# Union and intersection contracts are hard, actually

Anonymous Author(s)

## Abstract

Union and intersection types are a staple of gradually typed language such as TypeScript. While it's long been recognized that union and intersection types are difficult to verify statically, it may appear at first that the dynamic part of gradual typing is actually pretty simple.

It turns out however, that in presence of higher-order contracts union and intersection are deceptively difficult. The literature on higher-order contracts with union and intersection, while keenly aware of the fact, doesn't really explain why. We point and illustrate the problems and tradeoffs inherent to union and intersection contracts, via example and a survey of the literature.

***CCS Concepts:*** • **General and reference** → *Surveys and overviews*; • **Software and its engineering** → **Language features**; *Software verification and validation.*

***Keywords:*** contracts, higher-order contracts, union, intersection

**ACM Reference Format:**

Anonymous Author(s). 2021. Union and intersection contracts are hard, actually. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Union types, meaning a type `A ∪ B` containing values which belong either to a type `A` or `B`, are a popular tool when adding static types to a dynamic language. In particular, both TypeScript [9] and MyPy [7], use union types to model the frequent practice to use the value **`null`** (`None` in Python) to represent an absent optional value. This is why the gradual typing literature, concerned with formalising the interplay between static and dynamic type systems, has been quite interested in union types [11, 15, 18, 21, 23].

On the other hand, unions are not a common feature of static type systems, mostly because they are quite difficult to verify statically. So unions are really only worth it in gradually typed language where they formalise existing dynamically typed patterns. On the other hand, surely, for dynamic tests, unions are really easy: it is simply the Boolean disjunctions of two tests.

Unfortunately, as we document in this article, as soon as you extend dynamic checks to *contracts* [12], unions become actually pretty difficult, and threaten desirable properties of your language.

### 1.1 Configuration languages

To motivate contracts and the problem caused by unions, let's make a detour through configuration languages. A configuration language is a language concerned with describing the configuration of an application. In traditional configuration languages, such as YAML, TOML, or JSON, the configuration is fully, and explicitly, spelt out.

However, with the advent of DevOps, configurations have been extended to describe the entire state of a computer, or even a fleet of computers. For instance, with Kubernetes you need to configure a large fleet of (possibly replicated) docker containers. To describe this sort of configurations, you really want to be able to re-use and abstract parts of the configuration, like traditional programming languages do. To meet this need, languages such as Cue [2], Dhall [4], Jsonnet [5], or Nickel [6], where configurations are generated rather than spelt out, were created.

Another example is continuous integration systems: it's fairly typical to need a matrix of jobs, wherein the same tests are run on different infrastructures, or with different versions of a compiler. Traditional configuration would have you copy the same steps for each infrastructure. This is tedious, hard to maintain, and error prone. It's much better, instead, to write the steps once, and instantiate them for each infrastructure. Continuous integration systems typically do this using a templating system layered on top of YAML. Each of the configuration-generating languages above allow such job-matrix definition natively.

### 1.2 Nickel

In this article, we will use the Nickel language [6] as illustration and motivation. At its core, Nickel is the JSON data model, augmented with abstraction mechanisms, and its conformed of:

- dictionaries, written as:

```
{field1 = value1, …, fieldn = valuen}
```

- arrays:

```
[x1, x2, …, xn]
```



*Conference'17, July 2017, Washington, DC, USA*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

- functions:

  ```
  fun arg1 … argn ⇒ body
  ```

- and let-definitions:

  ```
  let id = value in exp
  ```

A Nickel configuration is then evaluated to an explicit configuration, *e.g.* in JSON, which can then be consumed by an application. Therefore a design constraint of Nickel is any Nickel data must have a straightforward interpretation in JSON.

### 1.3 Contracts

A useful feature of a configuration language is to provide facilities for schema validation. That is, help answer questions like: does our configuration have all the required fields? does the `url` field indeed contains a URL?

These are inherently dynamic questions, as they are all questions about the evaluated configuration. To this effect, Nickel lets us annotate any expression with a dynamic schema check: `exp | C`. There is also syntactic sugar to annotate definitions: `let id | C = value in exp` stands for `let id = ( value | C) in exp`.

Let us pause for a moment and consider the following: it is Nickel's ambition to be able to manipulate configurations like Nixpkgs. With over 50 000 packages, it is one of the largest repository of software packages in existence [8]. Concretely, Nixpkgs is a dictionary mapping packages to build recipes. That is, a massive, over-50 000-key-value-pair wide dictionary. It is absolutely out of the question to evaluate the entirety of this dictionary every time one needs to install 10 new packages: this would result in a painfully slow experience.

To be able to support such large dictionaries, Nickel's dictionaries are *lazy*, that is, the values are only evaluated when explicitly required. For instance, when writing `nixpkgs.hello`, only the `hello` package gets evaluated.

But let's consider now writing something like `nixpkgs | packages`, to guarantee that all the packages conform to the desired schema. If this were a simple Boolean test, it would have to evaluate all 50 000 package to check their validity, hence breaking the laziness of dictionaries. Do we have to choose between laziness and schema validation? Fortunately, we don't! Enter *contracts* [12]: dynamic checks which can be partially delayed, yet errors can be reported accurately. Contracts can respect laziness of dictionaries, and they can be ~~use~~ used to add schema validation to functions as well (in fact functions were the original motivation for contracts).

There is no Boolean function which can check that a value has type `Str → Str`. Instead, a contract for `Str → Str` checks for each call of the function whether

1. the argument has type `Str`, otherwise the caller of the function is faulty
2. if so, that the returned value has type `Str`, otherwise the implementation of the function is faulty

Like in the case of lazy dictionaries, the checks are delayed. Contracts keep track of whether the caller or the implementation is at fault for a violation, hence it can report precise error messages. Contracts are said to *blame* either the caller or the implementation. Compare Figure 1a and Figure 1b: in Figure 1a an error is reported inside the `catHosts` function, but `catHosts` is, in fact, correct, as is made clear by Figure 1b, where `catHosts` is decorated with the `Str → Str` contract, and correctly reports that the caller failed to call `catHosts` with a string argument.

Unfortunately, the delayed check of contract, while essential to ensuring that schema validation doesn't affect performance (or, indeed, is possible at all on functions), make union contracts (and their less appreciated sibling, intersection contracts) quite problematic.

### 1.4 Contributions

Our contributions are as follows

- We describe the fundamental difficulties caused by presence of union and intersection contracts in a language, which are kept implicit in the literature (Section 4)
- We survey the various trade-offs which appear in implemented languages and in the literature to work around these difficulties (Section 5)

## 2 A typology of language features

Union contracts are not only difficult to implement, their unrestricted presence is incompatible with potentially desirable properties of the language. In this section we present some of these properties; we will show how these properties interact with union contracts in Sections 4 and 5.

### 2.1 User-defined contracts

A strength of dynamic checking is that we can easily check properties which are impractical to check statically. For instance that a string represents a well-formed URL, or a number is a valid port.

This same property is desirable of contracts as well, otherwise we lose an important benefit of dynamic checking. Preferably, we want to be able to extend the universe of contracts with user-defined predicates.

For instance, Figure 2 shows the definition of a contract for valid ports in Nickel syntax. User-defined contracts can be combined with other contracts normally: `Int → Port` is a contract verified by functions which, given an integer returns a valid port.

This type of contracts are present in many different languages, for instance, the Eiffel programming language [17], the precursor of the Design by Contract philosophy, makes it possible to assert these kinds of expression as pre- and post-conditions on functions and as invariants on classes [3].

```
let catHosts = fun last ⇒
  let hosts = ["foo.com", "bar.org"] in
  lists.fold (fun val acc => val ++ "," ++ acc) hosts last in
  lists.fold (fun val acc =>
    val ++ "," ++ acc) hosts last in

let makeHost = fun server ext ⇒
  server ++ "." ++ ext in

catHosts (makeHost "google")
```

```
error: Type error
3 | […] "," ++ acc) hosts last in
  |             ^^^
%  | This expression has type Fun,
  | but Str was expected
4 |
5 | let makeHost = fun server ext ⇒ […] in
  |                --------------------
  |                evaluated to this
  = ++, 2nd argument
```

**(a)** Error reporting without contract

```
let catHosts | Str → Str = fun last ⇒
[…]
$  let hosts = ["foo.com", "bar.org"] in
  lists.fold (fun val acc =>
    val ++ "," ++ acc) hosts last in
%DIF >
let makeHost = fun server ext =>
  server ++ "." ++ ext in
%DIF >
catHosts (makeHost "google")$
```

```
error: Blame error: contract broken by the caller.
  |  Str → Str
  |  --- expected type of the argument […]
[…]
1 | let catHosts | Str → Str = fun last ⇒
  |                ^^^^^^^^^^ bound here
[…]
6 | catHosts (makeHost "google")
  | --------------------------- (2) calling <func>
```

**(b)** Error reporting with contract

**Figure 1.** Contracts improve error messages

```
let Port = contracts.fromPred (fun p ⇒
  num.isInt p && 0 ≤ p && p ≤ 65535) in
80 | Port
```

**Figure 2.** A contract for valid ports

The Racket programming language also has a system to work with contracts, powerful enough to define flat contracts, and to compose them with other kinds of dynamic checks, like higher order contracts or a lightweight take on union and intersection contracts [1].

### 2.2 Referential transparency

The performance of modern programs heavily relies on the optimizations performed by the compiler or the interpreter. Even more so for functional languages, whose execution model is often far removed from the hardware, causing naive execution to exhibit unacceptable slowdowns.

One such important optimization is inlining (Figure 3). Functional programs tend to make heavy use of functions, and a function call is not a free operation: it usually involves a number of low-level operations such as saving and storing registers, pushing a new stack frame and jumping to and back from the function's body. Inlining eliminates a function call by directly substituting the function for its definition at compile time (or before execution, for interpreted language). This is especially efficient for small functions that are called repeatedly.

```
let elem = fun elt ⇒
  lists.any (fun x ⇒ x == elt) in

let subList = fun l1 l2 ⇒
  elem (lists.head l1) l2
  && subList (list.tail l1) l2
```

Source program

```
let subList = fun l1 l2 ⇒
  lists.any (fun x ⇒ x == (lists.head l1))
    l2
  && subList (list.tail l1) l2
```

Optimized program

**Figure 3.** Inlining

While inlining expands an expression by substituting a definition for its value, an opposite transformation can be beneficial when a composite expression occurs at several places. In this case, the same expression is wastefully recomputed at each occurrence. Common subexpression elimination (CSE) consists in storing the expression in a variable that is then used in place of the original occurrences (Figure 4), thus evaluating the expression once and for all.

Beyond CSE, other optimizations such as loop-invariant code motion or let-floating [19] apply the same principle of

```
let elemAtOrLast = fun index list ⇒
  if index > lists.length list - 1 then
    lists.elemAt (lists.length list - 1)
    list
  else
    lists.elemAt index list
```

Source program

```
let elemAtOrLast = fun index list ⇒
  let l = lists.length list - 1 in
  if index > l then
    lists.elemAt l list
  else
    lists.elemAt index list
```

Optimized program

**Figure 4.** Common subexpression elimination

```
let f = fun x ⇒ g y (x + 1)
```

Source program

```
let g' = g y in
let f' = fun x ⇒ g' (x + 1)
```

Optimized program

**Figure 5.** Let-floating

extracting out an invariant expression to avoid recomputing it (respectively across loop iterations and function calls).

For example, take the code of Figure 5. The partial application `g y` is recomputed each time `f` is called. This may be costly, in particular in the presence of contracts: if the first argument of `g` must be a list with elements of a specific kind and if that precondition is enforced by a function contract (say `g | List Odd → Even → List Odd`), the additional cost is linear in the size of `y`. A sensible thing to do is to factor `g y` out of `f` as in Figure 5, which is something a let-floating transformation could indeed do (given `g` is pure, as detailed below).

The soundness of these optimizations is tied to the validity of specific program equivalences. Inlining requires that one can replace the application of a function by its body, which is basically $\beta$-reduction: as long as the arguments are evaluated following the language's strategy, this is usually a valid transformation. However, Section 4 exposes that the question of inlining a function with a contract attached is more subtle.

A CSE-like transformation on a term $M$ requires on the other hand an equivalence of the form:

$$M[N/x] \simeq let\ x\ =\ N\ in\ M \qquad (1)$$

$M[N/x]$ stands for the substitution of $x$ for the term $N$ in the term $M$. This equation clearly fails in presence of side-effects, as demonstrated in Figure 6. In that example, `(f 1, f 1)` prints `"hi"` two times while `let y = f 1 in (y,y)` only prints it once.

```
let f x = print "hi";(x+1)
```

Effectful function

```
(f 1,f 1)  ≉  let y = f 1 in (y,y)
```

Invalid expansion

**Figure 6.** Counter-example to (1) in presence of side-effects

However, (1) does hold for *pure* terms, that are terms without side-effects. In a pure language, all these transformations are valid. In impure languages, the situation varies: in some case a large subset of pure terms can be identified (in languages with effects tracking such as PureScript) to be safely transformed. Otherwise, the compiler must stay conservative and only apply CSE to expressions it can prove are without side-effects (arithmetic expressions, for example).

Higher-order contracts described in Section 1.3 already introduce side-effects: semantically, the `blame` operator signaling contract violation is equivalent to non-termination (error reporting aside). But non-termination is a special kind of effect, if only because it preserves referential transparency. In a lazy language like Nickel, (1) still holds for all terms in the presence of non-termination. In a strict language (call-by-value), (1) fails in all generality because the transformation may cause a previously unevaluated term to be suddenly evaluated. However, (1) holds given slight restrictions on the context $M$: for example, it holds as long as at least one occurrence of $x$ in $M$ is in scope (i.e. not inside a function). Beside, many languages out there (including Nickel) already feature non-termination. For such languages, adding contracts does not change what optimizations are valid.

Strikingly, we will see in Section 4 that the introduction of unions and intersections *breaks referential transparency*. Unions and intersections make (1) unsound ~~*even in a pure setting*~~even in a lazy language, preventing the kind optimization of Figure 5 to fire in general. In a strict language, the restrictions to (1) outlined above are not sufficient anymore. If the effect of plain higher-order contracts behaves as non-termination, the effect of union and intersection contracts rather behaves as shared state.

```
interface Circle {
  kind: "circle";
  radius: number; }

interface Square {
  kind: "square";
  sideLength: number; }

type Shape = Circle | Square;
```

**Figure 7.** A sum type as a tagged union

## 3 Union & intersection

Let us now consider union and intersection contracts, before we explain ~~,~~in Section 4 how they can compromise the properties that we described in Section 2.

### 3.1 Unions

A union type `A ∪ B` is a type of values which are either of type `A` or of type `B`: literally the union of `A` and `B`. Union types are popular in gradual typed systems such as TypeScript [9] and MyPy [7].

***In gradually typed systems.*** The problem that these practical gradual type systems are trying to solve is to capture, in static types, as many programming patterns as possible from the underlying dynamically typed language (JavaScript for TypeScript and Python for MyPy). One such pattern is heterogeneous collections. For instance, in TypeScript, an array which can contain both strings and numbers would have type `Array<string|number>`.

A probably even more common pattern is a variable which can contain either a value of type `A` (say, a number) or the **`null`** value. So much so, in fact, that MyPy defines a type alias `Optional[A]` for `Union[A,None]` (`None` is how Python renders the null value).

Yet another application of union types is, rather than capturing a pattern from JavaScript or Python, to capture a pattern from traditional statically typed language: sum types. In statically typed languages, values of sum types are usually thought of as being built out of constructors. But neither JavaScript nor Python have such constructors. So instead, sums are construed as "tagged unions" (or discriminated unions), that is, quite literally, the union of two types which contain a discriminating tag. See Figure 7 for an example from the TypeScript documentation: there the `kind` field is the tag, and its type in both alternatives is a singleton type which contains only the specified string.

***Union contracts.*** In the academic gradual type literature, it is common to use contracts as a glue between static and dynamic types. Therefore, the question of bringing union to contracts is natural, and have indeed been studied (*e.g.* [15, 23]).

Like for static types, a value which satisfies the contract `A ∪ B` is a value which satisfies either contract `A` or contract `B` (though in Section 5 we will see that it may be desirable to weaken this definition).

Nickel is a language built from scratch with contracts, so it may be less clear why unions are useful. However, Nickel's ambition is to have its data model canonically interpretable in common serialization formats, in particular JSON. It means that it is very convenient to represent optional value by the **`null`** value like in JavaScript. It also means that Nickel doesn't have built-in constructors: constructors don't have a canonical representation in JSON. So it would be quite natural to represent optional contracts and sum contracts as unions.

### 3.2 Intersections

An intersection type `A ∩ B` is satisfied by values which satisfy both contract `A` and contract `B`.

Intersection contracts (and types) are probably less prevalent than union in practical type systems. However, a function from a union is equivalent to an intersection. That is `(A ∪ B) → C` ≃ `(A → C) ∩ (B → C)`. So in a system with functions and unions, intersections are already morally present (and, for that matter, in a system with functions and intersections, unions are morally present). Some of our examples in Sections 4 and 5 are better expressed in terms of intersections, so it's best to include them.

Figure 8 gives a concrete example of this phenomenon. The function `appendDate` ~~append~~ appends an element to `list` whose type is only known to be in union of two lists, each using a different representation. Because the return type is the same as the input type, `appendDate` must preserve this representation (`Date` and `DateWeek` cannot be mixed in a same list). Both alternatives support falling back to a simple string for unparsed dates. Faced with these two possibilities, `appendDate` can only append a value which fits both types: this is precisely the intersection `(Date ∪ Str) ∩ (DateWeek ∪ Str)`, that is, `Str`.

```
\DIFaddendFL let Date = {day | Num, month |
    Num, year | Num} in
let DateWeek = {dayOfWeek | Num, week | Num,
     year | Num} in
let appendDate | (List (Date ∪ Str)
                 ∪ List (DateWeek ∪ Str))
                 -> List =
  fun list => lists.cons list "01/01/2021"
}
                 -> (List (Date @| Str)
                 @| List (DateWeek @| Str)) =
  fun list => lists.cons "01/01/2021" list
```

**Figure 8.** Adding an element to a union of two arrays

Yet, intersections are useful in their own right. They can be used to combine dictionaries in the style of object-oriented multiple inheritance. For instance, in Figure 9 two types are defined `Animal` and `Pet`, and a variable that is compatible with both types is declared, with type `Animal ∩ Pet`. This particular application is supported, for instance, by TypeScript.

```
let Animal =
  { species | Str, breed | Str, name | Str }
   in
let Pet = { owner | Str, name | Str } in
let myDog | Animal ∩ Pet =
  { species = "Canis Lupus",
    breed   = "Australian Cattle Dog",
    owner   = "Anonymous Author",
    name    = "Juno" }
```

**Figure 9.** An animal that is also a pet

Another application of intersections shows up when intersecting functions: it can be used to encode overloading. For instance, take a look at Figure 10, where the function `duplicate` works both as a function to duplicate arrays, as well as a function to duplicate strings. This is particularly useful when using unions, since it's a good way to express that a function can deal with different shapes of data. For instance, in the same figure, `duplicate` is (correctly) called on a value of type `(List Str) ∪ Str`.

```
let duplicate
    | (List Str → List Str)
      ∩ (Str → Str) =
  fun x ⇒ x ++ x in
let text | (List Str) ∪ Str = … in
duplicate text
```

**Figure 10.** Duplicating an array of Strings or a String

## 4 Incompatibilities

However appealing union and intersection contracts may be, they happen to be either hard to combine or even fundamentally incompatible with the desirable language features from Section 2. At least in their full-blown form: in Section 5 we will discuss pragmatic restrictions of union and intersection contracts to recover some or all of the features.

### 4.1 Union contracts as a side-effect

In Nickel, the failure of a function contract can always be traced back to a single call. For example, take the function `f` with a simple contract attached of Figure 11. The whole program fails with a contract error blaming `f` because the return value of the second call `f 5` violates the `Positive` contract. The first call to `f` does not matter, and `f 5` is a single and independent witness of the contract violation. The user is pointed to this one location in practice.

This single witness property can be justified as follows. Apart from the error reporting part (although this is the crucial bit in practice!), the current contract system of Nickel can be implemented purely as a library, requiring only a `fail` primitive to abort the execution. In practice, applying a function contract to `f` replaces it with a `f'` that performs the additional checks. Thus, since the core language is pure (albeit partial, if only because `fail`), the failure of `f' 5` must be independent of its environment and of any previous call to `f'`.

```
let f | Positive → Positive
      = fun x ⇒ x - 7 in
(f 10) + (f 5)
```

**Figure 11.** Simple contract violation

Union contracts are different. Consider the program presented in Figure 12. The same `f` is now given a union contract. `f` is violating this contract once again, as it neither maps positive numbers to positive numbers nor positive numbers to negative numbers.

```
let f | (Positive → Positive)
        ∪ (Positive → NonPositive)
      = fun x ⇒ x - 7 in
(f 10) + (f 5)
```

**Figure 12.** Union contract violation

This program must fail, because `f 10` is a witness of `f` failing the contract `Positive → NonPositive`, and `f 5` is a witness of `f` failing `Positive → Positive`. But, as opposed to the example from Figure 11, removing only one of the calls makes the program succeed! Indeed, each call only unveils the violation of one component of the union. In this example, a single call to `f` that would be the witness of the violation of the whole contract doesn't even exist: a minimum of two are always needed.

This behavior indicates that union contracts introduce side-effects. The result of `f 5` now depends on the previous execution and more specifically on any prior call to `f`. This behavior of union contracts ~~break~~ breaks referential transparency, as well as the property 1 introduced in Section 2.2, that is required to perform CSE-like optimizations~~in all generality. The candidate example of Figure 5 in Section 2.2 can't be optimized in general~~.

Figure 13 illustrates this point further. It contains an original program and an optimized version where the common subexpression `f 1` has been eliminated. While equivalent in a pure language ~~without~~ with only non-termination or plain higher-order contracts, these two programs behave differently because of unions:

- The original version returns `(1, "False")` without failing.
- The optimized version fails with a contract violation.

In the original version, each partial application `f 1` gives rise to a fresh instance of the contract `Bool → Num ∪ Bool → Str`. These instances are independent, and can pick a different component of the union to satisfy. Although `f` doesn't actually respect the contract, these calls are not enough to prove so. In the optimized version, `g` is endowed with a single contract, that must pick one of the two components of the union. There, the two calls refer to the same union contract, and shows that `f` does violate its initial contract.

```
let f | Num → (Bool → Num ∪ Bool → Str)
      = fun x y ⇒ if y then x else "False"
in (f 1 true, f 1 false)
```

Original

```
let f | Num → (Bool → Num ∪ Bool → Str)
      = fun x y ⇒ if y then x else "False"
let g = f 1 in
(g true, g false)
```

Optimized

**Figure 13.** Equivalent programs ~~,~~ with ~~inlining or~~ CSE applied

To sum up, the addition of union contracts introduce side-effects in a pure language. Side-effects have well-known pitfalls:

- For the programmer, they are hard to reason about. They prevent local reasoning. In our previous examples, removing or adding a function call somewhere can toggle a failure in a call at a totally different location.
- For the interpreter (or compiler), side-effects inhibit many optimizations and program transformations.

### 4.2 Intersection with user-defined contracts

A natural ~~-but naive --~~ — but naive — implementation of intersection contracts could be the following: to apply a contract `A ∩ B`, apply both contracts `A` and `B` sequentially, resulting in the naive decomposition rule of Figure 14.

This intuition works for simple contracts: checking that `x | Natural ∩ Odd` amounts to check that `x | Natural` and `x | Odd`. Unfortunately, this doesn't scale to higher-order

```
M | A ∩ B  ≃  (M | A) | B
```

Naive decomposition

```
(A → B) ∩ (C → D)  ≃  (A ∩ C) → (B ∩ D)
```

Exchange law

```
let g | Num → Num ∩ Str → Str
      = fun x ⇒ x in
g 1
```

Overloaded identity

**Figure 14.** Naive implementation of intersection

contracts. The overloaded identity example of Figure 14 illustrates the use of an intersection to model a simple overloading of the identity function. If we were to apply the naive decomposition, the argument `1` would fail the `Str → Str` contract and abort the execution. Perhaps the exchange rule given in Figure 14, which is a direct consequence of the naive decomposition, illustrates the issue better. It is clear that this exchange law isn't the right semantics for overloading. With this law, the contract for overloaded identity of Figure 14 would always fail because no argument can satisfy `Num ∩ Str`.

In a higher-order intersection contract, blame is raised when:

**Faulty caller** The argument fails *both* components.

**Faulty implementation** The function fails at least *one* component that the argument previously satisfied.

To fix the naive implementation, the interpreter can share state between the sub-contracts, in order to decide if blame must be raised or not when a sub-contract fails:

```
x | A ∩ B  ≃  (x | A[l]) | B[l]
```

Shared state is represented by the label `l`. This is in essence the approach proposed by Williams, Morris, and Wadler in [23].

```
let C = contracts.fromPred (fun f =>
  f 0 == 0) in
let g | (Str -> Str) @& C
      = fun x => x
in g 0
```

**Figure 15.** Intersection and user defined contracts

However, their approach has a major drawback: it is not straightforwardly compatible with user-defined contracts (introduced in Section 2.1). The issue is similar to our initial issue with higher-order contracts and the naive decomposition: user-defined contracts may apply functions and thus make a sub-contract of the intersection fail, but this failure shouldn't always result in raising blame. An example is given on Figure 15. Decomposing using the shared state approach, we end up with:

```
((fun x ⇒ x) | (Str → Str)[l]) | C[l]
```

Stateful decomposition

where `l` represents the shared state. At this point, applying the `C` contract results in evaluating:

```
((fun x ⇒ x) | (Str → Str)[l]) 0 == 0.
```

Applying a function wrapped in a `Str → Str` contract to `0` fails negatively. This is not the expected behavior, since the identity function does respect semantically both contracts. As opposed to built-in higher-order contracts, user-defined contracts are black-box from the interpreter's point of view, and it is thus not obvious how to extend the shared state approach to handle user-defined contracts.

```
~~let c = contracts.fromPred (fun f =>~~
~~  f 0 == 0) in~~
~~let g | (Str -> Str) @& c~~
~~        = fun x => x~~
~~in g 0~~
```

~~intersection and user defined contracts~~

Once again, intersection contracts introduce side-effects in the picture. What's more, these side-effects interact with user-defined contracts in a non-trivial way, while they are an important feature for validation.

## 5 Pragmatic trade-offs

Despite the difficulties of Section 4, union and intersection contracts are still sought after. In this section we turn to existing systems with union and intersection contracts in the literature and in implementations.

These systems all make trade-offs, sacrificing some features of union and intersection contracts to preserve language features. We survey and discuss those trade-offs and their implications.

### 5.1 A coinductive semantics

In order to give a precise definition to what values ought to satisfy union and intersection contracts, [15] give a coinductively defined semantics inspired by union and intersection type systems. The key innovation of [15] is recognizing that giving a semantics to higher-order contracts requires defining not only what values satisfy a contract, but also what *contexts* satisfy the contract. This models the situation where context may violate a contract by calling a function with an inappropriate argument.

Concretely, given a contract $C$, [15] introduce the two sets $[\![C]\!]^+$ and $[\![C]\!]^-$ of values and contexts, respectively, satisfying the contract. They are defined by mutual induction and coinduction.

This semantics has limited support for overloading. Consider the example in Figure 16: it could evaluate to the pair `(1, 1)`. But the coinductive semantics of [15] rejects it as a contract violation. This can be phrased pithily as the fact that the coinductive semantics doesn't validate the property `A → B ∩ A → C≃A → (B ∩ C)`.

```
let f = fun x y ⇒ x in
let g = f | (Num → Num → Num)
          ∩ (Num → Bool → Num) in
let h = g 1 in
(h 1, h true)
```

**Figure 16.** Intersection contracts don't distribute

Note that if the last two lines of Figure 16 had read `(g 1 1, g 1 true)` instead, then the coinductive semantics would accept the example. It implies that under the coinductive semantics, common-subexpression elimination (see Section 2.2) is quite perilous.

### 5.2 A first realization

In Section 4, we've seen that different calls to a function with a union contract must share information: the behavior of one call is influenced by the previous ones, as the function must pick one component of the union to satisfy across all usages. Conversely, following the semantics of overloading, each application of a function with an intersection contract can select a different branch and is thus independent from the others. A general contract composed of nested unions, intersections and higher-order contracts appears to require complex book-keeping in order to correctly raise blame.

All of this still holds true of the coinductive semantics described in Section 5.1. Nevertheless, [15] gives an algorithmic system which is complete for their coinductive semantics. In a remarkable technical tour de force, their algorithmic system is compatible with user-defined ~~contract~~ contracts (see Section 4.2).

A key aspect of the approach of [15] is to rewrite nested union and intersection contracts into a disjunctive normal form using the De Morgan's law `A ∩ (B ∪ C)≃(A ∩ B) ∪ (A ∩ C)`. The goal is to be able to delay the choice of branch in intersections as much as possible.

To implement contract verification, [15] resorts to specific reduction rules for unions and intersections which perform this rewriting on the fly. This aspect is critiqued in [23]: "the monitoring semantics for contracts of intersection and union types given by Keil and Thiemann are not uniform. (...) If uniformity helps composition, then special cases can hinder composition."

A cost of this approach is that the De Morgan's law `A ∩ (B ∪ C)≃(A ∩ B) ∪ (A ∩ C)` duplicates contract `A`, which will cause some contracts to be checked several times. This can be an issue with user-defined contracts which may include costly tests.

Efficiency is also affected another way: each time a function with a contract attached is applied, the whole *context* must be traversed to check for a compatibility property.

The algorithmic system of [15] is, on balance, a technically impressive realization of the coinductive semantics that supports user-defined contracts, though it is fairly complex and probably difficult to implement efficiently.

### 5.3 Monitoring properties

Another realization of the coinductive semantics described in Section 5.1 ~~, is~~ is given in [23], which aims at simplifying the algorithmic system proposed by [15] and described in Section 5.2.

A key ingredient of [23] is to disallow user-defined contracts. This choice gives the authors more freedom in the quest of a more uniform operational semantics. This is sensible trade-off in the context of gradual typing à la TypeScript: the problem is to match contracts with static types, and user-defined contracts don't have a static type equivalent. On the other hand, the cost would probably not be worth it for a configuration language like Nickel.

As a means of proving the correctness of their simplified system, the authors introduce what they call *sound monitoring properties*. Here is the sound monitoring property for contexts of intersection contracts:

$$K \in [\![A \cap B]\!]^- \textit{ if } K \in [\![A]\!]^- \vee K \in [\![B]\!]^-$$

This reads as: a context $K$ satisfies the intersection of $A$ and $B$ if it satisfies at least one of the two. Morally, the $K$s in $[\![A \cap B]\!]^-$ should be the ones that can have their hole filled with a term satisfying $A \cap B$ without violating the $A \cap B$ contract.

Although sound, this interpretation is weaker than what the coinductive semantics permits. Consider the two contexts presented on Figure 17. The first one is a context satisfying `Num → Num`, applying the hole to a number. Similarly, the second context from the same figure satisfies `Bool → Bool`.

```
□ 3
```

`Num → Num` context

```
□ true
```

`Bool → Bool` context

**Figure 17.** Two different contexts in Nickel

Now, combining these two ~~context~~ contexts as in Figure 18 gives a context that doesn't satisfy `Num → Num` nor `Bool → Bool`. According to the sound monitoring property of intersection, Figure 18 thus doesn't satisfy `Num → Num ∩ Bool → Bool`.

```
let f = □ in
(f 3, f true)
```

**Figure 18.** Combined context

The consequence is that [23] doesn't prove their system complete for the coinductive semantics. It's probably just an oversight in the proof: we believe their system to be indeed complete for the coinductive semantics. But it does speak to the intrinsic complexity of union and intersection contracts: it is remarkably easy to get details wrong. This difficulty contrasts with the standard framework of higher-order contracts where satisfaction is much more straightforward.

### 5.4 Racket

Racket is a language based on the Scheme dialect of Lisp. Among established languages, Racket is probably the one with the most comprehensive contract system [1]. Regarding union and intersections, Racket provides the **`and/c`** and **`or/c`** combinators for contract.

The intersection combinator **`and/c`** corresponds to the naive interpretation of intersection described in Section 4.2: applying contract `(and/c A B)` is like applying contract `A` then applying contract `B`. In particular **`and/c`** doesn't model overloading. As in Figure 14, the example given in Figure 19 always fail because no argument satisfies both `number?` and `string?`.

```
(define/contract overload
(and/c (→ number? number?)
       (→ string? string?))
(lambda (x) x))
```

**Figure 19.** **`and/c`** and overloading

The union combinator **`or/c`** is similarly simple. It must be able to decide immediately which branch holds: **`or/c`** is a simple Boolean disjunction. For instance, when higher-order contracts are combined using **`or/c`**, Racket imposes that contracts must be distinguishable by their arity. Doing so, there is at most one candidate that can be selected directly. This is illustrated in Figure 20 whose program is accepted. On the other hand, the program of Figure 21 is rejected.

```
(define/contract united
(or/c (→ number? number?)
      (→ string? string? string?))
(lambda (x) x))
```

**Figure 20.** Accepted use of **`or/c`** with higher-order contracts

```
(define/contract united
(or/c (→ number? number?)
      (→ even? even?))
(lambda (x) x))
```

**Figure 21.** Rejected use of **or/c** with higher-order contracts

***case->.*** To compensate for the fact that **and/c** doesn't support overloading, Racket provides a second intersection-like combinator: **case→**. As for the **or/c**, the candidate contracts must have distinct arities to avoid ambiguity, unlike **or/c** the alternative chosen when the function is called rather than when the contract is applied to the function. An example is provided in Figure 22. The resulting possibilities are similar to static overloading, where one function can take additional parameters for example (e.g. as supported for Java methods). On the other hand, it excludes the overloading of generic operations with fixed arity such as equality, comparison, arithmetic operators, and so on.

```
(define/contract overcase
  (case→ (→ string? string?)
         (→ number? number? number?)
  )
  (lambda (x [y 0]) (if (number? x)
                        (+ x y)
                        x)))

(overcase 1 2)

(overcase "hello")
```

**Figure 22.** Overloading with the **case→** combinator

In conclusion, Racket, a programming language with a large user base, avoids the difficulties of general unions and intersections (Section 4). They make the pragmatic choice of a simple semantics with limited support for higher-order contracts in intersections and unions.

## 6 (More) Related work

### 6.1 Higher-order contracts

Enforcing pre- and post-conditions at runtime is a widely established practice. In their foundational paper [12], Findler and Felleisen introduce higher-order contracts, a principled approach to run-time assertion checking that nicely supports functions. They introduce the notion of blame, which is crucial to good error reporting. It became apparent later that their contracts are closely related to the type casts introduced by gradual typing, excluding blame: both [16] and [20] see the value of contracts as a safe interface between typed and untyped code. In [22], the authors precisely introduce a system integrating gradual typing with contracts *à la Findler & Felleisen*. Nickel adopts a similar type system, with both statically typed terms, dynamically typed terms, and first-class contracts. Higher-order contracts are the basis of the work ~~–[15] and –~~ of [15] and [23] that this paper ~~explorted~~ explored extensively.

### 6.2 Unions and intersections in gradual typing

Castagna ~~et. al [11]~~ and Lanvin [10] introduce a gradual type system based on so-called set-theoretic types [13]. Set-theoretic types feature unions, intersections, negation types together with a notion of subtyping. This work adheres to the *static first school* (see [14]) of gradual typing: the reason for gradual types is to allow for a less precise type information. It follows that their goals and constraints are somehow different, resulting in the absence of first-class contracts. As for any gradual type system, they do have to implement casts ~~—~~ – that are very close to contracts ~~—~~ – for unions and intersections, but these casts are neither visible nor available to the programmer. Obviously, no contracts mean no user-defined contracts as well.

Castagna and Lanvin use abstract interpretation to derive the semantics of unions and intersections. In their own words, "the resulting definitions are quite technical and barely intuitive but they have the properties we seek for [...]". This makes any comparison with the co-inductive semantics of Kiel and Thiemann rather difficult. Castagna et al. [11] build on Castagna and Lanvin to add polymorphism in the picture. However, it is at the price of restricting the union and intersection part: it is not possible to assign intersection types to a function anymore.

## 7 Conclusion

Despite the fact that union and intersection of dynamic properties may at first appear like an easy task, as soon as they are combined with higher-order contracts for increased accuracy of error messages, they really aren't.

The problem of union and intersection contracts is that they are not orthogonal to apparently independent features of programming languages. The mere presence of intersection and union contract induces computational effects, and can make it quite difficult to perform simple program optimizations such as inlining.

Designing a language with union and intersection contracts means making difficult choices: some features of union and intersection contracts and of the rest of the language must be abandoned. Various trade-offs can be made, but it is worth mentioning that implementing a system with union and intersection contracts appears to be pretty complex a task when unions and intersections are fairly complete.

It's hard not to have sympathy for the minimalist end of this spectrum, where, like Racket, a language only has a very simple notion of unions and intersections. Many applications of unions and intersections are not possible in such a context, but the presence of union and intersection contracts doesn't interact with the rest of the language. It's probably more manageable.

To conclude, let us make clear that we do not think union and intersection contracts are fundamentally broken, that they can not be implemented correctly, or that they do not bear any value (quite the contrary). They may still make sense to have in a language, and some apparent difficulties in the implementation could be lifted some day. But as often, there are gaps between the theoretical foundation, a proof-of-concept, a prototype, and the integration in an actual language. We hope that our attempt may serve as a cautionary tale: for union and intersection contracts, these gaps may be larger than they appear.

## References


[1] [n.d.]. *Contracts - Racket Documentation.* https://docs.racket-lang.org/reference/contracts.html

[2] [n.d.]. *The CUE Configuration Language.* https://cuelang.org/

[3] [n.d.]. *Design by Contract and Assertions - Eiffel.* https://www.eiffel.org/doc/solutions/Design_by_Contract_and_Assertions

[4] [n.d.]. *The Dhall configuration language.* https://dhall-lang.org/

[5] [n.d.]. *Jsonnet - The Data Templating Language.* https://jsonnet.org/

[6] [n.d.]. *The Nickel repository on GitHub.* https://github.com/tweag/nickel

[7] [n.d.]. *Optional types and the None type.* https://mypy.readthedocs.io/en/latest/kinds_of_types.html#optional-types-and-the-none-type

[8] [n.d.]. *Repology package tracker.* https://repology.org/repositories/statistics/total

[9] [n.d.]. *Union Types in TypeScript, TypeScript handbook.* https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types

[10] Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285

[11] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290329

[12] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. *SIGPLAN Not.* 37, 9 (Sept. 2002), 48–59. https://doi.org/10.1145/583852.581484

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55 (09 2008). https://doi.org/10.1145/1391289.1391293

[14] Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6

[15] Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-Order Contracts with Intersection and Union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 375–386. https://doi.org/10.1145/2784731.2784737

[16] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/1190216.1190220

[17] Bertrand Meyer. 1987. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices* 22, 2 (1987), 85–94.

[18] Francisco Ortin and Miguel García. 2011. Union and intersection types to support both dynamic and static typing. *Inform. Process. Lett.* 111, 6 (2011), 278–286. https://doi.org/10.1016/j.ipl.2010.12.006

[19] Simon Peyton Jones, Will Partain, and André Santos. 1996. Let-Floating: Moving Bindings to Give Faster Programs. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/232627.232630

[20] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 964–974. https://doi.org/10.1145/1176617.1176755

[21] Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. 382–404. https://doi.org/10.1007/978-3-319-66706-5_19

[22] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[23] Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The Root Cause of Blame: Contracts for Intersection and Union Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 134 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276504