure Results	$\mathcal{F}$	$::= \omega \mid \overline{\{\mathscr{S}\,;\texttt{false}\}}$	

Pairs of environments, $\{\mathscr{S}\,;\mathscr{T}\}$, form the basis for our guard analysis. The first element, $\mathscr{S} = (\Gamma,\mathfrak{b})$, represents the *safe environment*, in which Γ (a mapping from variables to types) gives a necessary condition on the types of the variables in the guard, such that the guard does not error (i.e., does not evaluate to ω). This condition is sufficient only if the Boolean flag $\mathfrak{b}$ is $\texttt{true}$. Similarly, $\mathscr{T}$ denotes the *success environment*, which is a necessary condition on the type of the variables for the guard to evaluate to $\texttt{true}$. The fact that an environment is *sufficient* or not is mainly a matter of precision, because the system needs to find out the exact type of all the values that make a guard succeed. For example, for guard ($\texttt{size}\ x = 2$), the exact success set are all the tuples of size 2, thus its success environment is $(x : \{\mathbb{1},\mathbb{1}\},\texttt{true})$. Conversely, if such a type cannot be found, then the environment is an approximation: for instance, if the guard ($x \mathrel{?} \texttt{int} \texttt{ and } x > y$) succeeds, then x is an integer—but there's no way in our system to encode by a type the fact that $(x > y)$. Hence, the corresponding environment is $((x{:}\texttt{int}, y{:}\mathbb{1}),\texttt{false})$. The reason why y is not also guaranteed to be an integer is that Elixir allows comparing every two values (> is a total order on values).

The analysis of a guard g thus produces a list of pairs $\{\mathscr{S}\,;\mathscr{T}\}$, each describing one way for the guard to succeed. For instance, the analysis of a guard g_1 or g_2 produces one pair that describes type conditions that make g_1 succeed, and a second pair that describes the conditions for g_1 to reduce to $\texttt{false}$ *without erroring*, and for g_2 to succeed. Our analysis also finds faulty guards, either because they always error (i.e., evaluate to ω), or because they fail (i.e., evaluate to `false`) within safe-environment $\mathscr{S}$; the meta-variable $\mathcal{F}$ captures both cases.

Appendix E gives the complete rules for guard analysis. Hereafter, we comment only on the most significant ones by unrolling a series of examples. Consider the guard ($\{x, y\}$ `when` x ? `tuple`), which performs a type test on a capture *variable*. The rule that handles this case is [Var] given here on the right, where the notation $\Gamma[x \mathrel{\hat{=}} t]$ denotes the environment obtained from Γ after refining the typing of x with t (i.e., ascribing it to $\Gamma(x) \wedge t$: the complete definition can be found in Appendix E, Figure 29). This produces the judgement

$$[\textsc{var}]\ \frac{\Gamma(x) \not\leq t \qquad \Gamma(x) \wedge t \not\simeq \mathbb{0}}{\Gamma \vdash x \,?\, t \mapsto \{(\Gamma, \texttt{true})\,;\,(\Gamma[x \mathrel{\hat{=}} t], \texttt{true})\}}$$

$$(x\!:\!\mathbb{1}, y\!:\!\mathbb{1}) \vdash (x \,?\, \texttt{tuple}) \mapsto \{((x\!:\!\mathbb{1}, y\!:\!\mathbb{1}), \texttt{true})\ ;\ ((x\!:\!\texttt{tuple}, y\!:\!\mathbb{1}), \texttt{true})\}$$

in which the first element of the result leaves the variables unchanged, since this guard cannot error (paired with Boolean flag `true`, since this analysis is exact), while the second element containing $(x\!:\!\texttt{tuple}, y\!:\!\mathbb{1})$ indicates that the guard will succeed if and only if x is a tuple (and this condition is also sufficient as indicated by the Boolean flag `true`). If we refine this guard with a *conjunction*

$$\{x, y\}\ \texttt{when}\ (x \,?\, \texttt{tuple})\ \texttt{and}\ (\texttt{size}\ x = 2)$$

now it specifically matches tuples of size 2, and its analysis is done by rule [And]:

$$[\textsc{and}]\ \frac{\Gamma \vdash g_1 \mapsto \{(\Phi_1, \mathfrak{b}_1)\,;\,(\Delta_1, \mathfrak{c}_1)\} \quad \Delta_1 \vdash g_2 \mapsto \{(\Phi_2, \mathfrak{b}_2)\,;\,(\Delta_2, \mathfrak{c}_2)\}}{\Gamma \vdash g_1\ \texttt{and}\ g_2 \mapsto \{\mathscr{S}\,;\,(\Delta_2, \mathfrak{c}_1 \,\&\, \mathfrak{c}_2)\}} \qquad \mathscr{S} = \begin{cases}(\Phi_1, \mathfrak{b}_1) & \text{if } \mathfrak{b}_2 = \texttt{true} \text{ and } \Phi_2 = \Delta_1\\ (\Phi_2, \mathfrak{b}_1 \,\&\, \mathfrak{b}_2) & \text{otherwise}\end{cases}$$

In this rule, the success environment produced by the analysis of the first component x ? `tuple` of the and (in our case, $\Delta_1 = (x : \texttt{tuple}, y : \mathbb{1})$) is used to analyze the second component ($\texttt{size}\ x = 2$), which is then handled by successive uses of the rules [Eq$_2$] and [Size]:

$$[\textsc{eq}_2]\ \frac{\Gamma \vdash a_2 : c \quad \Gamma \vdash a_1 \,?\, c \mapsto \{\mathscr{S}\,;\,\mathscr{T}\}}{\Gamma \vdash a_1 = a_2 \mapsto \{\mathscr{S}\,;\,\mathscr{T}\}} \qquad [\textsc{size}]\ \frac{\Gamma \vdash a \,?\, \texttt{tuple} \mapsto \{\_\,;\,\mathscr{T}\} \quad \Gamma \vdash a \,?\, \texttt{tuple}^i \mapsto \{\_\,;\,\mathscr{T}'\}}{\Gamma \vdash \texttt{size}\, a \,?\, i \mapsto \{\mathscr{T}\,;\,\mathscr{T}'\}}$$

where $\texttt{tuple}^i$ is the type of all the tuples of size i. Rule [Eq$_2$] corresponds to the best-case scenario of a guard equality: when one of the terms has a singleton type ($\Gamma \vdash a_2 : c$), a sufficient condition for both terms to be equal is that the other term gets this type as well ($\Gamma \vdash a_1\ ?\ c$). In our example, this means doing the analysis $\Gamma \vdash \texttt{size}\ x\ ?\ 2$ with rule [Size]. This rule asks two questions (i.e., checks two premises): "can x be a tuple" (this produces a non-erroring environment), and "can x be a tuple of size 2?" (which in our case refines x to be of type $\{\mathbb{1}, \mathbb{1}\}$). The most general versions of these rules make approximations and can be found in Appendix E (Figure 25).

To go further, we can check that the second element of this tuple has type `int`, by adding another conjunct to the guard: $\{x, y\}$ `when` (x ? `tuple`) `and` ($\texttt{size}\ x = 2$) `and` ($\pi_1\, x$? `int`). Now, rule [Proj] applies:

$$[\textsc{proj}]\ \frac{\Gamma \vdash a' : i \quad \Gamma \vdash a\ ?\ \texttt{tuple}^{>i} \mapsto \{\_;\,(\Delta, \mathfrak{b})\} \quad \Delta \vdash a\ ?\ \{\,\overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{i \text{ times}}\,, t\,, ..\} \mapsto \mathscr{T}}{\Gamma \vdash \pi_{a'}\, a\ ?\ t \mapsto \{(\Delta, \mathfrak{b})\ ;\,\mathscr{T}\}}$$

where $\texttt{tuple}^{>i}$ represents tuples of size greater than i (e.g., $\texttt{tuple}^{>1} = \{\mathbb{1}, \mathbb{1}\,, ..\}$). This rule reads from left to right: after checking that the index is a singleton integer i (in our example, 1), the non-erroring environment is computed by checking that the tuple has more than i elements. In our example, ($\texttt{size}\ x = 2$) has already refined x to be of type $\{\mathbb{1}, \mathbb{1}\}$. Finally, the success environment checks that the tuple is of size greater than i with t in i-th position (in our example, it has type $\{\mathbb{1}, \texttt{int}\,, ..\}$); since x was a tuple of size two, the intersection of those two types is $\{\mathbb{1}, \texttt{int}\}$.

In the case of a disjunction, a guard can succeed if its first component succeeds, or if the first fails (but does not error) and the second succeeds (guards being evaluated in a left-to-right order). Consider $\{x, y\}$ `when` (x ? `tuple`) `and` ($\texttt{size}\ x = 2$) `or` (y ? `bool`) whose analysis uses rule [Or]

$$[\textsc{or}]\ \frac{\Gamma \vdash g_1 \mapsto \{(\Phi_1, \mathfrak{b}_1)\,;\,(\Delta_1, \mathfrak{c}_1)\} \qquad \Gamma, t_i/_p \vdash g_2 \mapsto \{(\Phi_2, \mathfrak{b}_2)\,;\,(\Delta_2, \mathfrak{c}_2)\}}{\Gamma \vdash g_1 \,\texttt{or}\, g_2 \mapsto \{(\Phi_1, \mathfrak{b}_1)\,;\,(\Delta_1, \mathfrak{c}_1)\}, \{\mathcal{S}\,;\,(\Delta_2, \mathfrak{c}_1 \,\&\, \mathfrak{c}_2)\}} \qquad \mathcal{S} = \begin{cases} (\Phi_1, \mathfrak{b}_1) \text{ if } (\mathfrak{b}_2 = 1) \text{ and } \left(\Phi_2 = \Gamma, t_i/_p\right) \\ (\Phi_2, \mathfrak{b}_1 \,\&\, \mathfrak{b}_2) \ \text{ otherwise} \end{cases}$$

$$t_i = \begin{cases} \llparenthesis p \rrparenthesis_{\Phi_1} \setminus \llparenthesis p \rrparenthesis_{\Delta_1} & \text{if } \mathfrak{c}_1 = 1 \\ \llparenthesis p \rrparenthesis_{\Phi_1} & \text{if } \mathfrak{c}_1 = 0 \end{cases}$$

The first term of the or, that is, $(x \,?\, \texttt{tuple})$ `and` $(\texttt{size}\ x = 2)$, is analyzed with rule [And] given before, which produces $\{((x{:}\mathbb{1}, y{:}\mathbb{1}), \texttt{true})\ ;\ ((x{:}\{\,\mathbb{1}, \mathbb{1}\,\}, y{:}\mathbb{1}), \texttt{true})\}$. The second term, thus, is analyzed under the environment $(x{:}\neg\{\,\mathbb{1}, \mathbb{1}\,\}, y{:}\mathbb{1})$ which is obtained by subtracting the success environment of the first guard from its non-erroring one (i.e., we realize that, since tuples of size two make the first guard succeed, they will never reach the second guard). This is done by computing the type

$$t = \llparenthesis \{\,x, y\,\} \rrparenthesis_{x:\mathbb{1}, y:\mathbb{1}} \setminus \llparenthesis \{\,x, y\,\} \rrparenthesis_{x:\{\,\mathbb{1},\mathbb{1}\,\}, y:\mathbb{1}} = \{\,\mathbb{1}, \mathbb{1}\,\} \setminus \{\,\{\,\mathbb{1}, \mathbb{1}\,\}, \mathbb{1}\,\} = \{\,\neg\{\,\mathbb{1}, \mathbb{1}\,\}, \mathbb{1}\,\}$$

where the notation $\llparenthesis p \rrparenthesis_\Gamma$ (defined in Appendix E, Figure 27) denotes the type of values that are accepted by a pattern p *and* which, when matched against p, bind the capture variables of p to types in Γ (e.g., $\llparenthesis \{\,x, y\,\} \rrparenthesis_{x:\texttt{int}, y:\texttt{bool}} = \{\,\texttt{int}, \texttt{bool}\,\}$). This choice of t is motivated by the fact that the analysis of the first term is *exact* (since the Boolean flag is `true`), therefore it is safe to assume that the values that make the first guard succeed, never end up in the second guard. Because this is a disjunction, the two ways that the guard succeeds are not mixed into a single environment, but split into two distinct solutions that are concatenated. Then, a little of administrative work on the Boolean flags ensures which results are exact and which are not.

So far our guards could not error, but it is a common feature in Elixir that guards that error short-circuit a branch of a case expression. For example, the guard

$$\{x, y\}\ \texttt{when}\ (\texttt{size}\ x = 2)\ \texttt{or}\ x \,?\, \texttt{bool}$$

only succeeds when the first projection of the matched value is a tuple of size two, and fails for all other values *including* when the first projection is a boolean (in which case `size` raises an error). This is handled by rule [Or] as well, by considering the non-erroring environment of a guard and using it as a base to analyze the second term of a disjunction. In our example, the non-erroring environment is $(x : \texttt{tuple}, y : \mathbb{1})$, and the second term is found instantly to be false. This could potentially raise a warning, as a part of a guard that only evaluates to false is a sign of a mistake.

In a last processing step, the guard analysis judgment $\Gamma \vdash g \mapsto \mathcal{R}$, is used to derive the judgment $\Gamma; t \vdash (p_i g_i)_{i \le n} \rightsquigarrow (s_{ij}, \mathfrak{b}_{ij})_{i \le n, j \le m_i}$ to be used during the typing of a case expression. Rule [accept] takes care of a single pattern-guard pair, and translates a list of possible success environments $(\Delta_i, \mathfrak{b}_i)_{i \le n}$ into a list of pairs formed by an accepted type and its precision $(\llparenthesis p \rrparenthesis_{\Delta_i}, \mathfrak{b}_i)_{i \le n}$. Guards that always fail are handled by rule [fail].

$$[\textsc{accept}]\ \frac{\Gamma, t/_p \vdash g \mapsto \{\_;\,(\Delta_i, \mathfrak{b}_i)\}_{i \le n}}{\Gamma; t \vdash pg \rightsquigarrow (\llparenthesis p \rrparenthesis_{\Delta_i}, \mathfrak{b}_i)_{i \le n}} \qquad [\textsc{fail}]\ \frac{\Gamma, t/_p \vdash g \mapsto \mathcal{F}}{\Gamma; t \vdash pg \rightsquigarrow (\mathbb{O}, \texttt{true})}$$

The sequence of successive guard-pattern pairs in a case expression is handled by [Sequence], which takes care to refine the possible types as the analysis advances, by subtracting from the potential type t the surely accepted types $\bigvee_{(s,\text{true}) \in \mathcal{A}} s$ of the analysis of the current guard-pattern.

$$[\textsc{sequence}]\ \frac{\Gamma; t \vdash pg \rightsquigarrow \mathcal{A} \qquad \Gamma; t \setminus \left(\bigvee_{(s,\text{true}) \in \mathcal{A}} s\right) \vdash \overline{pg} \rightsquigarrow \overline{\mathcal{A}}}{\Gamma; t \vdash pg\ \overline{pg} \rightsquigarrow \mathcal{A}\ \overline{\mathcal{A}}}$$

This last rule will then be used to produce the auxiliary types $\Gamma\,; t \vdash (p_i g_i)_{i \le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i \le n, j \le m_i}$ used in typing case expressions.

4 Arity (and Strong Arrows)

Function arity plays an important role both in Elixir and in Erlang, being it used to identify functions. This is reflected by the presence among the guards of the test `is_function(f, n)` whose usage we showed for `curry` in lines 23–24. To encompass multi-arity, we extend the syntax as follows:

$$\begin{array}{lll}\textbf{Expressions} & e ::= \cdots \mid \lambda^{\mathbb{I}}\,\overline{x}\,.e \mid e(\,\overline{e}\,) \\ \textbf{Types} & t ::= \cdots \mid \overline{t} \to t\end{array}$$

In the previous sections we assumed the subtyping relation to be given and working out of the box. This is true for all the types we used, except for strong types and, now, non-unary functions. This illustrates a difficulty of semantic subtyping from a practical point of view: it requires some difficult machinery to be implemented, and although this machinery is extensively explained in the literature (e.g., [6, 8, 10, 26]), it is not obvious how to adapt it to specific situations. There are two ways to do so: either by defining an encoding of your custom types into existing types or by extending the machinery to support it. For instance, it is possible to encode functions with a given arity by using a tuple type with two fields: one that contains the arrow type, and one that contains the arity. But this is not always possible: strong arrows require checking properties that are usually not within the scope of the existing theory of semantic subtyping. Thus, the steps required to extend semantic subtyping with a new type consist of:

(1) defining the semantic interpretation of the new type;
(2) deriving from this interpretation the decomposition rules to check subtyping for the new type.

We succinctly describe below these steps for multi-arity functions, assuming the basic definition of semantic subtyping, and defer to Appendix G the corresponding development for strong arrows.

Introducing a new type constructor in the form of multi-arity function type requires an interpretation in the domain of semantic subtyping.

Definition 4.1. Let $X_1, .., X_n$ and Y be subsets of the domain D. We define

$$(X_1, .., X_n) \to Y = \left\{R \in \mathcal{P}_f\,(D^n \times D_\omega) \mid \forall (d_1, .., d_n, \delta) \in R.\ (\forall i \in \{1, ..., n\}.\ d_i \in X_i) \implies \delta \in Y\right\}$$

In a nutshell, the space of multi-arity functions is defined as the set of finite sets of $n+1$-tuples $(d_1, .., d_n, \delta)$ such that if the first n components are in the domain of the function type, then the last component δ is in its codomain. This definition is used to define the interpretation $[\![.]\!]$ of types for multi-arity function types, and define their subtyping relation. In particular, using set-theoretic equivalences, the subtyping problem $t_1 \le t_2$ is simplified to an emptiness checking problem: $t_1 \le t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \setminus [\![t_2]\!] \subseteq \varnothing \iff [\![t_1 \setminus t_2]\!] \subseteq \varnothing$. This emptiness check can itself be decomposed over each disjoint component of a type (i.e., tuples, integers, etc.); such algorithms are described by Castagna [6, Section 4] and are defined for *disjunctive normal forms* of literals ℓ that range over the different possible type components. For multi-arity functions, this means that we have the form $\bigvee_{i\in I}\bigwedge_{j\in J}\ell_{ij}$ where all the ℓ_{ij} are multi-arity arrows $\overline{t} \to t$ or their negations. To simplify the problem of checking that such a form is empty, consider that, according to the interpretation of Definition 4.1, intersections of different arity are empty; so the problem simplifies into checking that every member of the union above, where the literals go over arrows of the same arity $n \in N$, is contained in the union of the negations:

$$\bigwedge_{i\in P}(t_i^{(1)}, ..., t_i^{(n)}) \to t_i \le \bigvee_{j\in N}(t_j^{(1)}, ..., t_j^{(n)}) \to t_j \qquad (1)$$

(note that if any negative arrow were not of arity n, then the subtyping relation above would not hold). Since each literal can be interpreted as a set using Definition 4.1 above, this problem is then reformulated and solved as a set-containment problem in Theorem 4.2 (proof in Appendix F.3) using set manipulation techniques devised by Frisch [24].

Theorem 4.2 (Multi-arity Set-Containment)**.** Let $n \in \mathbb{N}$. Let $(X_i^{(1)})_{i\in P}, .., (X_i^{(n)})_{i\in P}$, $(X_i)_{i\in P}$, $(Y_i^{(1)})_{i\in N}, .., (Y_i^{(n)})_{i\in N}$, $(Y_i)_{i\in N}$ be families of subsets of the domain D. Then,

$$\bigcap_{i\in P}\left(X_i^{(1)}, .., X_i^{(n)}\right) \to X_i \subseteq \bigcup_{j\in N}\left(Y_j^{(1)}, .., Y_j^{(n)}\right) \to Y_j \iff \begin{array}{c}\exists j_0 \in N.\ \textbf{such that}\\ \forall \iota : P \to [1, n+1]\end{array} \begin{cases} \exists k \in [1, n].\ Y_{i_0}^{(k)} \subseteq \bigcup_{\{i\in P \mid \iota(i)=k\}} X_i^{(k)} \\ \textbf{or} \quad \bigcap_{\{i\in P \mid \iota(i)=n+1\}} X_i \subseteq Y_{j_0} \end{cases}$$

This theorem reduces subtyping on multi-arity arrows to multiple smaller subtyping checks on their domain and return types; thus, it enables the definition of a recursive algorithm that decides subtyping. Following Frisch [24], we can define a backtrack-free algorithm that for all $n \in \mathbb{N}$ decides

$$\textstyle\bigwedge_{f\in P} f \ \leq \ (t_1, .., t_n) \to t \tag{2}$$

where P is a set of arrows of arity n. This is expressed by function Φ_n of $n + 2$ arguments defined as:

$$\begin{array}{lcl}\Phi_n(t_1, .., t_n, t, \varnothing) & = & (\exists k \in [1, n].\ t_k \leq \mathbb{0})\ \textbf{or}\ (t \leq \mathbb{0}) \\ \Phi_n(t_1, .., t_n, t, \{(t'_1, .., t'_n) \to t'\} \cup P) & = & (\Phi_n(t_1, .., t_n, t \wedge t', P)\ \textbf{and} \\ & & \quad \forall k \in [1, n].\ \Phi_n(t_1, .., t_k \setminus t'_k, .., t_n, t, P))\end{array}$$

Now, calling $\Phi_n(t_1, .., t_n, \neg t, P)$ decides (2) (see Theorem F.4). Thus, because an intersection of arrows is a subtype of a union if and only if it is a subtype of one of the arrows in the union (which is a Corollary of Theorem 4.2: see the $\exists j_0 \in N$ in the statement), the subtyping problem formulated in (1) consists in finding one negative arrow ($j_0 \in N$) such that $\Phi_n(t_{j_0}^{(1)}, .., t_{j_0}^{(n)}, t_{j_0}, P)$ returns true.
Strong Arrows. *A similar process is required to integrate strong arrows in the semantic subtyping framework; their semantics is in Definition G.1 of the Appendix, followed by a subtyping algorithm G.2.*

5 Inference

The problem of inference for Elixir consists of finding the right type for functions defined by several pattern-matching clauses. Inference appears in this work mainly as a convenience tool: indeed, one could simply decide that every function must be annotated, and inference would not be required. In our case, it is both an interesting research question and a practical one: writing annotations for untyped code is not without effort, and inference can help by suggesting annotations to the programmer. In the case of anonymous functions, being able to infer their types means that annotating can be made optional (e.g., this is convenient when passing short anonymous functions created on the fly, to a module enumerable data, as done by the code in line 28). To study inference, we add *non-annotated* λ-abstractions with pattern-matching to the syntax of Core Elixir:

$$e ::= \ \cdots \mid \lambda(\,\overline{pg \to e}\,)$$

To infer the type of such functions, we use the guard analysis defined in Section 3 to infer a list of accepted types t_i that represent every type potentially accepted by the clause patterns. We then type the body of the function for each of these types, producing t'_i, and take the intersection of the resulting types $\bigwedge_i (t_i \to t'_i)$ as the type of the function. For instance, the analysis of the guard in

$$\lambda(x\ \texttt{when}\ (x\,?\,\texttt{int}\ \texttt{or}\ x\,?\,\texttt{bool}) \to x)$$

produces the two accepted types `int` and `bool`; type-checking the function with `int` as input gives `int` as result, and likewise for `bool`. Hence, the inferred type is $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool})$. Formally, the new expression is typed by the rule (INFER) below

$$(\textsc{infer})\ \frac{\Gamma\,;\mathbb{1} \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i} \qquad \forall i \forall j \quad \Gamma, x : t_{ij} \vdash \mathsf{case}\, x\, \mathsf{do}\,(\,\overline{pg \to e}\,) : t'_{ij}}{\Gamma \vdash \lambda(\,\overline{pg\to e}\,) : \bigwedge_{ij}(t_{ij}\to t'_{ij})}$$

where x is a fresh variable, $\mathbb{1}$ is chosen as the initial type (meaning that the argument could be of any type), and the Boolean flags $\mathfrak{b}_{ij}$ are discarded (the exactness analysis is not required). Note that this typing rule shows how to encode multi-clause definitions into a case expression.

In some cases, this analysis may fail to infer the precise domain of the function (i.e., $\bigvee_{ij} t_{ij}$), in which case, we can imagine the programmer may help the inference process by providing it: in this case, it would suffice to replace this type for $\mathbb{1}$ in the rule (infer). For example if, in the first clause of `test` in line 17, we swap the order of the or-guards, then the type inferred for the function would be the one in lines 19–22 but where the second arrow (line 20) has domain `{:int, ..}` instead of `{:int, term(), ..}`. Although, the type checker would produce a warning (because of the use of (proj_ω)), this type would accept as input `{:int}`, which fails. This can be avoided if the programmer provides the input type `{tuple(), tuple(), ..} or {boolean()}` to the inference process.

Inference in a Dynamic Language. Inferring static function types for existing code in a dynamic language can disrupt continuity, as existing code may rely on invariants that are not captured by types. Furthermore, in a set-theoretic type system, no property guarantees that a given inferred type is the most general; consider, for example, that the successor function could be given types $\texttt{int} \to \texttt{int}$ but also any variation of $(0 \to 1) \wedge (1 \to 2) \wedge ((\texttt{int}\backslash(0\vee 1)) \to \texttt{int})$ using singleton types. While both types are correct and can be related by subtyping, it is the role of the programmers to choose the one that corresponds to their intent and to annotate the function accordingly.

Thus, we need to introduce some flexibility so that inferred static types do not prematurely enforce this choice. We achieve this by adding a dynamic arrow intersection that points the full domain (the union of the t_i's) to ?.

$$(\textsc{infer}_\star)\ \frac{\Gamma\,;\mathbb{1} \vdash (p_i g_i)_{i\le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i\le n, j\le m_i} \qquad \forall i \forall j \quad \Gamma, x : t_{ij} \vdash \mathsf{case}\, x\, \mathsf{do}\,(\,\overline{pg \to e}\,) : t'_{ij}}{\Gamma \vdash \lambda(\,\overline{pg\to e}\,) : \bigwedge_{ij}(t_{ij}\to t'_{ij}) \wedge (\bigvee_{ij} t_{ij} \to ?)}$$

Now '?' gets automatically intersected with each possible return type during function application. While using rule ($\textsc{infer}_\star$) by default appears necessary when typing a dynamic language, being able to type-check using only rule (infer) gives a stronger type safety guarantee, as eliminating the use of '?' during type-checking (and thus, of gradual rules) allows controlling for explicit errors (see Theorems 2.1, 2.2, 2.3).

Multi-arity. *We have presented inference for single-arity functions, but the same principle straightforwardly applies to multi-arity functions presented in Section 4: anonymous functions become* $\lambda(\,\overline{\overline{p}g \to e}\,)$*, and the current guard analysis can be repurposed to produce accepted types for each argument by wrapping these arguments into a tuple pattern.*

6 Implementation

All the features presented in this paper have been implemented in Elixir 1.17 and (forthcoming) 1.18. We used the latter version to assess the overhead introduced by our typechecker on five substantial codebases: Remote is one of the largest Elixir codebases; Credo, Livebook, and Phoenix are among the most popular Elixir packages; and Hex is the package manager for the Elixir ecosystem.

Codebase	LoC	Files	Modules	Type Checking Time	Total Compilation Time
Remote	1,000,000+	10,000+	18,059	11.116 s	707.598 s
Livebook	61093	254	299	0.177 s	4.112 s
Credo	29181	252	264	0.059 s	1.305 s
Phoenix	21389	71	88	0.049 s	0.525 s
Hex	15632	196	241	0.091 s	1.339 s

The “type-checking time” includes type-checking, checking for deprecated APIs, and verifying function definitions. These overheads are comparable to previously existing checks embedded within the Elixir compiler, which have since been replaced by our type system. It is important to stress that type-checking essentially consists of checking subtyping relations and that the complexity of subtype checking for arrow types is not greater than the one for tuple types or for map types (included in 1.18). Thus, the 1.18 implementation of gradual set-theoretic types, with atom singletons, tuple and map types provides a precise assessment of the performances of the monomorphic system presented here, and the results demonstrate that our implementation scales effectively, handling large codebases with minimal performance impact.

The initial implementation of our type system in Elixir 1.17, has being distributed since June 2024 and has already garnered positive feedback from developers, particularly for its ability to uncover previously hidden errors without introducing significant overhead. Notably, it has identified concealed bugs in widely-used open-source libraries such as Phoenix and Livebook, some of which had persisted undetected in production code for over two years. For instance, a bug in the Phoenix framework, used by over 14,600 public websites, was discovered and fixed [23].

The implementation in the Elixir compiler is directly based on the system presented here. There is just a slight gap between the guards of Core Elixir and the one used in Elixir. To fill this gap we defined a more concrete Elixir syntax, we dubbed Featherweight Elixir, together with its translation into Core Elixir. For space reasons, their presentation is deferred to Appendix A.

7 Related work

Two works closely align with ours by using semantic subtyping to establish a type system for Erlang and Elixir (the latter being a compatible superset of the former with which it shares a common functional core). The work most akin to ours is by Castagna et al. [9] focusing on the design principles of incorporating semantic subtyping to Elixir, but omitting all the technical specifics. Our work complements [9], since we develop and formalize the type system and all the technical details that make their design possible. The other relevant work is by Schimpf et al. [35] who propose a type system for Erlang based on semantic subtyping, implement it, and provide useful benchmarks regarding its expressiveness compared both to Dialyzer [31] and Gradualizer [37]. The work by Schimpf et al. [35] is rather different from ours, since they adapt the existing theory of semantic subtyping to Erlang, while the point of our work is to show how to *extend* semantic subtyping with features specific to Elixir: how to add gradual typing without modifying Elixir’s compilation and how to extract the most information from the expressive guards of Erlang/Elixir. Notably, both Castagna et al. [9] and Schimpf et al. [35] provide extensive comparison of the semantic subtyping approach with existing typing efforts for Erlang and Elixir, which we defer to their analyses.

Elixir and Erlang are among the latest languages to embrace semantic subtyping techniques. Other languages in this category include CDuce [18] which lacks gradual typing and guards, but supply the latter with powerful regular expression patterns; Ballerina [2] which is a domain-specific language for network-aware applications whose emphasis is on the use of read-only and write only types and shares with Elixir the typing of records given by Castagna [7]; Lua*u* [28, 32] Roblox’s gradually typed dialect of Lua, a dynamic scripting language for games with emphasis on performance, with a type system that switches to semantic subtyping when the original syntactic subtyping fails [1]; Julia [3] with a type system that is based on a combination of syntactic and semantic subtyping and sports an advanced type system for modules. Although some of these languages use gradual typing and/or guards, none of them have the same focus on these features as Elixir and, ergo, on the typing techniques we developed in this work. Nevertheless, we believe that some of our work could be transposed to these languages, especially the techniques for safe erasure gradual typing (strong functions and dynamic propagation) and the extension of semantic subtyping to multi-arity function spaces.

The thesis by Lanvin [29] defines a semantic subtyping approach to gradual typing, which forms the basis of the gradual typing aspects of our system, since we borrow from Lanvin [29] the definitions of subtyping, precision, and consistent subtyping for gradual types. The main difference with [29] is that he considers that sound gradual typing is achievable by inserting casts in the compiled code whenever necessary, while our work shows a way to adapt gradual typing to achieve soundness while remaining in a full erasure discipline. The relations defined by [29] are also implemented at Meta for the gradual typing of the (Erlang) code of WhatsApp [22], and whose differences with the semantic subtyping approach are detailed in [9], to which we defer this discussion. Lanvin [29] builds on and extends the work by [11] who show how to perform ML-like type reconstruction in a gradual setting with set-theoretic types. As anticipated in Section 1.2, this is one of the limitations of our work. To address it we count on adapting the results of [12] on type inference for dynamic languages. An alternative option is to utilize the approach by Castagna et al. [16] which employs traditional, less computationally demanding type reconstruction techniques than those in [12], but lacks the capability to infer intersection types for functions.

The erasure discipline is a design choice that is popular in industry (as per [27]). For instance, in TypeScript [4], the types leave no trace in the JavaScript emitted by the compiler. But Typescript forgoes soundness, and it requires alterations to the compiler (by addition of static checks [34]) in order to recover it. This issue is shared with Flow [19] and others [30, 33]. Our improvement on this status-quo is to introduce and promote an approach that maintains soundness, in a full erasure context, but recovers as much static information as possible by type-checking functions with strong types that can filter out the dynamic type.

The system we present controls dynamic types via proxies that exist at the type level: strong functions. Thus, we can state that functions with a strong type will work *in the wild* (i.e., when called with dynamic code) and still give a static type (or error on an explicit type test). This property can be related to the notion of *open-world soundness* developed in [40] which states that if a program is well-typed and translated from a gradually-typed surface language into an untyped target, it may interoperate with arbitrary untyped code without producing uncaught type errors. In a sense, our type system can be seen as a proof that Elixir follows already the open-world soundness property when endowed with a gradual type system.

8 Conclusion

This work establishes the theoretical foundation for the Elixir type system, by extending the existing theory of semantic subtyping with key features to capture Elixir programming patterns: safe-erasure gradual typing, multi-arity functions, guard analysis. The resulting type system is expressive enough to capture idiomatic Elixir code, and provide relevant type information to the developer, via warnings and error messages. It is progressively being integrated in Elixir since release 1.17 [21] and, for the time being, has met with a positive reception from the Elixir developer community. This type information can be used to improve the quality of the code and to provide better tooling support. Although the aspects we developed are tailored to Elixir, the theoretical foundation we established can be used to extend the theory of semantic subtyping to other languages, notably dynamic ones, and to provide a more general framework for the design of gradual type systems therein. Our next steps, already underway, are to implement the missing parts of the type system in the Elixir compiler according to the roadmaps sketched by [9, 20], and to evaluate its performance and usability in real-world scenarios. From a theoretical standpoint we aim to extend the type system to include Elixir's first-class module system and to devise types to support concurrency and distribution.

References

[1] 2023. Lua*u* .594 Release. GitHub release. https://github.com/luau-lang/luau/releases/tag/0.594 Released on September the 8th 2023.

[2] Ballerina. [n. d.]. Ballerina. https://ballerina.io/ Accessed on Feb 28, 2024.

[3] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 120 (oct 2018), 23 pages. https://doi.org/10.1145/3276490

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

[5] Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. 2020. A gradual type system for Elixir. In *Proceedings of the 24th Brazilian Symposium on Context-oriented Programming and Advanced Modularity*. Association for Computing Machinery, New York, NY, USA, 17–24.

[6] 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* Volume 16, Issue 1 (Feb. 2020). https://doi.org/10.23638/LMCS-16(1:15)2020

[7] Giuseppe Castagna. 2023. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP, Article 196 (Sept. 2023). https://doi.org/10.1145/3607838

[8] Giuseppe Castagna. 2024. Programming with union, intersection, and negation types. In *The French School of Programming*, Bertrand Meyer (Ed.). Springer, 309–378. https://doi.org/10.1007/978-3-031-34518-0_12 Preprint at arXiv:2111.03354.

[9] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (2024). https://doi.org/10.22152/programming-journal.org/2024/8/4

[10] Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 198–199. https://doi.org/10.1145/1069774.1069793

[11] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G Siek. 2019. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32. https://doi.org/10.1145/3290329

[12] Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024. Polymorphic Type Inference for Dynamic Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 40 (Jan. 2024). https://doi.org/10.1145/3632882

[13] Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. 2022. On type-cases, union elimination, and occurrence typing. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 75. https://doi.org/10.1145/3498674

[14] 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-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 289–302. https://doi.org/10.1145/2676726.2676991

[15] 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-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 5–17. https://doi.org/10.1145/2676726.2676991

[16] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-Theoretic Types for Polymorphic Variants. In *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming*. 378–391. https://doi.org/10.1145/2951913.2951928

[17] 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

[18] cduce [n. d.]. CDuce. https://www.cduce.org/ Accessed on Feb 28, 2024.

[19] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 48:1–48:30. https://doi.org/10.1145/3133872

[20] Elixir. 2024. *Elixir documentation: Gradual set-theoretic types.* https://hexdocs.pm/elixir/gradual-set-theoretic-types.html

[21] Elixir. 2024. *Elixir v1.17 released: set-theoretic types in patterns, calendar durations, and Erlang/OTP 27 support.* https://elixir-lang.org/blog/2024/06/12/elixir-v1-17-0-released

[22] eqWAlizer [n. d.]. eqWAlizer. https://github.com/WhatsApp/eqwalizer.

[23] Phoenix Framework. 2023. Bug fix commit: "Address bug found by typesystem". Retrieved October 10, 2024 from https://github.com/phoenixframework/phoenix/commit/34d0ffef6aebcb5d4f210978aabca53b0e57f1ae

[24] Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. Ph. D. Dissertation. PhD thesis, Université Paris 7.

[25] 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

[26] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008), 1–64. https://doi.org/10.1145/1391289.1391293

[27] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *ACM Transactions on Programming Languages and Systems* 45, 1 (March 2023), 1–54. https://doi.org/10.1145/3579833

[28] Alan Jeffrey. 2022. Semantic Subtyping in Lua*u*. Blog post. https://blog.roblox.com/2022/11/semantic-subtyping-luau Accessed on May 6th 2023.

[29] Victor Lanvin. 2021. *A semantic foundation for gradual set-theoretic types*. Ph. D. Dissertation. Université Paris Cité.

[30] Jukka Lehtosalo, G v Rossum, Ivan Levkivskyi, Michael J Sullivan, David Fisher, Greg Price, Michael Lee, N Seyfer, R Barton, S Ilinskiy, et al. 2017. Mypy-optional static typing for python. *URL: http://mypy-lang. org/[cited 2021-11-30]* (2017).

[31] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*. Association for Computing Machinery, New York, NY, USA, 167–178.

[32] Luau [n. d.]. Lua*u*. https://luau-lang.org/.

[33] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 481–494. https://doi.org/10.1145/2103656.2103714

[34] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & efficient gradual typing for TypeScript. In *POPL '15*. ACM, 167–180.

[35] Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-theoretic Types for Erlang. In *Proc. of IFL 2022*. ACM, Copenhagen, Denmark, Article 4. https://doi.org/10.1145/3587216.3587220

[36] Erik Stenman. 2024. *The Erlang Runtime System*. Retrieved February 28, 2024 from https://blog.stenmans.org/theBeamBook/

[37] Josef Svenningsson. [n. d.]. Gradualizer. https://github.com/josefs/Gradualizer.

[38] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *POPL '08*. ACM, Association for Computing Machinery, New York, NY, USA, 395–406.

[39] Michael M. Vitousek. [n. d.]. Reticulated Python. https://github.com/mvitousek/reticulated.

[40] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

$$\begin{array}{llll}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda(\overline{pg \to e}) \mid e(\overline{e}) \mid \{\overline{e}\} \mid \pi_e\, e \mid \texttt{case}\ e\ \overline{pg \to e} \mid e + e \\
\textbf{Patterns} & p & ::= & c \mid x \mid \{\overline{p}\} \\
\textbf{Guards} & g & ::= & a\,?\,\tau \mid a = a \mid a \mathrel{!=} a \mid g \ \texttt{and}\ g \mid g\ \texttt{or}\ g \\
\textbf{Guard atoms} & a & ::= & c \mid x \mid \pi_a\, a \mid \texttt{size}\, a \mid \{\overline{a}\} \\
\textbf{Test types} & \tau & ::= & b \mid c \mid \texttt{function}_n \mid \{\overline{\tau}\} \mid \{\overline{\tau}\} \mid \tau \vee \tau \mid \neg\tau \\
\textbf{Base types} & b & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{function} \mid \texttt{tuple} \\
\textbf{Types} & t & ::= & b \mid c \mid \overline{t} \to t \mid \{\overline{t}\} \mid \{\overline{t}, ..\} \mid t \vee t \mid \neg\, t \mid ?
\end{array}$$

Fig. 9. Core Elixir

$$\begin{array}{llll}
\textbf{Expressions} & \mathsf{E} & ::= & \mathsf{L} \mid x \mid \texttt{fn}\ \overline{\mathsf{P}\ \texttt{when}\ \mathsf{G} \texttt{->} \mathsf{E}}\ \texttt{end} \mid \mathsf{E}\,(\mathsf{E}_1, .., \mathsf{E}_n) \mid \mathsf{E} + \mathsf{E} \\
 & & \mid & \texttt{case}\ \mathsf{E}\ \ \overline{\mathsf{P}\ \texttt{when}\ \mathsf{G}\ \texttt{->}\ \mathsf{E}} \mid \{\, \mathsf{E}_1, .., \mathsf{E}_n\,\} \mid \texttt{elem}(\mathsf{E}, \mathsf{E}) \\
\textbf{Singletons} & \mathsf{L} & ::= & n \mid k \mid \{\overline{\mathsf{L}}\} \\
\textbf{Patterns} & \mathsf{P} & ::= & \mathsf{L} \mid x \mid \{\mathsf{P}_1, .., \mathsf{P}_n\} \\
\textbf{Guards} & \mathsf{G} & ::= & \mathsf{D} \mid \mathsf{C} \mid \texttt{not}\ \mathsf{G} \mid \mathsf{G}\ \texttt{and}\ \mathsf{G} \mid \mathsf{G}\ \texttt{or}\ \mathsf{G} \mid \mathsf{G} == \mathsf{G} \mid \mathsf{G} \mathrel{!=} \mathsf{G} \\
\textbf{Selectors} & \mathsf{D} & ::= & \mathsf{L} \mid x \mid \texttt{elem}(\mathsf{D}, \mathsf{D}) \mid \texttt{tuple\_size}(\mathsf{D}) \mid \{\overline{\mathsf{D}}\} \\
\textbf{Checks} & \mathsf{C} & ::= & \texttt{is\_integer}(\mathsf{D}) \mid \texttt{is\_atom}(\mathsf{D}) \mid \texttt{is\_tuple}(\mathsf{D}) \\
 & & \mid & \texttt{is\_function}(\mathsf{D}) \mid \texttt{is\_function}(\mathsf{D}, n)
\end{array}$$

(where x ranges over variables, n ranges over integers, and k ranges over atoms)

Fig. 10. Featherweight Elixir

A Language Formalization

Throughout this paper, we have used Core Elixir as the language for technical discussions. This appendix provides a comprehensive overview of Core Elixir's syntax and its relationship to Featherweight Elixir, a subset of the full Elixir language.

Figure 9 presents the full syntax of Core Elixir, which encompasses all the language features discussed in the main body of the paper. This includes expressions, patterns, guards, guard atoms, test types, base types, and types.

To bridge the gap between Core Elixir and a more concrete Elixir syntax, we introduce Featherweight Elixir (FW-Elixir). FW-Elixir is a strict subset of Elixir that covers all the language features discussed in this paper, including tuples, anonymous and multi-arity functions, case-expressions with patterns and guards, and more. Figure 10 presents the formal syntax for FW-Elixir.

It's important to note that all the examples provided in the main text of the paper are valid syntax for both Elixir and FW-Elixir. FW-Elixir actually extends beyond the examples shown in the main text, as it allows for negated guards, which are absent from both the previous examples and Core Elixir. For instance, consider the following alternative definition of the `negate` function:

```elixir
def negate(x) when not(is_function(x) or is_tuple(x)), do: not x
```

This definition is equivalent to the one presented earlier in the paper, assuming `integer()` and `boolean()` are the only basic types used in FW-Elixir. The type `boolean()` is the complement of `integer() or function() or tuple()` (where the type `function()` denotes the type of all functions) and all values of type `integer()` are captured by the first clause of `negate`.

To illustrate how Core Elixir relates to the examples discussed in the main text, consider the function `second_strong` from the paper. It can be encoded in Core Elixir as:

$$\lambda^{\{\{\mathbb{1},\texttt{int}\}\to\texttt{int}\}}(x).\, \texttt{case}\, x\ (\,\{\mathbb{1}, \texttt{int}\} \to \pi_1\, x)$$

$$
\begin{aligned}
\mathbf{T}_{\mathsf{G}}(\mathtt{D}) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{true}\\
\mathbf{T}_{\mathsf{G}}(C) &= \mathbf{T}_{\mathsf{C}}(C)\\
\mathbf{T}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{and}\ \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{and}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{or}\ \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{or}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathtt{not\,G}) &= \mathbf{N}_{\mathsf{G}}(\mathtt{G})\\
\mathbf{T}_{\mathsf{G}}(\mathtt{G}_1 \mathtt{==} \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{=}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{T}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{!=}\ \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{!=}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{T}_{\mathsf{D}}(\mathtt{elem}(\mathtt{D}_1,\mathtt{D}_2)) &= \pi_{\mathbf{T}_{\mathsf{D}}(\mathtt{D}_1)}\,\mathbf{T}_{\mathsf{D}}(\mathtt{D}_2)\\
\mathbf{T}_{\mathsf{D}}(\mathtt{tuple\_size}(\mathtt{D})) &= \mathtt{size}\ \mathbf{T}_{\mathsf{D}}(\mathtt{D})\\
\mathbf{T}_{\mathsf{D}}(\{\mathtt{D}_1,..,\mathtt{D}_n\}) &= \{\mathbf{T}_{\mathsf{D}}(\mathtt{D}_1),..,\mathbf{T}_{\mathsf{D}}(\mathtt{D}_n)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_integer}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{int}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_atom}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{atom}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_tuple}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{tuple}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_function}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{function}\\
\mathbf{T}_{\mathsf{C}}(\mathtt{is\_function}(\mathtt{D},n)) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{function}_n
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{N}_{\mathsf{G}}(\mathtt{D}) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,\mathtt{false}\\
\mathbf{N}_{\mathsf{G}}(\mathtt{C}) &= \mathbf{N}_{\mathsf{C}}(\mathtt{C})\\
\mathbf{N}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{and}\ \mathtt{G}_2) &= \mathbf{N}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{or}\ \mathbf{N}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{or}\ \mathtt{G}_2) &= \mathbf{N}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{and}\ \mathbf{N}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathtt{not\,G}) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G})\\
\mathbf{N}_{\mathsf{G}}(\mathtt{G}_1 \mathtt{==} \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{!=}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)\\
\mathbf{N}_{\mathsf{G}}(\mathtt{G}_1\ \mathtt{!=}\ \mathtt{G}_2) &= \mathbf{T}_{\mathsf{G}}(\mathtt{G}_1)\ \mathtt{=}\ \mathbf{T}_{\mathsf{G}}(\mathtt{G}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_integer}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,(\neg\mathtt{int})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_atom}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,(\neg\mathtt{atom})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_tuple}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,(\neg\mathtt{tuple})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_function}(\mathtt{D})) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,(\neg\mathtt{function})\\
\mathbf{N}_{\mathsf{C}}(\mathtt{is\_function}(\mathtt{D},n)) &= \mathbf{T}_{\mathsf{D}}(\mathtt{D})\,?\,(\neg\mathtt{function}_n)
\end{aligned}
$$

Fig. 11. Guard Compilation

where $\mathbb{1}$ denotes the top type. Similarly, the second clause of the `test` function can be expressed in Core Elixir as a branch of a case-expression with pattern x and guard:

$$(x\,?\,\mathtt{bool})\ \mathtt{or}\ (\pi_0\,x = \pi_1\,x)$$

An important point is the link between our analysis of guards, and the assumptions behind it. In our study, we only considered guards with disjunctions and conjunctions, because we had a technique to eliminate negations in the first place. This technique relies on a compilation step for guards, that we now present.

In Figure 11 we define two mutually recursive functions from the set $\mathcal{G}_{\text{Elixir}}$ of Elixir concrete guards of FW-Elixir to the set $\mathcal{G}_{\text{Core}}$ of Core Elixir guards (syntax in Figure 6). Precisely, the $\mathbf{T} : \mathcal{G}_{\text{Elixir}} \longrightarrow \mathcal{G}_{\text{Core}}$ function compiles a concrete guard into a core guard, and the $\mathbf{N} : \mathcal{G}_{\text{Elixir}} \longrightarrow \mathcal{G}_{\text{Core}}$ does so as well, but also pushes down a logical negation into the guard, which means that, say, a type-check of `int` becomes a type-check of $\neg$ `int`, and that conjunctions and disjunctions are swapped using De Morgan rules. There is no **N** defined on selectors `D`: **N** is just an auxiliary function for **T**, which does not call it on `D` productions (a selector `D` is directly translated into checking whether it has the singleton type `true`).

A.1 Operational Semantics

The language has strict reduction semantics defined by the reduction rules in the Figures from 12 to 15. The semantics is defined in terms of values (ranged over by v), evaluation contexts (ranged over by $\mathcal{E}$), and guard evaluation contexts (ranged over by $\mathcal{G}$), the latter used to define the semantics of pattern matching. They are defined as follows:

$$
\begin{array}{llll}
\textbf{Values} & v & ::= & c \mid \lambda^{\mathbb{I}}\overline{x}.e \mid \{\overline{v}\}\\
\textbf{Context} & \mathcal{E} & ::= & \square \mid \mathcal{E}(e) \mid v(\mathcal{E}) \mid \{\overline{v},\mathcal{E},\overline{e}\} \mid \pi_{\mathcal{E}}\,e \mid \pi_v\,\mathcal{E} \mid \mathtt{case}\ \mathcal{E}\,(\tau_i \to e_i)_{i\in I}\\
 & & & \mid \mathcal{E} + e \mid v + \mathcal{E} \mid \mathtt{size}\ \ \mathcal{E} \mid \mathtt{case}\ \mathcal{E}\ \overline{pg \to e}\\
\textbf{Guard Context} & \mathcal{G} & ::= & \square \mid \mathcal{G}\ \mathtt{and}\ g \mid \mathcal{G}\ \mathtt{or}\ g \mid \mathcal{G}\ ?\ t \mid \mathcal{G} = a \mid v = \mathcal{G} \mid \mathcal{G}\ \mathtt{!=}\ a \mid v\ \mathtt{!=}\ \mathcal{G}
\end{array}
$$

Since patterns contain capture variables, the reduction of pattern matching implies the creation of a substitution σ that binds the capture variables of the pattern to the values they capture.

Finally, the semantics of pattern matching includes the evaluation of guards. A given branch succeeds iff the value matches a pattern, and the guard evaluates to true. Note that a guard can fail,

$$
\begin{array}{llcll}
[\textsc{App}] & (\lambda^{\mathbb{I}} x.e)\, v & \hookrightarrow & e[v/x] & \\
[\textsc{Proj}] & \pi_i\,\{v_0, .., v_n\} & \hookrightarrow & v_i & \text{if } i \in [0\,..\,n] \\
[\textsc{Plus}] & v + v' & \hookrightarrow & v'' & \text{where } v'' = v + v' \\
 & & & & \text{and } v, v' \text{ are integers} \\
[\textsc{Size}] & \texttt{size}\ \{v_1, .., v_n\} & \hookrightarrow & n & \\
[\textsc{Match}] & \texttt{case}\, v\ \texttt{do}\ \ (p_i g_i \to e_i)_{i<n} & \hookrightarrow & e_j\,\sigma & \text{if } v/(p_j g_j) = \sigma \text{ and} \\
 & & & & \forall (i<j<n).\ v/(p_i g_i) = \texttt{fail} \\
[\textsc{Context}] & \mathcal{E}[e] & \hookrightarrow & \mathcal{E}[e'] & \text{if } e \hookrightarrow e' \\
 & & & & \\
[\textsc{App}_\omega] & v(v') & \hookrightarrow & \omega_{\textsc{BadFunction}} & \text{if } v \neq \lambda^{\mathbb{I}} x.e \\
[\textsc{Proj}_{\omega,\textsc{bound}}] & \pi_v\,\{v_0, .., v_n\} & \hookrightarrow & \omega_{\textsc{OutOfRange}} & \text{if } v \neq i \text{ for } i = 0\,..\,n \\
[\textsc{Proj}_{\omega,\textsc{nonTuple}}] & \pi_{v'}\, v & \hookrightarrow & \omega_{\textsc{NotTuple}} & \text{if } v \neq \{\overline{v}\} \\
[\textsc{Plus}_\omega] & v + v' & \hookrightarrow & \omega_{\textsc{ArithError}} & \text{if } v \text{ or } v' \text{ not integers} \\
[\textsc{Size}_\omega] & \texttt{size}\ \ v & \hookrightarrow & \omega_{\textsc{Size}} & \text{if } v \neq \{\overline{v}\} \\
[\textsc{Match}_\omega] & \texttt{case}\, v\ \texttt{do}\ \ (p_i g_i \to e_i)_{i<n} & \hookrightarrow & \omega_{\textsc{CaseEscape}} & \text{if } v/p_i g_i = \texttt{fail} \text{ for all } i < n \\
[\textsc{Context}_\omega] & \mathcal{E}[e] & \hookrightarrow & \omega_p & \text{if } e \hookrightarrow \omega_p
\end{array}
$$

Fig. 12. Standard and Failure Reductions

$$
\begin{array}{lll}
v/c & = \{\} & \text{if } v = c \\
v/x & = \{x \mapsto v\} & \\
v/(p_1 \& p_2) & = \sigma_1 \cup \sigma_2 & \text{if } v/p_1 = \sigma_1 \text{ and } v/p_2 = \sigma_2 \\
\{v_1, \ldots, v_n\}/\{p_1, \ldots, p_n\} & = \bigcup_{i=1..n} \sigma_i & \text{if } v_i/p_i = \sigma_i \text{ for all } i = 1..n \\
v/p & = \texttt{fail} & \text{otherwise} \\
 & & \\
v/(pg) & = \sigma & \text{if } v/p = \sigma \text{ and } g\,\sigma \hookrightarrow^* \text{true} \\
v/(pg) & = \texttt{fail} & \text{otherwise}
\end{array}
$$

where σ denotes substitutions from variables to values

Fig. 13. Definition of $\boldsymbol{v/p}$ and $\boldsymbol{v/(pg)}$

$$
\begin{array}{rrcll}
[\textsc{And}_\top] & \texttt{true and}\ g & \hookrightarrow & g & \\
[\textsc{And}_\bot] & v\ \texttt{and}\ g & \hookrightarrow & \texttt{false} & \text{if } v \neq \texttt{true} \\
[\textsc{Or}_\top] & \texttt{true or}\ g & \hookrightarrow & \texttt{true} & \\
[\textsc{Or}_\bot] & \texttt{false or}\ g & \hookrightarrow & g & \\
[\textsc{Eq}_\top] & v\ =\ v' & \hookrightarrow & \texttt{true} & \text{if } \ v = v' \\
[\textsc{Eq}_\bot] & v\ =\ v' & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{NEq}_\top] & v\ \ \texttt{!=}\ v' & \hookrightarrow & \texttt{true} & \text{if } \ v \neq v' \\
[\textsc{NEq}_\bot] & v\ \ \texttt{!=}\ v' & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{OfType}_\top] & v\,?\,t & \hookrightarrow & \texttt{true} & \text{if } \ v \in t \\
[\textsc{OfType}_\bot] & v\,?\,t & \hookrightarrow & \texttt{false} & \text{else} \\
[\textsc{Ctx}] & \mathcal{G}[g] & \hookrightarrow & \mathcal{G}[g'] & \text{if } g \hookrightarrow g' \\
[\textsc{CtxAtom}] & \mathcal{G}[a] & \hookrightarrow & \mathcal{G}[a'] & \text{if } a \hookrightarrow a' \\
[\textsc{Context}] & \mathcal{G}[a] & \hookrightarrow & \texttt{false} & \text{if } a \hookrightarrow \omega
\end{array}
$$

Fig. 14. Guard Reductions.

in which case the branch is skipped (it does not constitute a failure of the whole pattern matching): this is stated by the rule Context in Figure 14

$$\begin{array}{rrcll}
[\textsc{Size}] & \texttt{size}\ (\{v_1, .., v_n\}) & \hookrightarrow & n & \text{where } n \in \mathbb{N} \\
[\textsc{Size}_\omega] & \texttt{size}\ (v) & \hookrightarrow & \omega_{\textsc{Size}} & \text{if } v \neq \{\overline{v}\} \\
[\textsc{Proj}] & \pi_i\, \{v_1, .., v_n\} & \hookrightarrow & v_i & \text{if } i \in \{1, .., n\} \\
[\textsc{Proj}_{\omega,\textsc{bound}}] & \pi_v\, \{v_1, .., v_n\} & \hookrightarrow & \omega_{\textsc{OutOfRange}} & \text{if } v \notin \{1, .., n\} \\
[\textsc{Proj}_{\omega,\textsc{nonTuple}}] & \pi_{v'}\, v & \hookrightarrow & \omega_{\textsc{NotTuple}} & \text{if } v \neq \{\overline{v}\} \\
[\textsc{Context}] & \mathcal{E}[a] & \hookrightarrow & \mathcal{E}[a'] & \text{if } a \hookrightarrow a'
\end{array}$$

Fig. 15. Guard Atom Reductions.

B Soundness for Section 2

Definition B.1. The terms constituting the source language of Section 2 are defined by the following grammar:

$$\begin{array}{ll}
\textbf{Terms} & e ::= x \mid c \mid \lambda^{\mathbb{I}}x.e \mid e\,e \mid \texttt{case}\ e\ (\tau_i \to e_i)_{i\in I} \\
\textbf{Values} & v ::= c \mid \lambda^{\mathbb{I}}x.e \\
\textbf{Interfaces} & \mathbb{I} ::= \{t_i \to s_i \mid i \in I\}
\end{array}$$

In this section we consider, without loss of generality only interfaces $\mathbb{I} = \{t_i \to s_i \mid i \in I\}$ that satisfy the conditions $\forall(i, j) \in I^2, (t_i \wedge t_j)^{\Uparrow} \leq \mathbb{O}$, and $\forall i \in I, t_i^{\Uparrow} \not\leq \mathbb{O}$. In words, the domains of the arrows in the interface must always be pairwise disjoint, meaning that they do not overlap. While this restriction might seem limiting, any arbitrary interface can be statically converted into a valid one that adheres to this rule, though this conversion process may lead to a considerable increase in the size of the interface. Consider, for instance, the following interface that does not satisfy this restriction: $\{\texttt{int} \to \texttt{int}; 5 \to 5; ? \to ?\}$. Through static transformation, we can derive an (intuitively) equivalent, valid interface:

$$\{(\texttt{int} \setminus 5) \to \texttt{int}; (\texttt{int} \wedge 5) \to (\texttt{int} \wedge 5); (5 \setminus \texttt{int}) \to 5; (? \setminus (\texttt{int} \wedge 5)) \to ?\}$$

which simplifies to:

$$\{(\texttt{int} \setminus 5) \to \texttt{int}; 5 \to 5; (? \setminus (\texttt{int} \wedge 5)) \to ?\}$$

This technique extends to interfaces containing multiple overlapping gradual types. As an illustration, the interface $\{? \to \texttt{int}; ? \to 5\}$ can be simplified to $\{? \to \texttt{int} \vee 5\}$.

Such transformations enhance type safety while preserving the original interface's semantic intent, albeit at the cost of increased complexity in some cases.

To simplify the typing rule for pattern matching (and the associated proof of soundness), we assume that the restriction also applies to gradual domains, that is, $\forall(i, j) \in I^2, i \neq j \Longrightarrow \tau_i \wedge \tau_j \leq \mathbb{O}$. This definition is not restrictive either, as any non-disjoint case expression can be compiled into a disjoint one by subtracting the union of the previous cases from the current one for each branch.

The typing rules for Core Elixir are given in a declarative style, and grouped into two figures: Figure 16 shows the gradual type system that is used to typecheck programs, Figure 17 shows the strong system which is used as an auxiliary stystem in the inference of strong function types. We prove subject reduction for this latter system.

When type-checking gradually typed programs, rule $(\lambda_\star^{\mathbb{1}})$ from Figure 16 is used instead of rule (λ) given in Figure 3. Once again this modification does not affect the set of well-type terms since the $(\lambda_\star^{\mathbb{1}})$ is admissible for the system of Figure 3, as its extra premise $\Gamma, x : ? \vdash e : \mathbb{1}$ is verified when no ill-typed expression can hide in an unreachable branch of a case expression, which is guaranteed by the condition of Remark 1. In other terms, under the hypothesis of Remark 1, rules $(\lambda_\star^{\mathbb{1}})$ and (λ) are equivalent.

Type-checking a program using only the static rules of Figure 16, which are those not annotated with any subscript $\star$ or ?, gives the strongest safety guarantee as it prevents all runtime errors

$$\text{(cst)}\ \frac{}{c : c\ \wedge ?} \qquad \text{(var)}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad (\lambda)\ \frac{\forall (t_i \to s_i) \in \mathbb{I} \quad (\Gamma, x : t_i \vdash e : s_i)}{\Gamma \vdash \lambda^{\mathbb{I}} x\,.\,e : \bigwedge_{i \in I}(t_i \to s_i)}$$

$$(\lambda_\star^{\mathbb{1}})\ \frac{\forall (t_i \to s_i) \in \mathbb{I} \quad (\Gamma, x : t_i \vdash e : s_i) \qquad \Gamma, x : ? \vdash e \mathrel{\substack{\circ\\ \circ}} \mathbb{1}}{\Gamma \vdash \lambda^{\mathbb{I}} x\,.\,e : \bigwedge_{i \in I}(t_i \to s_i)}$$

$$(\lambda_\star)\ \frac{\Gamma \vdash_\circ \lambda^{\mathbb{I}} x.e : t_1 \to t_2 \qquad \Gamma, x : ? \vdash e \mathrel{\substack{\circ\\ \circ}} t_2 \wedge ?}{\Gamma \vdash_\circ \lambda^{\mathbb{I}} x.e : (t_1 \to t_2)^\star}$$

$$\text{(app)}\ \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{e_1(e_2) : t_2} \qquad (\text{app}_\star)\ \frac{e_1 : (t_1 \to t)^\star \quad e_2 : t_2}{e_1(e_2) : t \wedge ?}\,(t_2 \lesssim t_1)$$

$$(\text{app}_?)\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1(e_2) :\ ?}\,(\exists t.(t_1 \lesssim t \to \mathbb{1}) \text{ and } (t_2 \lesssim t))$$

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad \forall i \in I.\ (t \wedge t_i \not\leq \mathbb{0} \implies \Gamma \vdash e_i : t')}{\Gamma \vdash \mathsf{case}\, e\, (\tau_i \to e_i)_i : t'}\,(t \leq \vee_i \tau_i)$$

$$(\text{case}_?)\ \frac{\Gamma \vdash e : t \qquad \forall i \in I.\ (t \wedge t_i \not\leq \mathbb{0} \implies \Gamma \vdash e_i : t')}{\Gamma \vdash \mathsf{case}\, e\, (\tau_i \to e_i)_i : t' \wedge ?}\,(t \lesssim \vee_i t_i)$$

$$\text{(plus)}\ \frac{e_1 : \mathtt{int} \quad e_2 : \mathtt{int}}{e_1 + e_2 : \mathtt{int}} \qquad (\text{plus}_?)\ \frac{e_1 : t_1 \quad e_2 : t_2}{e_1 + e_2 : \mathtt{int} \wedge ?} \begin{cases} t_1 \lesssim \mathtt{int} \\ t_2 \lesssim \mathtt{int} \end{cases} \qquad \text{(sub)}\ \frac{e : t_1 \quad t_1 \leq t_2}{e : t_2}$$

Fig. 16. Typing rules for the gradual system

defined in the operational semantics. For extra clarity, any judgement of a proof that uses only this kind of rules is denoted by $\Gamma \vdash_{\text{static}} e : t$.

Type-checking with the full gradual system ensures that a well-typed program evaluates to a value of the expected type, but admits various runtime errors.

After that, we show how to extend the type system to handle tuples and projections, and show that different static rules can be used to provide different levels of safety (see Figure 18 and Theorem B.17).

B.1 Static safety

Lemma B.2 (Permutation)**.** For every expression e, types t, t_1, t_2, environment Γ, and variables $x, y \notin \mathsf{dom}(\Gamma)$,

$$(\Gamma, x : t_1, y : t_2 \vdash e : t) \Rightarrow (\Gamma, y : t_2, x : t_1 \vdash e : t)$$

Proof of Lemma B.2. By induction on the size of the derivation tree and case analysis on the last typing rule used to derive $\Gamma, x : t_1, y : t_2 \vdash e : t$. Every non-base case is handled by directly applying the induction hypothesis to the premises. □

Lemma B.3 (Weakening)**.** For every expression e, types t, s, environment Γ, and variable $x \notin \mathsf{dom}(\Gamma)$,

$$x \notin \mathsf{fv}(e) \wedge (\Gamma, x : s \vdash e : t) \Rightarrow (\Gamma \vdash e : t)$$

Proof of Lemma B.3. By induction on the size of the derivation tree and case analysis on the last typing rule used to derive $\Gamma, x : s \vdash e : t$.

(var): $e = y$ and $y \neq x$ by assumption. This implies $(y : t) \in \Gamma$, hence $\Gamma \vdash e : t$ by rule (var).

(cst): Immediate since e is a constant so its typing does not depend on the environment.

(λ): $e = \lambda^{\mathbb{I}} y.e'$. By inversion of the typing rule, we have $\Gamma, x : s, y : t_i \vdash e' : s_i$ for all $i \in I$. Rearranging the environment by Permutation B.2, and by induction hypothesis, we deduce that $\Gamma, y : t_i \vdash e' : s_i$ for all $i \in I$. Therefore, $\Gamma \vdash \lambda^{\mathbb{I}} y.e' : t' \to t$.

(tuple), (app), (case), (plus), ($\leq$): These rules maintain the same environment in the conclusion and premises, and involve sub-expressions in the premises.

□

Lemma B.4 (Static Substitution). For all expressions e, e_1, types t, t_1 and variable $x \notin \mathtt{dom}(\Gamma)$,

$$(\Gamma, x : t_1 \vdash_{\text{static}} e : t) \wedge (\Gamma \vdash_{\text{static}} e_1 : t_1) \Longrightarrow (\Gamma \vdash_{\text{static}} e[e_1/x] : t)$$

Proof of Lemma B.4. By induction on the size of the derivation tree and case analysis on the last typing rule used to derive $\Gamma, x : t_1 \vdash e : t$.

(cst). Immediate since e is a constant and does not depend on x.

(var). $e = y$. There are two cases:

- $y = x$. Then $e[e_1/x] = e_1$ so by assumption $\Gamma, x : t_1 \vdash x : t$ and $x \notin \mathtt{dom}(\Gamma)$. By inversion of rule ($\leq$), we have $t_1 \leq t$. Applying rule ($\leq$) to $\Gamma \vdash e_1 : t_1$ concludes.
- $y \neq x$. Then $e[e_1/x] = y$, and the result follows since $\Gamma, x : t_1 \vdash y : t$.

(λ). $e = \lambda^{\mathbb{I}} y.e_1$. By inversion, $\Gamma, x : t_1, y : t_i \vdash e_i : s_i$ for all $(t_i \to s_i) \in \mathbb{I}$. Rearranging the variables by Permutation B.2, and by induction hypothesis, $\Gamma, y : t_i \vdash e_i[e_1/x] : s_i$ for all $i \in I$. This concludes by re-applying the (λ) typing rule.

(tuple), (app), (case), (proj), (proj$_\omega$), (proj$^{\mathbb{1}}_\omega$), (+), ($\leq$). maintain the same environment in the conclusion and premises, and involve sub-expressions in the premises. Hence, they are handled in the same way as the (λ) rule by directly applying the induction hypothesis to their premises.

□

Lemma B.5 (Static Progress). If $\varnothing \vdash_{\text{static}} e : t$, then either:

- $\exists v$ s.t. $e = v$;
- $\exists e'$ s.t. $e \hookrightarrow e'$;

Proof. Our set of reduction rules (see Figure 2), including failure reductions, is complete. This means that every expression that is not a variable—thus, *a fortiori*, every closed expression—is either a value, or it can be reduced to another expression (which will be closed, too) or to a failure $\omega \in \{\omega_{\text{CaseEscape}}, \omega_{\text{OutOfRange}}, \omega_{\text{NotTuple}}, \omega_{\text{BadFunction}}, \omega_{\text{ArithError}}\}$.

We will prove that for a well-typed expression in $\vdash_{\text{static}}$, the failure cases are impossible. Let's assume there exists an expression e such that $e \hookrightarrow \omega_p$, where p is one of the failure cases. We'll analyze each case:

1. Case p = CaseEscape: In this case, $e = \mathtt{case}\ v\ (\tau_i \to e_i)_{i \in I}$ where $v \notin \bigvee_{i \in I} \tau_i$. However, by inverting the (case) typing rule used for e, we have $\varnothing \vdash v : t'$ where $t' \leq \bigvee_{i \in I} \tau_i$. This contradicts our assumption, as v must belong to $\bigvee_{i \in I} \tau_i$.
2. Case p = NotTuple: Here, $e = \pi_{v'}\ v$ where v is not a tuple. The rules that introduce projections imply, by inversion, that either $v : \{t_0, .., t_n, ..\}$ or $v : \mathtt{tuple}$. In both cases, v must be a tuple, contradicting our assumption.
3. Case p = BadFunction: In this scenario, $e = v(v')$ where v is not a lambda-abstraction. By inverting the (app) typing rule, we have $v : t_1 \to t_2$ and $v' : t_1$. This contradicts our assumption, as v must be a lambda-abstraction.

(4) Case $p = \textsc{NonIntPlus}$: Here, $e = v_1 + v_2$ where either v_1 or v_2 is not an integer. Inverting the (+) typing rule gives us $v_1 : \texttt{int}$ and $v_2 : \texttt{int}$. This contradicts our assumption, as both v_1 and v_2 must be integers.

It's worth noting that $p = \textsc{OutOfRange}$ is not prevented by the typing rules (proj$_\omega$) and (proj$^{\mathbb{1}}_\omega$), which allow expressions like $\pi_3 \{1, 2\}$ to be typed as $1 \vee 2$ or $\mathbb{1}$. While these rules seem necessary to avoid burdening programmers with statically proving index bounds, a practical implementation should include a rule that raises a type error for $\pi_e\, e'$ when $e' : \{t_0, .., t_n\}$ and $e : \neg[0..n]$. □

Lemma B.6 (Static Preservation). If $\varnothing \vdash_{\text{static}} e : t$ and $e \hookrightarrow e'$, then $\varnothing \vdash_{\text{static}} e' : t$

Proof. By induction on the size of the derivation tree and case analysis on the typing rule used to derive $\varnothing \vdash e : t$. The reduction hypothesis excludes rules (cst), (var) and (λ). In every case, if $e \hookrightarrow e'$ is a context reduction, then we apply the induction hypothesis to its premises and conclude by re-applying the typing rule. Thus, we only explicitly treat rules for which there is a distinct reduction:

(app). $e = e_1(e_2)$. By inversion of the typing rule, we have $\varnothing \vdash e_1 : t_1 \to t_2$ and $\varnothing \vdash e_2 : t_1$. Since the reduction is β-reduction, we have $e_1 = \lambda^{\mathbb{I}} x.e_1'$ and $e' = e_1'[e_2/x]$. By Substitution B.4, we deduce that $\varnothing \vdash e_1'[e_2/x] : t_2$.

(case). $e = \texttt{case}\ e'\ (\tau_i \to e_i)_{i \in I}$. By inversion of the typing rule, we have $\varnothing \vdash e' : t'$ and $\forall i \in I\ \big((t' \wedge \tau_i) \smallsetminus (\bigvee_{j<i}\ \tau_j) \not\leq \mathbb{0} \Rightarrow e_i : t\big)$. Due to the (case) reduction, e' is a value of type t'. The exhaustiveness condition on the case typing rule tells us that $t' \leq \bigvee_{i \in I} \tau_i$, so there exists $i_0 \in I$ (the first τ_i that matches) such that $\texttt{case}\ e'\ (\tau_i \to e_i)_{i \in I} \hookrightarrow e_{i_0}$ and $(t' \wedge \tau_{i_0}) \smallsetminus (\bigvee_{j<i_0} \tau_j) \not\leq \mathbb{0}$, thus $\varnothing \vdash e_{i_0} : t$ which concludes.

□

Theorem B.7 (Static Type Safety). If $\varnothing \vdash_{\text{static}} e : t$ then either:

- $e \hookrightarrow^* v$ and $\varnothing \vdash_{\text{static}} v : t$;
- e diverges

Proof. By standard application of Lemmas B.5 and B.6. □

B.2 Safety of the Gradual System

The safety of the gradual type system is established on judgments $\Gamma \vdash e \fatsemi t$, for which we prove progress and preservation lemmas.

Lemma B.8 (Progress). If $\varnothing \vdash e \fatsemi t$ then either:

- $\exists v$ s.t. $e = v$;
- $\exists e'$ s.t. $e \hookrightarrow e'$;
- or $\exists p$ s.t. $e \hookrightarrow \omega_p$.

Proof. Straightforward by the definition of the reduction semantics. □

Lemma B.9 (Substitution). If $\Gamma, x : s \vdash e \fatsemi t$ then, for all $\Gamma \vdash e' \fatsemi s$, we have $\Gamma \vdash e[e'/x] \fatsemi t$.

Proof of Substitution. By induction on the derivation of $\Gamma, x : s \vdash e \fatsemi t$.

(1) Rule (var). $e = y$.
If $y = x$, then $s \leq t$ by inversion, Thus $e[e'/x] = e'$ and we conclude from $\Gamma \vdash e \fatsemi s$ by subsumption.
Otherwise, $y \neq x$ and $e[e'/x] = y$. By weakening from $\Gamma, x : s \vdash y \fatsemi t$, we get $\Gamma \vdash y \fatsemi t$.

(2) Rule (cst). Immediate by weakening.

$$(\text{cst}^\circ)\ \frac{}{\Gamma \vdash c \fatsemi c \wedge ?} \qquad (\text{var}^\circ)\ \frac{\Gamma(x) = t}{\Gamma \vdash x \fatsemi t}$$

$$(\lambda^\circ)\ \frac{\forall (t_i \to s_i) \in \mathbb{I} \quad (\Gamma, x : t_i \vdash e \fatsemi s_i) \qquad \Gamma, x : ? \vdash e \fatsemi \mathbb{1}}{\Gamma \vdash \lambda^{\mathbb{I}} x \,.\, e \fatsemi \bigwedge_{i \in I}(t_i \to s_i)}$$

$$(\lambda^\circ_\star)\ \frac{\Gamma \vdash \lambda^{\mathbb{I}} x.e \fatsemi t_1 \to t_2 \qquad \Gamma, x : ? \vdash e \fatsemi t_2 \wedge ?}{\Gamma \vdash \lambda^{\mathbb{I}} x.e \fatsemi (t_1 \to t_2)^\star} \qquad (\lambda^\circ_?)\ \frac{\Gamma \vdash \lambda^{\mathbb{I}} x.e \fatsemi t}{\Gamma \vdash \lambda^{\mathbb{I}} x.e \fatsemi ?}$$

$$(\text{app}^\circ)\ \frac{\Gamma \vdash e_1 \fatsemi t_1 \to t_2 \quad \Gamma \vdash e_2 \fatsemi t_1}{\Gamma \vdash e_1(e_2) \fatsemi t_2} \qquad (\text{app}^\circ_?)\ \frac{\Gamma \vdash e_1 \fatsemi t_1 \quad \Gamma \vdash e_2 \fatsemi t_2}{\Gamma \vdash e_1(e_2) \fatsemi ?}$$

$$(\text{app}^\circ_\star)\ \frac{\Gamma \vdash e_1 \fatsemi (t_1 \to t)^\star \quad \Gamma \vdash e_2 \fatsemi t_2}{\Gamma \vdash e_1(e_2) \fatsemi t \wedge ?}$$

$$(\text{case}^\circ_?)\ \frac{\Gamma \vdash e \fatsemi t \qquad \forall i \in I.\ (t \wedge \tau_i \not\leq \mathbb{0} \implies \Gamma \vdash e_i \fatsemi t')}{\Gamma \vdash \texttt{case}\, e\, (\tau_i \to e_i)_i \fatsemi t' \wedge ?}$$

$$(\text{plus}^\circ_?)\ \frac{\Gamma \vdash e_1 \fatsemi t_1 \quad \Gamma \vdash e_2 \fatsemi t_2}{\Gamma \vdash e_1 + e_2 \fatsemi \texttt{int} \wedge ?} \qquad (\text{sub}^\circ)\ \frac{\Gamma \vdash e \fatsemi t_1 \quad t_1 \leq t_2}{\Gamma \vdash e \fatsemi t_2}$$

Fig. 17. Typing rules for the strong system

(3) Rule (app).
We have: $e = e_1\, e_2, \Gamma, x : s \vdash e_1 \fatsemi t_1 \to t_2, \Gamma, x : s \vdash e_2 \fatsemi t_1$.
By IH: $\Gamma \vdash e_1[e'/x] \fatsemi t_1 \to t_2$ and $\Gamma \vdash e_2[e'/x] \fatsemi t_1$.
Applying (app) gives: $\Gamma \vdash (e_1\, e_2)[e'/x] \fatsemi t_2$

(4) Rule ($\text{app}_?$).
We have $e = e_1\, e_2, \Gamma, x : s \vdash e_1 \fatsemi t_1, \Gamma, x : s \vdash e_2 \fatsemi t_2$.
By IH: $\Gamma \vdash e_1[e'/x] \fatsemi t_1$ and $\Gamma \vdash e_2[e'/x] \fatsemi t_2$.
Applying ($\text{app}_?$) gives $\Gamma \vdash (e_1\, e_2)[e'/x] \fatsemi ?$.

(5) Rule ($\text{app}_\star$). Same as above.

(6) Rule (λ). $e = \lambda^{\mathbb{I}} y.e_0$
By inversion,

$$\begin{cases} \forall (t_i \to s_i) \in \mathbb{I}, (\Gamma, x : s, y : t_i \vdash e_0 \fatsemi s_i) \\ \Gamma, x : s, y : ? \vdash e_0 \fatsemi \mathbb{1} \end{cases}$$

We use the permutation lemma to switch x and y.
Then, by IH,

$$\begin{cases} \forall (t_i \to s_i) \in \mathbb{I}, (\Gamma, y : t_i \vdash e_0[e'/x] \fatsemi s_i) \\ \Gamma, y : ? \vdash e_0[e'/x] \fatsemi \mathbb{1} \end{cases}$$

Applying (λ) gives: $\Gamma \vdash (\lambda^{\mathbb{I}} y.e_0)[e'/x] \fatsemi \bigwedge_{i \in I}(t_i \to s_i)$

(7) Rule ($\lambda_\star$).
By inversion $\Gamma, x : s \vdash \lambda^{\mathbb{I}} y.e \fatsemi (t_1 \to t_2)^\star$ and $\Gamma, x : s, y : ? \vdash e \fatsemi t_2$.
By permutation on the second premise and IH, $\Gamma, y : ? \vdash e[e'/x] \fatsemi t_2$.
By IH on the first premise, $\Gamma \vdash \lambda^{\mathbb{I}} y.e[e'/x] \fatsemi t_1 \to t_2$.
We can then reapply ($\lambda_\star$) to conclude.

(8) Rule ($\lambda_?$). Immediate by IH.

(9) Rule ($\text{case}_?$). $e = \texttt{case}\, e\, (t_i \to e_i)_i$ and $\Gamma \vdash e \fatsemi t$.

For all i, if $t \wedge \tau_i \not\leq \mathbb{0}$, $\Gamma \vdash e_i ⦂ t'$.
By IH, $\Gamma \vdash e[e'/x] ⦂ t$ and for all i, $t \wedge \tau_i \not\leq \mathbb{0} \implies \Gamma \vdash e_i[e'/x] ⦂ t'$.
Applying (case$_?$) gives $\Gamma \vdash \texttt{case}\, e[e'/x]\ (\tau_i \to e_i[e'/x])_i ⦂ t'$.

(10) Rule (plus$_?$). Immediate by IH.
(11) Rule (sub). Immediate by IH.

□

Definition B.10 (Value type operator). We define the operator $\texttt{type}(\cdot)$ from values to types:

$$\texttt{type}(c) = c \wedge ?$$
$$\texttt{type}(\lambda^{\mathbb{I}}x.e) = \bigwedge_{(t \to s) \in \mathbb{I}} (t \to s)$$

Lemma B.11 (Value type operator). If $\varnothing \vdash_\circ v ⦂ t$, then $\varnothing \vdash_\circ v ⦂ \texttt{type}(v)$.

Proof. Trivial for a constant c as it is typed with $c \wedge ? \leq ?$. A simple application of subsumption concludes.

For a lambda-abstraction, both rules ($\lambda_?$) and ($\lambda_\star$) rely on rule (λ) being applied earlier, and (λ) typechecks exactly the interface of a function, which is $\texttt{type}(v)$. □

In the system for $\vdash e ⦂ t$, every well-typed closed expression can be typed with ?.

Lemma B.12 (Static typing implies dynamic typing). If $\varnothing \vdash e ⦂ t$ then $\varnothing \vdash e ⦂ ?$

Proof. We proceed by induction on the derivation of $\Gamma \vdash e ⦂ t$.

Rule (cst): By subtyping, $c\ \wedge ?\ \leq\ ?$.
Rule (var): Impossible in an empty context.
Rules (λ), ($\lambda_\star$): Add one use of the ($\lambda_?$) rule.
Rule ($\lambda_?$): Immediate.
Rules (app), (app$_\star$): Replace the use of (app) with (app$_?$).
Rule (app$_?$): Immediate.
Rule (case$_?$): By induction hypothesis, all branches can be typed with ?. Re-apply the rule with $t' = ?$.
Rule (plus$_?$): Immediate by subtyping $\texttt{int} \wedge ? \leq ?$.
Rule (sub): Immediate by induction hypothesis.

□

Lemma B.13 (Substitution by value). If $\Gamma, x : ? \vdash e ⦂ t$ then, for all well-typed value $\varnothing \vdash v ⦂ t'$, we have $\Gamma \vdash e[v/x] ⦂ t$.

Proof of Substitution by value. By induction on e.

- Case $e = c$: Immediate since $e[v/x] = c$.
- Case $e = y \neq x$: Immediate since $y \in \Gamma$ and $e[v/x] = y$.
- Case $e = x$: Then necessarily x is typed by rule (var) with $x : ?$. Hence $? \leq t$ with either $t = ?$ or $t = \mathbb{1}$. Since v is well-typed, we have $\varnothing \vdash v ⦂ ?$ by Lemma B.12. $e[v/x] = v$ and we conclude by subsumption.
- Case $e = e_1\, e_2$: Consider the typing rule used to type e.
  - Rule (app): By inversion, $\Gamma, x : ? \vdash e_1 ⦂ t_1 \to t_2$ and $\Gamma, x : ? \vdash e_2 ⦂ t_1$. By IH, $\Gamma \vdash e_1[v/x] ⦂ t_1 \to t_2$ and $\Gamma \vdash e_2[v/x] ⦂ t_1$. So by (app), $\Gamma \vdash (e_1\, e_2)[v/x] ⦂ t_2$.
  - Rule (app$_?$): Immediate by IH similar to above.
  - Rule (app$_\star$): Immediate by IH similar to above.
- Case $e = \lambda^{\mathbb{I}}y.e_0$: Consider the typing rule used.

  - Rule (λ): By inversion,

$$\begin{cases} \forall (t_i \to s_i) \in \mathbb{I}, (\Gamma, x : ?, y : t_i \vdash e_0 \fatsemi s_i) \\ \Gamma, x : ?, y : ? \vdash e_0 \fatsemi \mathbb{1} \end{cases}$$

    We use the permutation lemma to switch x and y. Then, by IH,

$$\begin{cases} \forall (t_i \to s_i) \in \mathbb{I}, (\Gamma, y : t_i \vdash e_0[v/x] \fatsemi s_i) \\ \Gamma, y : ? \vdash e_0[v/x] \fatsemi \mathbb{1} \end{cases}$$

    So we re-apply (λ) to get $\Gamma \vdash \lambda^{\mathbb{I}} y.e_0[v/x] \fatsemi \bigwedge_{i \in I}(t_i \to s_i)$.
  - Rule ($\lambda_?$): Immediate by IH.
  - Rule ($\lambda_\star$): By inversion $\Gamma, x : ? \vdash \lambda^{\mathbb{I}} y.e \fatsemi t_1 \to t_2$ and $\Gamma, x : ?, y : ? \vdash e \fatsemi t_2$. By permutation on the second premise and IH, $\Gamma, y : ? \vdash e[v/x] \fatsemi t_2$ By IH on the first premise, $\Gamma \vdash \lambda^{\mathbb{I}} y.e[v/x] \fatsemi t_1 \to t_2$. We can then reapply ($\lambda_\star$) to conclude.
- Case $e = \texttt{case}\, e'\ (\tau_i \to e_i)_i$: Rule (case$_?$). By inversion, $\Gamma, x : ? \vdash e' \fatsemi t$ and $\forall i$, if $t \wedge \tau_i \not\leq \mathbb{0}$ then $\Gamma, x : ? \vdash e_i \fatsemi t'$. By IH, $\Gamma \vdash e'[v/x] \fatsemi t$ and for all i, if $t \wedge \tau_i \not\leq \mathbb{0}$ then $\Gamma \vdash e_i[v/x] \fatsemi t'$. We can then reapply (case$_?$) to conclude.
- Case $e = e_1 + e_2$: By inversion, $\Gamma, x : ? \vdash e_1 \fatsemi t_1$ and $\Gamma, x : ? \vdash e_2 \fatsemi t_2$. By IH, $\Gamma \vdash e_1[v/x] \fatsemi t_1$ and $\Gamma \vdash e_2[v/x] \fatsemi t_2$. We can then reapply (plus$_?$) to conclude.
- Rule (sub): Immediate by IH and reapplying (sub).

□

Lemma B.14. (Subject Reduction) If $\Gamma \vdash e \fatsemi t$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' \fatsemi t$.

Proof of Subject Reduction. By induction on the derivation of $\Gamma \vdash e \fatsemi t$ and case analysis on the reduction rule: If the last rule is subsumption, we can directly apply the IH to the premise and obtain the result.

Reduction $\mathcal{E}$: $e = \mathcal{E}[e_0]$ with $e_0 \hookrightarrow e_0'$ and $\mathcal{E} \neq \square$. Expression e_0 is typed by a subtree of the derivation tree of $\Gamma \vdash e \fatsemi t$. Thus, by IH, its type is preserved after reduction. Hence the type of $\mathcal{E}[e_0']$ is preserved.

Reduction [β]: $e = (\lambda^{\mathbb{I}} x.e_1)\ v_2$

Consider the last rule used to type the application.

- **Rule (app):** This case implies type preservation by substitution lemma. Indeed, by inversion we have $\Gamma \vdash \lambda^{\mathbb{I}} x.e_1 \fatsemi t' \to t$. With $\Gamma \vdash v_2 \fatsemi t'$, by substitution lemma, $\Gamma \vdash e_1[v_2/x] \fatsemi t$.
- **Rule (app$_?$):** We prove that the result of the reduction is "$\fatsemi$-well-typed". By inversion, $\Gamma, x : ? \vdash e_1 \fatsemi \mathbb{1}$. Since $\Gamma \vdash v_2 \fatsemi t_2$, by Lemma B.13, we have $\Gamma \vdash e_1\ [v_2/x] \fatsemi \mathbb{1}$. We conclude by Lemma B.12 that $\Gamma \vdash e_1\ [v_2/x] \fatsemi ?$.
- **Rule (app$_\star$):** By inversion, $\Gamma, x : ? \vdash e_1 \fatsemi t \wedge ?$. Since $\Gamma \vdash v_2 \fatsemi t_2$, by lemma B.13, $\Gamma \vdash e_1[v_2/x] \fatsemi t \wedge ?$ which concludes.

Reduction [+]: The result is immediately a well-typed integer.

Reduction [case]: We reduce to a branch of the same type.

□

To link back to the gradual system, we use the fact that every expression well-typed in the former is well-typed in the latter.

Lemma B.15 (Gradual typing implies strong typing)**.** If $\Gamma \vdash e : t$ then $\Gamma \vdash e \fatsemi t$.

Proof. Every rule in the gradual system of Figure 16 has a more general counterpart in 17, hence this is trivial. □

$$\text{(tuple)}\ \frac{\forall i = 1..n. \quad (e_i : t_i)}{\{e_1, \ldots, e_n\} : \{t_1, \ldots, t_n, ..\}} \qquad \text{(proj)}\ \frac{e' : \bigvee_{i \in K} i \quad e : \{t_0, \ldots, t_n, ..\}}{\pi_{e'}\, e : \bigvee_{i \in K} t_i}\ K \subseteq [0, n]$$

$$\text{(proj}_\omega\text{)}\ \frac{e' : \texttt{int} \quad e : \{t_0, \ldots, t_n\}}{\pi_{e'}\, e : \bigvee_{i \leq n} t_i} \qquad \text{(proj}^{\mathbb{1}}_\omega\text{)}\ \frac{e' : \texttt{int} \quad e : \texttt{tuple}}{\pi_{e'}\, e : \mathbb{1}}$$

Fig. 18. Typing rules for tuples

Now, the type safety in the gradual system is ensured by the safety of the strong system.

Theorem B.16. If $\varnothing \vdash e : t$, then either e diverges, or e crashes on a runtime error ω, or e evaluates to a value v such that $\varnothing \vdash v \fatsemi t$.

Proof of Type safety. Corollary of the subject reduction B.14 and progress B.8 lemmas. □

B.3 Extension for tuples

The rules to type-check tuples and projections are in Figure 18. Using only rules (tuple) and (proj) is going to prevent runtime errors $\omega_{\text{OutOfRange}}$ in the system $\vdash$, while adding rules (proj$_\omega$) and (proj$^{\mathbb{1}}_\omega$) allows to type-check unsafe projections.

Deriving a type using the static rules of $\vdash_{\text{static}}$ and the ω rules will be written $\vdash_{\text{static}_\omega}$.

Adapting previous Lemmas to account for the new runtime errors gives us this type safety result:

Theorem B.17 (Type Safety with Tuples)**.** If $\varnothing \vdash_{\text{static}_\omega} e : t$ then either e diverges, or e crashes on a runtime error ω, or e evaluates to a value v such that $\varnothing \vdash v \vdash_{\text{static}_\omega} t$.

C Dynamic type tests

$B(c)$ maps constants onto their base types (e.g. integers i onto `int`)

$\forall c \qquad c \in B(c)$

$\forall x, e, t \qquad (\lambda^{\mathbb{I}} x.e) \in \texttt{function}$

$\forall v_1, .., v_n \qquad \{v_1, .., v_n\} \in \{\tau_1, .., \tau_n\} \iff \forall i = 1..n \quad v_i \in \tau_i$

Fig. 19. Inductive Definition for $v \in \tau$ (Section 2)

D Gradual Strong Function Types

Definition D.1 (Gradual Strong function type)**.** Consider F the operator $(\bullet)^\star$ that, given a type t, returns its strong type. This operator is not monotonic, so with Remark 6.16 of [29] we define its gradual extension as:

$$\tilde{F}(t) = (F(t^{\Downarrow}) \wedge F(t^{\Uparrow})) \vee ((F(t^{\Downarrow}) \vee F(t^{\Uparrow})) \wedge ?)$$

For instance, $(? \to ?)^\star = (F(\mathbb{1} \to \mathbb{0}) \wedge F(\mathbb{0} \to \mathbb{1})) \vee ((F(\mathbb{1} \to \mathbb{0}) \vee F(\mathbb{0} \to \mathbb{1})) \wedge ?)$ What are those?

- $\mathbb{0} \to \mathbb{1}$ cannot be applied. Every function is a subtype of it;
- $(\mathbb{0} \to \mathbb{1})^\star \simeq \mathbb{0} \to \mathbb{1}$ (it is already strong, in that if a value is returned, it will be of type $\mathbb{1}$);
- $\mathbb{1} \to \mathbb{0}$ is a function that, for every input, errors or diverges;
- $(\mathbb{1} \to \mathbb{0})^\star$ puts a strong condition that every input outside the domain leads to a value in the codomain. But the domain is $\mathbb{1}$ so there are no such values. Thus $(\mathbb{1} \to \mathbb{0})^\star \simeq \mathbb{1} \to \mathbb{0}$.
- similarly, for every static type t, $(\mathbb{1} \to t)^\star = \mathbb{1} \to t$ (the negation of the domain is empty) and $(t \to \mathbb{1})^\star = t \to \mathbb{1}$ (the codomain is $\mathbb{1}$), but also

Fig. 20. **Guard Judgments.**

$$\text{with } \mathfrak{b}, \mathfrak{c} \in \{\mathtt{true}, \mathtt{false}\}$$

$$\begin{array}{ll}
\textbf{Pattern Matching Analysis} & \Gamma; t \vdash \overline{pg} \rightsquigarrow \overline{\mathcal{A}} \\
\textbf{Guard Analysis} & \Gamma; p \vdash g \mapsto \mathcal{R}
\end{array}$$

$$\begin{array}{lrcl}
\textbf{Accepted Types} & \mathcal{A} & ::= & \overline{(t, \mathfrak{b})} \\
\textbf{Results} & \mathcal{R} & ::= & \overline{\{\mathcal{S}\,;\,\mathcal{T}\}} \mid \mathcal{F} \\
\textbf{Environments} & \mathcal{S}, \mathcal{T} & ::= & (\Gamma, \mathfrak{b}) \\
\textbf{Failure Results} & \mathcal{F} & ::= & \omega \mid \{\mathcal{S}; \mathtt{false}\}
\end{array}$$

Fig. 21. **Guard Syntax.**

$$\begin{array}{lcll}
\textbf{Guards} & g & ::= & a \mathrel{?} \tau \mid a = a \mid a \mathrel{\mathtt{!=}} a \mid g \mathrel{\mathtt{and}} g \mid g \mathrel{\mathtt{or}} g \\
\textbf{Guard atoms} & a & ::= & c \mid x \mid \pi_a\, a \mid \mathtt{size}\, a \mid \{\overline{a}\} \\
\textbf{Test types} & \tau & ::= & b \mid c \mid \mathtt{function}_n \mid \{\overline{\tau}\} \mid \tau \vee \tau \mid \neg\tau
\end{array}$$

In the end,

$$\begin{aligned}
(? \to ?)^\star &= ((\mathbb{1} \to \mathbb{0})^\star \wedge (\mathbb{0} \to \mathbb{1})^\star) \vee (((\mathbb{1} \to \mathbb{0})^\star \vee (\mathbb{0} \to \mathbb{1})^\star) \wedge ?) \\
&= (\mathbb{1} \to \mathbb{0}) \wedge (\mathbb{0} \to \mathbb{1}) \vee (((\mathbb{1} \to \mathbb{0}) \vee (\mathbb{0} \to \mathbb{1})) \wedge ?) \\
&= (\mathbb{1} \to \mathbb{0}) \vee ((\mathbb{0} \to \mathbb{1}) \wedge ?)
\end{aligned}$$

Another example (contravariant dynamic):

$$\begin{aligned}
(? \to \mathtt{int})^\star &= ((\mathbb{1} \to \mathtt{int})^\star \wedge (\mathbb{0} \to \mathtt{int})^\star) \vee (((\mathbb{1} \to \mathtt{int})^\star \vee (\mathbb{0} \to \mathtt{int})^\star) \wedge ?) \\
&= ((\mathbb{1} \to \mathtt{int}) \wedge (\mathbb{0} \to \mathtt{int})^\star) \vee (((\mathbb{1} \to \mathtt{int}) \vee (\mathbb{0} \to \mathtt{int})^\star) \wedge ?) \\
&= (\mathbb{1} \to \mathtt{int}) \vee ((\mathbb{0} \to \mathtt{int})^\star \wedge ?)
\end{aligned}$$

Another example (covariant dynamic):

$$\begin{aligned}
(\mathtt{int} \to ?)^\star &= ((\mathtt{int} \to \mathbb{0})^\star \wedge (\mathtt{int} \to \mathbb{1})^\star) \vee (((\mathtt{int} \to \mathbb{0})^\star \vee (\mathtt{int} \to \mathbb{1})^\star) \wedge ?) \\
&= ((\mathtt{int} \to \mathbb{0})^\star \wedge (\mathtt{int} \to \mathbb{1})) \vee (((\mathtt{int} \to \mathbb{0})^\star \vee (\mathtt{int} \to \mathbb{1})) \wedge ?) \\
&= (\mathtt{int} \to \mathbb{0})^\star \vee ((\mathtt{int} \to \mathbb{1})^\star \wedge ?)
\end{aligned}$$

E Guard Analysis

Note that in Figure 21 there is no negation on guards. Indeed, the first thing we do is eliminate all negations from guards by pushing them on the terminal guards, e.g., $\mathtt{not}\ a = a$ becomes $a \neq a$.

Fig. 22. **Accepted Types Productions**

$$[\textsc{accept}]\ \frac{\Gamma, t/_p \vdash g \mapsto \{\_;(\Delta_i, \mathfrak{b}_i)\}_i}{\Gamma; t \vdash pg \rightsquigarrow \big(\llparenthesis p \rrparenthesis_{\Delta_i}, \mathfrak{b}_i\big)_i}
\qquad
[\textsc{fail}]\ \frac{\Gamma, t/_p \vdash g \mapsto \mathcal{F}}{\Gamma; t \vdash pg \rightsquigarrow (\mathbb{0}, 1)}$$

$$[\textsc{seq}]\ \frac{\Gamma; t \vdash pg \rightsquigarrow \mathcal{A} \qquad \Gamma; t \setminus \big(\bigvee_{(s,\mathtt{true}) \in \mathcal{A}} s\big) \vdash \overline{pg} \rightsquigarrow \overline{\mathcal{A}}}{\Gamma; t \vdash pg\ \overline{pg} \rightsquigarrow \mathcal{A}\ \overline{\mathcal{A}}}$$

Fig. 23. **Guard Analysis Rules**

$$[\textsc{true}]\ \frac{\Gamma \vdash a : t}{\Gamma \vdash a\ ?\ t \mapsto \{(\Gamma, 1)\,;\,(\Gamma, 1)\}} \qquad [\textsc{false}]\ \frac{\Gamma \vdash a : s \qquad s \wedge t \simeq \mathbb{0}}{\Gamma \vdash a\ ?\ t \mapsto \{(\Gamma, 1)\,;\,\texttt{false}\}}$$

$$[\textsc{var}]\ \frac{\Gamma(x) \not\leq t \qquad \Gamma(x) \wedge t \not\simeq \mathbb{0}}{\Gamma \vdash x\,?\,t \mapsto \big\{(\Gamma, 1)\,;\,\big(\Gamma[x \mathrel{\hat{=}} t]_p, 1\big)\big\}} \qquad [\textsc{size}]\ \frac{\begin{array}{c}\Gamma \vdash a\,?\,\texttt{tuple} \mapsto \{\_\,;\,(\Phi, \mathfrak{b})\}\\ \Gamma \vdash a\,?\,\texttt{tuple}^{i} \mapsto \{\_\,;\,\mathfrak{A}\}\end{array}}{\Gamma \vdash \texttt{size}\,a\,?\,i \mapsto \{(\Phi, \mathfrak{b})\ ;\ \mathfrak{A}\}}$$

$$[\textsc{proj}]\ \frac{\begin{array}{c}\Gamma \vdash a' : i\\ \Gamma \vdash a\ ?\ \texttt{tuple}^{>i} \mapsto \{\_\,;\,(\Delta, \mathfrak{b})\}\end{array} \quad \Delta \vdash a\ ?\ \{\ \overbrace{\mathbb{1}, \ldots, \mathbb{1}}^{i \text{ times}}\ , t, ..\} \mapsto \mathscr{T}}{\Gamma \vdash \pi_{a'}\ a\ ?\ t \mapsto \{(\Delta, \mathfrak{b})\ ;\ \mathscr{T}\}}$$

$$[\textsc{eq}_1]\ \frac{\Gamma \vdash a_1 : c \quad \Gamma \vdash a_2\,?\,c \mapsto \mathcal{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathcal{R}} \qquad [\textsc{eq}_2]\ \frac{\Gamma \vdash a_2 : c \quad \Gamma \vdash a_1\,?\,c \mapsto \mathcal{R}}{\Gamma \vdash a_1 = a_2 \mapsto \mathcal{R}}$$

Fig. 24. **Guard Analysis Boolean Rules**

$$[\textsc{and}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i=1..n}\\ \forall i \text{ such that } \Delta_i \vdash g_2 \mapsto \big\{(\Phi_{ij}, \mathfrak{b}_{ij})\,;\,(\Delta_{ij}, \mathfrak{c}_{ij})\big\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1\,\mathsf{and}\,g_2 \mapsto \big\{\mathfrak{A}_{ij}\,;\,(\Delta_{ij}, \mathfrak{c}_i \,\&\, \mathfrak{c}_{ij})\big\}_{ij}} \qquad \mathfrak{A}_{ij} = \begin{cases}(\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = 1\\ & \text{and } \Phi_{ij} = \Delta_i\\ (\Phi_{ij}, \mathfrak{b}_i \,\&\, \mathfrak{b}_{ij}) & \text{else}\end{cases}$$

$$[\textsc{or}]\ \frac{\begin{array}{c}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i=1..n}\\ \forall i \quad \Gamma, {}^{t_i}\!/_{p} \vdash g_2 \mapsto \big\{(\Phi_{ij}, \mathfrak{b}_{ij})\,;\,(\Delta_{ij}, \mathfrak{c}_{ij})\big\}_{j=1..m_i}\end{array}}{\Gamma \vdash g_1\,\mathsf{or}\,g_2 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_i \ @\ \big\{\mathfrak{A}_{ij}\,;\,(\Delta_{ij}, \mathfrak{c}_i \,\&\, \mathfrak{c}_{ij})\big\}_{ij}} \qquad \begin{array}{l}\mathfrak{A}_{ij} = \begin{cases}(\Phi_i, \mathfrak{b}_i) & \text{if } \mathfrak{b}_{ij} = 1\\ & \text{and } \Phi_{ij} = \Gamma, \left({}^{t_i}\!/_{p}\right)\\ (\Phi_{ij}, \mathfrak{b}_i \,\&\, \mathfrak{b}_{ij}) & \text{else}\end{cases}\\ t_i = \begin{cases}\lbag p \rbag_{\Phi_i} \setminus \lbag p \rbag_{\Delta_i} & \text{if } \mathfrak{c}_i = 1\\ \lbag p \rbag_{\Phi_i} & \text{if } \mathfrak{c}_i = 0\end{cases}\end{array}$$

Fig. 25. **Guard Analysis Approx Rules**

$$[\textsc{proj}]\ \frac{\begin{array}{c}\Gamma \vdash a'\,?\,\texttt{int} \mapsto \{\_\,;\,(\Phi, \mathfrak{b})\}\\ \Phi \vdash a\,?\,\texttt{tuple} \mapsto \{\_\,;\,(\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash \pi_{a'}\ a\,?\,t \mapsto \{(\Delta, 0)\,;\,(\Delta, 0)\}} \qquad [\textsc{eq}]\ \frac{\begin{array}{c}\Gamma \vdash a_0\,?\,\mathbb{1} \mapsto \{\_\,;\,(\Phi, \mathfrak{b})\}\\ \Phi \vdash a_1\,?\,\mathbb{1} \mapsto \{\_\,;\,(\Delta, \mathfrak{c})\}\end{array}}{\Gamma \vdash a_0 = a_1 \mapsto \{(\Delta, \mathfrak{b} \,\&\, \mathfrak{c})\,;\,(\Delta, 0)\}}$$

$$[\textsc{size}]\ \frac{\Gamma \vdash a\,?\,\texttt{tuple} \mapsto \{\_\,;\,(\Delta, \mathfrak{c})\} \qquad t \wedge \texttt{int} \not\leq \mathbb{0}}{\Gamma \vdash \texttt{size}\,a\,?\,t \mapsto \{(\Delta, \mathfrak{c})\,;\,(\Delta, 0)\}}$$

Fig. 26. **Guard Analysis False/Failure Rules**

$$[\textsc{size}_\omega]\ \frac{\Gamma \vdash a\ ?\ \texttt{tuple} \mapsto \mathcal{F}}{\Gamma \vdash \texttt{size}\,a\,?\,t \mapsto \omega} \qquad [\textsc{eq}_\omega]\ \frac{\Gamma \vdash a_i\ ?\ \mathbb{1} \mapsto \mathcal{F}}{\Gamma \vdash a_0 = a_1 \mapsto \omega}\ i \in \{0, 1\} \qquad [\textsc{proj}_\omega]\ \frac{\Gamma \vdash a\ ?\ \texttt{tuple} \mapsto \mathcal{F}}{\Gamma \vdash \pi_{a'}\ a\,?\,t \mapsto \omega}$$

$$[\textsc{proj}_\omega]\ \frac{\Gamma \vdash a'\,?\,\texttt{int} \mapsto \mathcal{F}}{\Gamma \vdash \pi_{a'}\ a\,?\,t \mapsto \omega} \qquad [\textsc{bound}_\omega]\ \frac{\Gamma \vdash a' : i \qquad \Gamma \vdash a\,?\,\texttt{tuple}^{>i} \mapsto \mathcal{F}}{\Gamma \vdash \pi_{a'}\ a\,?\,t \mapsto \omega}$$

$$[\textsc{orF}]\ \frac{\begin{array}{l}\Gamma \vdash g_1 \mapsto \{(\Phi, \mathfrak{b})\,;\,\texttt{false}\}\\ \Phi \vdash g_2 \mapsto \mathcal{R}\end{array}}{\Gamma \vdash g_1\,\mathsf{or}\,g_2 \mapsto \mathcal{R}} \qquad [\textsc{andF}]\ \frac{\Gamma \vdash g_1 \mapsto \mathcal{F}}{\Gamma \vdash g_1\,\mathsf{and}\,g_2 \mapsto \mathcal{F}} \qquad [\textsc{andF}]\ \frac{\begin{array}{l}\Gamma \vdash g_1 \mapsto \{(\Phi_i, \mathfrak{b}_i)\,;\,(\Delta_i, \mathfrak{c}_i)\}_{i \leq n}\\ \forall i \leq n \quad \Delta_i \vdash g_2 \mapsto \mathcal{F}_i\end{array}}{\Gamma \vdash g_1\,\mathsf{and}\,g_2 \mapsto \begin{cases}\omega & \text{if } \forall i,\ \mathcal{F}_i = \omega\\ \overline{\mathcal{F}}^{j} & j \text{ s.t. } \mathcal{F}_j \neq \omega\end{cases}}$$

Fig. 27. **Accepted types**

$$\begin{array}{ll} \lbag x \rbag_\Gamma = \Gamma(x) \text{ if } x \in \mathsf{dom}\,(\Gamma) & \lbag c \rbag_\Gamma = c \\ \lbag x \rbag_\Gamma = \mathbb{1} \text{ if } x \notin \mathsf{dom}\,(\Gamma) & \lbag p_1 \,\&\, p_2 \rbag_\Gamma = \lbag p_1 \rbag_\Gamma \wedge \lbag p_2 \rbag_\Gamma \\ \lbag \{p_1, .., p_n\} \rbag_\Gamma = \{\lbag p_1 \rbag_\Gamma, .., \lbag p_n \rbag_\Gamma\} & \end{array}$$

Fig. 28. **Typing Environments**

If $t \leq \lbag p \rbag$ then $t/_p$ is a map from the variables of p to types:

$$\begin{array}{lll} t/_x(x) & = t & \\ t/_{\{p_1,\ldots,p_n\}}(x) & = t/_{p_i}(x) & \text{where } \exists i \text{ unique s.t. } x \in \text{vars}(p_i) \\ t/_{p_1 \& p_2}(x) & = t/_{p_1}(x) & \text{if } x \in \mathsf{Vars}\,(p_1) \\ t/_{p_1 \& p_2}(x) & = t/_{p_2}(x) & \text{if } x \notin \mathsf{Vars}\,(p_1) \text{ and } x \in \mathsf{Vars}\,(p_2) \end{array}$$

Fig. 29. **Environment Updates**

$$\begin{array}{lll} \forall y \in \mathsf{dom}\,(\Gamma),\ \Gamma[x \mathrel{\hat{=}} t](y) & = \begin{cases} \Gamma(y) & \text{if } y \neq x \\ \Gamma(x) \wedge t & \text{if } y = x \end{cases} & \\ \Gamma[x \mathrel{\hat{=}} t]_p & = (\Gamma[x \mathrel{\hat{=}} t]\,,\, t'/p) & \text{where } t' = \lbag p \rbag_{\Gamma[x \hat{=} t]} \end{array}$$

Definition E.1 (Skeleton). For all expressions e, we define the skeleton of this expression $\mathsf{sk}\,(e)$ as:

$$\begin{array}{rll} \mathsf{sk}\,(\mathsf{x}) & = \mathsf{x} & \\ \mathsf{sk}\,(\{e_1, \ldots, e_n\}) & = \{\mathsf{sk}\,(e_1), \ldots, \mathsf{sk}\,(e_n)\} & \\ \mathsf{sk}\,(e) & = \mathbb{1} & \text{for any other expression} \end{array}$$

The skeleton of an expression is a pattern that matches the structure and variables of that expression while leaving out any functional parts (for example, the skeleton of an application $e(e_1, \ldots, e_n)$ is $\mathbb{1}$ which is the pattern that matches any expression).

E.1 Typing with Guards

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0} \ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash e_i : s)}{\Gamma \vdash \mathsf{case}\ e\ (p_i g_i \to e_i)_{i \leq n}\ : s}\ t \leq \bigvee_{i \leq n} \lbag p_i g_i \rbag$$

$$\text{(case}_\omega\text{)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0} \ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash e_i : s)}{\Gamma \vdash \mathsf{case}\ e\ (p_i g_i \to e_i)_{i \leq n}\ : s}\ t \leq \bigvee_{i \leq n} \rbag p_i g_i \lbag$$

$$\text{(case}_\star\text{)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{0} \ \Rightarrow\ \Gamma, t_{ij}/_{p_i} \vdash e_i : s)}{\Gamma \vdash \mathsf{case}\ e\ (p_i g_i \to e_i)_{i \leq n}\ : ? \wedge s}\ t \mathrel{\widetilde{\leq}} \bigvee_{i \leq n} \rbag p_i g_i \lbag$$

where $\Gamma\,; t \vdash (p_i g_i)_{i \leq n} \leadsto (t_{ij}, \mathfrak{b}_{ij})_{i \leq n, j \leq m_i}$ and $\rbag p_i g_i \lbag = \bigvee_{j \leq m_i} t_{ij}$ and $\lbag p_i g_i \rbag = \bigvee_{\{j \leq m_i \,|\, \mathfrak{b}_{ij}\}} t_{ij}$

Fig. 30. Case Typing Rules

We tie the guard analysis to the operational semantics. First, the

Lemma E.1 (Success environment). If $\Gamma; p \vdash g \mapsto \mathcal{R}$ then for all $\{\mathscr{S}\,;\,\mathscr{T}\}$ for all $\{(\Phi, \mathfrak{b})\,;\,(\Delta, \mathfrak{c})\} \in \mathcal{R}$, for all value v,

$$\begin{align}
(v/(pg) \not\hookrightarrow^{*} \omega) &\Longrightarrow (v : \Lbag p \Rbag_\Delta) \tag{3}\\
\text{if } (\mathfrak{b} = \texttt{true}) \text{ then } (v : \Lbag p \Rbag_\Delta) &\Longrightarrow v/(pg) \not\hookrightarrow^{*} \texttt{true}) \tag{4}\\
v/(pg) \hookrightarrow^{\star}\texttt{true} \;&\Longrightarrow\; (v : \Lbag p \Rbag_\Delta) \tag{5}\\
\text{if } (\mathfrak{c} = \texttt{true}) \text{ then } (v : \Lbag p \Rbag_\Delta) \;&\Longrightarrow\; v/(pg) \hookrightarrow^{\star}\texttt{true}) \tag{6}\\
& \tag{7}
\end{align}$$

Lemma E.2 (Surely accepted types are sufficient). Given $\Gamma\,; t \vdash (p_i g_i)_{i \le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i \le n, j \le m_i}$, for all i, j such that $\mathfrak{b}_{ij}$, for all value v,

$$(v : t_{ij}) \;\Longrightarrow\; \exists (i_0 \le i) \text{ s.t. } (v/p_{i_0} g_{i_0} \neq \texttt{fail})$$

Lemma E.3 (Possibly accepted types are necessary). Given $\Gamma\,; t \vdash (p_i g_i)_{i \le n} \rightsquigarrow (t_{ij}, \mathfrak{b}_{ij})_{i \le n, j \le m_i}$, for all i, j such that $\mathfrak{b}_{ij}$, for all value v,

$$(v/(p_i g_i) \neq \texttt{fail}) \;\Longrightarrow\; \exists j \le m_i \text{ s.t. } (v : t_{ij})$$

Our previous theorem for type soundness can be extended

Theorem E.4 (Static Soundness). For every expression e such that $\varnothing \vdash e : t$ derived with the rules of Figure 3 except the ω-rules, and the rule (case) of Figure 30, then either:

- $\exists v$ s.t. $v : t$ and $e \hookrightarrow^{*} v$;
- e diverges

Proof. Updating this proof to account for the new case expression requires updating the proofs of progress and preservation. Here is a sketch of the proof:

(1) For progress, if e is a case $\texttt{case}\ e\ \overline{pg \longrightarrow e}$ that reduces to Ω, then for all $i \le n$, $v/p_i g_i = \texttt{fail}$. This contradicts the fact that every $v \in \bigvee_{i \le n} \Lbag p_i g_i \Rbag$ is a surely accepted value, i.e. should be accepted by at least one guarded-pattern per Lemma E.2

(2) Preservation comes almost immediately from rule (case): since every branch is typed with the same type from the whole case (this is enabled by subtyping), a case reduction will preserve the typing. Making sure that we always reduce into a well-typed expression comes from Lemma E.3 which ensures that our value (which is captured and substituted into the branch) is of type t_{ij} for some i, j; thus the Substitution Lemma B.4 applies and the branch is well-typed.

□

Theorem E.5 (Soundness). For every expression e such that $\varnothing \vdash e : t$ derived with the rules of Figure 3, and the non-gradual rules of Figure 30, then either:

- $\exists v$ s.t. $v : t$ and $e \hookrightarrow^{*} v$;
- $e \hookrightarrow^{*} \omega_{\text{OutOfRange}}$ or $e \hookrightarrow^{*} \omega_{\text{CaseEscape}}$;
- e diverges

Proof. Our analysis of the guarded-patterns is best-case/worst-case: when the surely accepted types do not coincide with the possibly accepted types, and are too small to type-check the program, we can use rule (case$_\omega$) to type-check. But this allows values to escape pattern matching, thus adding error $\omega_{\text{CaseEscape}}$ to the soudness result. □

Theorem E.6 (Gradual Soundness)**.** For every expression e such that $\varnothing \vdash e : t$ derived with the rules of Figures 3,4,5 and the rules of Figure 30,

- $e \hookrightarrow^* v$ with $v : t'$ and $t' \lesseqqgtr t$;
- $e \hookrightarrow^* \omega_p$ for some p;
- e diverges

F Semantic subtyping: multi-arity functions

F.1 Set Semantics

Definition F.1. Let $X_1, .., X_n$ and Y be subsets of D. We define

$$(X_1, .., X_n) \to Y = \left\{ R \in \mathcal{P}_f\,(D^n \times D_\omega) \mid \forall (d_1, .., d_n, \delta) \in R.\ (\forall i \in \{1, ..., n\}.\ d_i \in X_i) \implies \delta \in Y \right\}$$

Lemma F.2. For all $X_1, .., X_n, Y$ subsets of D,

$$(X_1, .., X_n) \to Y = \mathcal{P}\left(\overline{X_1 \times .. \times X_n \times \overline{Y}^{D_\omega}}^{D^n \times D_\omega} \right)$$

Theorem F.3 (Multi-arity Set-Containment)**.** Let $n \in \mathbb{N}$. Let $(X_i^{(1)})_{i\in P}, .., (X_i^{(n)})_{i \in P}$, $(X_i)_{i\in P}$, $(Y_i^{(1)})_{i\in N}, .., (Y_i^{(n)})_{i\in N}$, $(Y_i)_{i\in N}$ be families of subsets of the domain D. Then,

$$\bigcap_{i\in P} \left(X_i^{(1)}, .., X_i^{(n)}\right) \to X_i \subseteq \bigcup_{i\in N} \left(Y_i^{(1)}, .., Y_i^{(n)}\right) \to Y_i \iff \begin{array}{c} \exists i_0 \in N.\ \textbf{such that} \\ \forall \iota : P \to [1, n+1] \end{array} \begin{cases} \exists j \in [1, n].\ Y_{i_0}^{(j)} \subseteq \bigcup\limits_{\{i \in P \mid \iota(i) = j\}} X_i^{(j)} \\ \textbf{or} \bigcap\limits_{\{i\in P \mid \iota(i) = n+1\}} X_i \subseteq Y_{i_0} \end{cases}$$

Proof. Using Theorems (4.7) and (4.8) from [24].

$$\begin{aligned}
& \bigcap_{i\in P} \left(X_i^{(1)}, .., X_i^{(n)}\right) \to X_i \subseteq \bigcup_{i\in N} \left(Y_i^{(1)}, .., Y_i^{(n)}\right) \to Y_i \\
\overset{(4.7)}{\Leftrightarrow}\quad & \bigcap_{i\in P} \mathcal{P}\left(\overline{X_i^{(1)} \times .. \times X_i^{(n)} \times \overline{X_i}^{D_\omega}}^{D^n \times D_\omega} \right) \subseteq \bigcup_{i\in N} \mathcal{P}\left(\overline{Y_i^{(1)} \times .. \times Y_i^{(n)} \times \overline{Y_i}^{D_\omega}}^{D^n \times D_\omega} \right) \\
\overset{(4.8)}{\Leftrightarrow}\quad & \exists i_0 \in N.\ \bigcap_{i\in P} \overline{X_i^{(1)} \times .. \times X_i^{(n)} \times \overline{X_i}^{D_\omega}}^{D^n \times D_\omega} \subseteq \overline{Y_{i_0}^{(1)} \times .. \times Y_{i_0}^{(n)} \times \overline{Y_{i_0}}^{D_\omega}}^{D^n \times D_\omega} \\
\Leftrightarrow\quad & \exists i_0 \in N.\ \bigcup_{\iota : P \to [\![1;n+1]\!]} \left(\bigcap_{\{i\in P\,;\,\iota(i)=1\}} \overline{X_i^{(1)}}^{D} \times \cdots \times \bigcap_{\{i\in P\,;\,\iota(i)=n\}} \overline{X_i^{(n)}}^{D} \times \bigcap_{\{i\in P\,;\,\iota(i)=n+1\}} X_i \right) \subseteq \overline{Y_{i_0}^{(1)} \times .. \times Y_{i_0}^{(n)} \times \overline{Y_{i_0}}^{D_\omega}}^{D^n \times D_\omega} \\
\Leftrightarrow\quad & \exists i_0 \in N.\ \bigcup_{\iota : P \to [\![1;n+1]\!]} \left(\left(Y_{i_0}^{(1)} \cap \bigcap_{\{i\in P\,;\,\iota(i)=1\}} \overline{X_i^{(1)}}^{D} \right) \times \cdots \times \left(Y_{i_0}^{(n)} \cap \bigcap_{\{i\in P\,;\,\iota(i)=n\}} \overline{X_i^{(n)}}^{D} \right) \times \left(\overline{Y_{i_0}}^{D} \cap \bigcap_{\{i\in P\,;\,\iota(i)=n+1\}} X_i \right) \right) = \varnothing
\end{aligned}$$

□

F.2 Subtyping algorithm

From the proof of Theorem F.3 we know that the subtyping problem

$$\bigwedge_{i\in P} (t_i^{(1)}, ..., t_i^{(n)}) \to t_i \leq \bigvee_{j\in N} (t_j^{(1)}, ..., t_j^{(n)}) \to t_j \tag{8}$$

is decided by finding a single arrow on the right hand side such that

$$\bigwedge_{f\in P} f \;\leq\; (t_1,..,t_n) \to t \tag{9}$$

where P is a set of arrows of arity n. Following Frisch [24], we can define a backtrack-free algorithm that for all $n \in \mathbb{N}$ decides (9). This is expressed by function Φ_n of $n+2$ arguments defined as:

$$\begin{array}{lcl} \Phi_n(t_1,..,t_n,t,\varnothing) & = & \big(\exists j \in [\![1;n]\!].\ t_j \leq \mathbb{0}\big) \textbf{ or } (t \leq \mathbb{0}) \\ \Phi_n(t_1,..,t_n,t,\{(t'_1,..,t'_n) \to t'\} \cup P) & = & (\Phi_n(t_1,..,t_n,t \wedge t',P) \textbf{ and} \\ & & \qquad \forall j \in [\![1;n]\!].\ \Phi_n(t_1,..,t_j \setminus t'_j,..,t_n,t,P)) \end{array}$$

Theorem F.4. For all $n \in \mathbb{N}$, for P a set of arrows of arity n,

$$\bigwedge_{f\in P} f \leq (s_1,..,s_n) \to s \iff \Phi_n(s_1,..,s_n,\neg s,P)$$

Proof. From the proof of F.3, we know that deciding (9) is equivalent to the Boolean proposition (where the arrows in P are indexed by P as in (8) – e.g., $(t_i^{(1)},...,t_i^{(n)}) \to t_i$ for $i \in P$ – and the single arrow is $(s_1,..,s_n) \to s$):

$$\forall \iota : P \to [1,n+1] \begin{cases} \exists j \in [1,n].\ s_j \leq \bigvee\limits_{\{i\in P \mid \iota(i)=j\}} t_i^{(j)} \\ \textbf{or} \bigwedge\limits_{\{i\in P \mid \iota(i)=n+1\}} t_i \leq s \end{cases}$$

Then we can re-arrange the subtyping proposition as emptiness checks:

$$\left(\exists j \in [1,n].\ s_j \wedge \bigwedge_{\{i\in P \mid \iota(i)=j\}} s_i^{(j)} \leq \mathbb{0}\right) \textbf{ or } \left((\neg\, s) \wedge \bigwedge_{\{i\in P \mid \iota(i)=n+1\}} t_i \leq \mathbb{0}\right)$$

Exploring the domain space ι is now equivalent to distributing intersections over $(n+1)$ types, and checking whenever one becomes empty. The values of those initial types being $s_1,..,s_n$ and $\neg\, s$; we have just described the algorithm Φ_n. □

G Semantic subtyping: strong arrows

G.1 Set Semantics

Definition G.1. Let X be a subset of D.

- $\mathsf{dom}(X) = \{d : D \mid \forall R : X.\ (d,\Omega) \notin R\}$
- $\mathsf{cod}(X) = \{d' : D \mid (\mathsf{dom}\,(X) = \varnothing) \vee (\exists R : X.\ \exists d : \mathsf{dom}(X).\ (d,d') : R)\}$
- $X^\star = \begin{cases} \varnothing & \text{if } X \not\subseteq \mathcal{P}_f\,(D \times D_\omega) \\ X \cap \mathcal{P}_f\,(D \times (\mathsf{cod}(X) \cup \{\omega\})) & \text{otherwise} \end{cases}$

We want to adapt the algorithm for deciding subtyping for strong arrows. According to the previous section, this means finding an algorithm to decide the containment problem:

$$\bigwedge_{i\in I}(t_i \to s_i) \wedge \bigwedge_{j\in P}(t_j \to s_j)^\star \wedge \bigwedge_{k\in R} \neg(t_k \to s_k) \wedge \bigwedge_{l\in Q} \neg(t_l \to s_l)^\star \quad \leq \quad \mathbb{0}$$

The introduction of strong arrows, compared to Alain Frisch's thesis [24], requires the new following set of lemmas to be able to decide the subtyping problem.

Lemma G.1. With I finite, this lemma is used to simplify intersections of strong arrows:

$$\bigcap_{i\in I}(X_i \to Y_i)^\star \quad = \quad \left(\bigcup_{i\in I} X_i \to \bigcap_{i\in I} Y_i\right)^\star \tag{10}$$

Proof. Proving both inclusions.

(1) Suppose $R \in (\bigcup_{i\in I}(X_i) \to \bigcap_{i\in I}(Y_i))^\star$. Let $(d,\delta)\in R$. Let $i\in I$.
   - if $d\in X_i$, then $d\in\bigcup_{i\in I}(X_i)$ so $\delta\in\bigcap_{i\in I}(Y_i)\subseteq Y_i$.
   - if $d\notin X_i$, then by definition $\delta\in\bigcap_{i\in I}(Y_i\cup\{\Omega\})\subseteq Y_i\cup\{\Omega\}$.

(2) Now, suppose $R\in\bigcap_{i\in I}\left((X_i\to Y_i)^\star\right)$. Let $(d,\delta)\in R$.
   - if $d\in\bigcup_{i\in I}(X_i)$. For all $i\in I$, either $d\in X_i$, thus $\delta\in Y_i$, or $d\notin X_i$, thus $\delta\in Y_i\cup\{\Omega\}$. Since there exists at least one i_0 such that $d\in X_{i_0}$ (which does not contain Ω, then we have proven that $\delta\in\bigcap_{i\in I}(Y_i)$.
   - if $d\notin\bigcup_{i\in I}(X_i)$, by the same reasoning, except it's not certain Ω can be subtracted, we have $\delta\in\bigcap_{i\in I}(Y_i\cup\{\Omega\})$.

□

Lemma G.2.

$$\bigcup_{i\in I}(X_i\to Y_i)\cap(X\to Y)^\star\subseteq(W\to Z)\iff\begin{cases}W\subseteq\bigcup_{i\in I}X_i\cup X\\ \forall J\subseteq I.\ \left(W\subseteq\bigcup_{j\in J}X_j\right)\vee\left(\bigcap_{j\in I\setminus J}Y_j\cap Y\subseteq Z\right)\end{cases}$$

Proof. Using Theorems (4.7) and (4.8) from [24], proof is similar to the one used to derive subtyping for single-arity arrows (see [24] p.73 Lemma 4.9). □

Lemma G.3.

$$\begin{aligned}&\bigcap_{i\in I}(X_i\to Y_i)\cap(X\to Y)^\star\subseteq\mathcal{P}_f\,(D{\times}Z\cup\{\Omega\})\\ \iff&\left(\bigcap_{i\in I}Y_i\cap Y\subseteq Z\right)\wedge\left(\forall\, J\overset{J\neq\emptyset}{\subseteq}I\,.\,\left(D\subseteq\bigcup_{j\in J}X_j\right)\vee\left(\bigcap_{j\in I\setminus J}Y_j\cap Y\subseteq Z\right)\right)\end{aligned}$$

Proof. Using Theorems (4.7) and (4.8) from [24], proof is similar to the one used to derive subtyping for single-arity arrows (see [24] p.73 Lemma 4.9). □

Remark: This lemma uses the set Z to represent the codomain of some function $W\to Z'$. In the case where $W=\emptyset$, then Z should be D for any value of Z'. Hence why the restriction to $a\neq\mathbb{0}$ in Lemma (G.5).

Lemma G.4.

$$\bigwedge_{i\in I}(t_i\to s_i)\wedge(c\to d)^\star\leq(a\to b)\iff\begin{cases}a\leq\bigvee_{i\in I}t_i\vee c\\ \forall J\subseteq I.\quad\left(a\leq\bigvee_{j\in J}t_j\right)\vee\left(\bigwedge_{j\in I\setminus J}s_j\wedge d\leq b\right)\end{cases}$$

Proof. By application of Lemma (G.2). □

Lemma G.5. If $a\neq\mathbb{0}$, then

$$\bigwedge_{i\in I}(t_i\to s_i)\wedge(c\to d)^\star\leq a\to b^\star\iff\begin{cases}a\leq\bigvee_{i\in I}t_i\vee c\\ \forall J\subseteq I.\ \left(\bigvee_{j\in J}t_j=\mathbb{1}\right)\vee\left(\bigwedge_{j\in I\setminus J}s_j\wedge d\leq b\right)\end{cases}$$

Proof. By application of Lemmas (G.2) and (G.3). □

Lemma G.6.

$$\bigcap_{i\in I}\mathcal{P}(X_i) \subseteq \bigcup_{i\in P}\mathcal{P}(Y_i) \cup \bigcup_{i\in Q}(\mathcal{P}(Z_i)\cap\mathcal{P}(W_i))$$

$$\iff \left(\exists i_0\in P.\ \bigcap_{i\in I} X_i \subseteq Y_{i_0}\right) \vee \left(\exists i_0 \in Q.\ \bigcap_{i\in I} X_i \subseteq Z_{i_0}\cap W_{i_0}\right)$$

Proof. Corollary of Lemmas G.2 and G.3 □

Examples

- $(t \to s) \wedge (c \to d)^\star \leq (a \to b)^\star$ where $a \neq \mathbb{0}$. This case raises the condition

$$\begin{cases} (a \leq t \vee c) \wedge (s \wedge d \leq b) \\ (t = \mathbb{1}) \vee (d \leq b) \end{cases}$$

G.2 Subtyping Algorithm

With $t = \bigwedge_{i\in P} u_i$ and $s = \bigvee_{i\in P} w_i$, using Lemma G.1, we rewrite the containment problem:

$$\textstyle\bigwedge_{i\in I}(t_i \to s_i) \wedge \bigwedge_{i\in P}(u_i \to w_i)^\star \wedge \bigwedge_{i\in R}\neg(t_i \to s_i) \wedge \bigwedge_{i\in Q}\neg(a_i \to b_i)^\star \leq \mathbb{0}$$

$$\iff \textstyle\bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \leq \bigvee_{i\in R}(t_i \to s_i) \vee \bigvee_{i\in Q}(a_i \to b_i)^\star$$

$$\overset{G.6}{\iff} \textbf{ or } \begin{cases} \exists i_0 \in R.\ \bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \leq (t_{i_0} \to s_{i_0}) \\ \exists i_0 \in Q.\ \bigwedge_{i\in I}(t_i \to s_i) \wedge (t \to s)^\star \leq (a_{i_0} \to b_{i_0})^\star \end{cases}$$

This last equivalence can now be solved with algorithms derived from Lemmas (G.4) and (G.5).

H Extension: Parameterized Strong Types

A strong function is one that behaves normally on its domain, and outside of it either outputs value from its codomain or errors on an explicit VM check. In some cases, it is possible to refine this definition, and parametrize the return type outside of the domain, as long as it is a subtype of the codomain. For example, the function

```
def f(x) when is_integer(x), do: x+1
def f(x) when is_boolean(x), do: not x
def f(x), do: 42
```

clearly always outputs an integer outside of its domain. It has a strong type

$$((\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}))^\star$$

and if we parameterize it we can write it has the more precise parameterized strong type:

$$((\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}))^{\star(\texttt{int})}$$

Thanks to such a parametrization we can deduce the type `int` for the application of this function to an argument of type $\neg(\texttt{int} \vee \texttt{bool})$.

Inference for these types is already supported by the way strong types are checked; instead of inferring the codomain with the system in Figure 5, inferring any subtype of the codomain will ensure that the function is strong for this subtype. In particular, a function that is strong for type $\mathbb{0}$ is one that *fails on a runtime check* on every input outside its domain. E.g. function

```
def f(x) when is_integer(x), do: x+1
def f(x) when is_boolean(x), do: not x
```

has the parameterized strong type $((\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}))^{\star(\mathbb{O})}$

This definition is equivalent to the one presented earlier in the paper, assuming `integer()` and `boolean()` are the only basic types used in FW-Elixir.

In Section A of the main text, we formally demonstrate that negations of guards can be compiled out during the translation of patterns and guards. This translation process ensures that FW-Elixir expressions can be properly represented in Core Elixir, which forms the basis of our type system for Elixir.

For space reasons, some proofs and parts of the system for guard analysis have been omitted from the main text. These can be found in the full version of this paper.