The combination of width and depth subtyping cover the most interesting cases of object subtyping. A type system that implemented only these two would, however, needlessly annoy programmers. Other convenient (and mathematically necessary) rules include the ability to permute names, reflexivity (every type is a subtype of itself, because it is more convenient to interpret the subtype relationship as $\subseteq$), and transitivity. Languages like Typed JavaScript employ all these features to provide maximum flexibility to programmers.

16 Checking Program Invariants Dynamically: Contracts

Contents

Type systems offer rich and valuable ways to represent program invariants. However, they also represent an important trade-off, because not all non-trivial properties of programs can be verified statically. Furthermore, even if we can devise a method to settle a certain property statically, the burdens of annotation and computational complexity may be too great. Thus, it is inevitable that some of the properties we care about must either be ignored or settled only at run-time. Here, we will discuss run-time enforcement.

This is a formal property, known as Rice's Theorem.

Virtually every programming language has some form of assertion mechanism that enables programmers to write properties that are richer than the language's static type system permits. In languages without static types, these properties might start with simple type-like assertions: whether a parameter is numeric, for instance. However, the language of assertions is often the entire programming language, so any predicate can be used as an assertion: for instance, an implementation of a cryptography package might want to ensure certain parameters pass a primality test, or a balanced binary search-tree might want to ensure that its subtrees are indeed balanced and preserve the search-tree ordering.

16.1 Contracts as Predicates

It is therefore easy to see how to implement simple contracts. A contract embodies a predicate. It consumes a value and applies the predicate to the value. If the value passes the predicate, the contract returns the value unmolested; if the value fails, the contract reports an error. Its only behaviors are to return the supplied value or to error: it should not change the value in any way. In short, on values that pass the predicate, the contact itself acts as the identity function.

In what follows we will use the language `#lang plai`, for two reasons. First, this better simulates programming in an untyped language. Second, for simplicity we will write type-like assertions as contracts, but in the typed language these will be flagged by the type-checker itself, not letting us see the run-time behavior. In effect, it is easier to "turn off" the type checker. However, contracts make perfect sense even in a typed world, because they enhance the set of invariants that a programmer can express.

We can encode this essence in the following function:

```
(define (make-contract pred?)
  (lambda (val)
    (if (pred? val) val (blame "violation"))))

(define (blame s) (error 'contract "~a" s))
```

Here's an example contract:

```
(define non-neg?-contract
  (make-contract
   (lambda (n) (and (number? n)
                    (>= n 0)))))
```

(In a typed language, the `number?` check would of course be unnecessary because it can be encoded—and statically checked!—in the type of the function using the contract.) Suppose we want to make sure we don't get imaginary numbers when computing square roots; we might write

```
(define (real-sqrt-1 x)
  (sqrt (non-neg?-contract x)))
```

In many languages assertions are written as statements rather than as expressions, so an alternate way to write this would be:

```
(define (real-sqrt-2 x)
  (begin
    (non-neg?-contract x)
    (sqrt x)))
```

(In some cases this form is clearer because it states crisply at the beginning of the function what is expected of the parameters. It also enables parameters to be checked just once. Indeed, in some languages the contract can be written in the function header itself, thereby improving the information given in the interface.) Now if `real-sqrt-1` or `real-sqrt-2` are applied to `4` they produce `2`, but if applied to `-1` they raise a contract violation error.

16.2 Tags, Types, and Observations on Values

At this point we've reproduced the essence of assertion systems in most languages. What else is there to say? Let's suppose for a moment that our language is not statically typed. Then we will want to write assertions that reproduce at least traditional type-like invariants, if not more. `make-contract` above can capture all standard type-like properties such as checking for numbers, strings, and so on, assuming the appropriate predicates are either provided by the language or can be fashioned from the ones given. Or can it?

Recall that even our simplest type language had not just base types, like numbers, but also constructed types. While some of these, like lists and vectors, appear to not be very challenging, they are once we care about mutation, performance, and blame, which we discuss below. However, functions are immediately problematic.

As a working example, we will take the following function:

```
(define d/dx
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x 0.001))
            (f x))
         0.001))))
```

Statically, we would give this the type

```
((number -> number) -> (number -> number))
```

(it consumes a function, and produces its derivative—another function). Let us suppose we want to guard this with contracts.

The fundamental problem is that in most languages, we cannot directly express this as a predicate. Most language run-time systems store very limited information about the types of values—so limited that, relative to the types we have seen so far, we should use a different name to describe this information; traditionally they are called

tags. Sometimes tags coincide with what we might regard as types: for instance, a number will have a tag identifying it as a number (perhaps even a specific kind of number), a string will have a tag identifying it as a string, and so forth. Thus we can write predicates based on the values of these tags.

When we get to structured values, however, the situation is more complex. A vector would have a tag declaring it to be a vector, but not dictating what kinds of values its elements are (and they may not even all be of the same kind); however, a program can usually also obtain its size, and thus traverse it, to gather this information. (There is, however, more to be said about structured values below [REF].)

There have been a few efforts to preserve rich type information from the source program through lower levels of abstraction all the way down to assembly language, but these are research efforts.

Do Now!

Write a contract that checks that a list consists solely of even numbers.

Here it is:

```
(define list-of-even?-contract
  (make-contract
   (lambda (l)
     (and (list? l) (andmap number? l) (andmap even? l)))))
```

(Again, note that the first two questions need not be asked if we know, statically, that we have a list of numbers.) Similarly, an object might simply identify itself as an object, not providing additional information. But in languages that permit reflection on the object's structure, a contract can still gather the information it needs.

In every language, however, this becomes problematic when we encounter functions. We might think of a function as having a type for its domain and range, but to a run-time system, a function is just an opaque object with a function tag, and perhaps some very limited metadata (such as the function's arity). The run-time system can hardly even tell whether the function consumes and produces functions—as opposed to other kinds of values—much less whether they it consumes and produces ones of `(number -> number)` type.

This problem is nicely embodied in the (misnamed) `typeof` operator in JavaScript. Given values of base types like numbers and strings, `typeof` returns a string to that effect (e.g., `"number"`). For objects, it returns `"object"`. Most importantly, for functions it returns `"function"`, with no additional information.

For this reason, perhaps `typeof` is a bad name for this operator. It should have been called `tagof` instead, leaving open the possibility that future static type systems for JavaScript could provide a true `typeof`.

To summarize, this means that at the point of being confronted with a function, a function contract can only check that it is, indeed, a function (and if it is not, that is clearly an error). It cannot check anything about the domain and range of the function. Should we give up?

16.3 Higher-Order Contracts

To determine what to do, it helps to recall what sort of guarantee contracts provide in the first place. In `real-sqrt-1` above, we demanded that the argument be non-negative. However, this is only checked if—and when—`real-sqrt-1` is actually used, and then only on the actual values that are passed to it. For instance, if the program contains this fragment

```
(lambda () (real-sqrt-1 -1))
```

but this thunk is never invoked, the programmer would never see this contract violation. In fact, it may be that the thunk is not invoked on this run of the program, but in a later run it will be; thus, the program has a lurking contract error. For this reason, it is usually preferable to express invariants through static types; but where we do use contracts, we understand that it is with the caveat that we will only be notified of errors when the program is suitably exercised.

This is a useful insight, because it offers a solution to our problem with functions. We check, immediately, that the purported function value truly is a function. However, instead of ignoring the domain and range contracts, we *defer* them. We check the domain contract when (and each time) the function is actually applied to a value, and we check the range contract when the function actually returns a value.

This is clearly a different pattern than `make-contract` followed. Thus, we should give `make-contract` a more descriptive name: it checks *immediate* contracts (i.e., those that can be checked in their entirety now).

In the Racket contract system, immediate contracts are called *flat*. This term is slightly misleading, since they can also protect data structures.

```
(define (immediate pred?)
  (lambda (val)
    (if (pred? val) val (blame val))))
```

In contrast, a function contract takes two *contracts* as arguments—representing checks to be made on the domain and range—and returns a predicate. This is the predicate to apply on values purporting to satisfy that contract. First, this checks that the given value actually is a function: this part is still immediate. Then, we create a *surrogate* procedure that applies the "residual" contracts—to check the domain and range—but otherwise behaves the same as the original function.

This creation of a surrogate represents a departure from the traditional assertion mechanism, which simply checks values and then leaves them alone. Instead, for functions we must use the created surrogate if we want contract checking. In general, therefore, it is useful to have a wrapper that consumes a contract and value, and creates a guarded version of that value:

```
(define (guard ctc val) (ctc val))
```

As a very simple example, let us suppose we want to wrap the `add1` function in numeric contracts (with `function`, the constructor of function contracts, to be defined momentarily):

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1))
```

We want `a1` to be bound to essentially the following code:

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

Here, the `(lambda (x) ...)` is the surrogate; it applies two numeric contracts around the invocation of `add1`. Recall that contracts must behave like the identity function in the absence of violations, so this procedure has precisely the same behavior as `add1` on non-violating uses.

To achieve this, we use the following definition of `function`. Remember that we have to also ensure that the given value is truly a function (as `add1` above indeed is, and can be checked immediately, which is why the check has disappeared by the time we bind the surrogate to `a1`):

For simplicity we assume single-argument functions here, but the extension to multiple arity is straightforward. Indeed, more complex contracts can even check for relationships *between* the arguments.

```
(define (function dom rng)
  (lambda (val)
    (if (procedure? val)
        (lambda (x) (rng (val (dom x))))
        (blame val))))
```

To understand how this works, let us substitute arguments. To keep the resulting code readable, we will first construct the `number?` contract checker and give it a name:

```
  (define num?-con (immediate number?))
= (define num?-con
    (lambda (val)
      (if (number? val) val (blame val))))
```

Now let's return to the definition of `a1`. First we apply `guard`:

```
(define a1
  ((function num?-con num?-con)
   add1))
```

Now we apply the function contract constructor:

```
(define a1
  ((lambda (val)
     (if (procedure? val)
         (lambda (x) (num?-con (val (num?-con x))))
         (blame val)))
   add1))
```

Applying the left-left-lambda gives:

```
(define a1
  (if (procedure? add1)
      (lambda (x) (num?-con (add1 (num?-con x))))
      (blame val)))
```

Notice that this immediately checks that the guarded value is indeed a function. Thus we get

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

which is precisely the surrogate we desired, with the behavior of `add1` on non-violating executions.

Do Now!

How many ways are there to violate the above contract for `add1`?

There are three ways, corresponding to the three contract constructors:

1. the value wrapped might not be a function;
2. the wrapped value might be a function that is applied to a non-numeric value; or,
3. the wrapped value might be a function that consumes numbers but produces values of non-numeric type.

Exercise

Write examples that perform each of these three violations, and observe the behavior of the contract system. Can you improve the error messages to better distinguish these cases?

The same wrapping technique works for `d/dx` as well:

```
(define d/dx
  (guard (function (function (immediate number?) (immediate number?))
                   (function (immediate number?) (immediate number?)))
         (lambda (f)
           (lambda (x)
             (/ (- (f (+ x 0.001))
                   (f x))
                0.001)))))
```

Exercise

There are seven ways to violate this contract, corresponding to each of the seven contract constructors. Violate each of them by passing arguments or modifying code, as needed. Can you improve error reporting to correctly identify each kind of violation?

Notice that the nested function contract defers the checking of the immediate contracts for two applications, rather than one. This is what we should expect, because immediate contracts only report problems with actual values, so they cannot report anything until applied to actual values. However, this does mean that the notion of "violation"' is subtle: the function value passed to `d/dx` may in fact truly be in violation of the contract, but this violation will not be *observed* until numeric values are passed or returned.

16.4 Syntactic Convenience

Earlier we saw two styles of using flat contracts, as embodied in `real-sqrt-1` and `real-sqrt-2`. Both styles have disadvantages. The latter, which is reminiscent of traditional assertion systems, simply does not work for higher-order values, because it is the wrapped value that must be used in the computation. (Not surprisingly, traditional assertion systems only handle immediate contracts, so they fail to notice this subtlety.) The style in the former, where we wrap each use with a contract, works in theory but suffers from three downsides:

1. The developer may forget to wrap some uses.
2. The contract is checked once per use, which is wasteful when there is more than one use.
3. The program comingles contract checking with its functional behavior, reducing readability.

Fortunately, a judicious use of syntactic sugar can solve this problem in common cases. For instance, suppose we want to make it easy to attach contracts to function parameters, so a developer could write

```
(define/contract (real-sqrt (x :: (immediate positive?)))
  (sqrt x))
```

with the intent of guarding `x` with `positive?`, but performing the check only once, on function invocation. This should translate to, say,

```
(define (real-sqrt new-x)
  (let ([x (guard (immediate positive?) new-x)])
    (sqrt x)))
```

That is, the macro generates a fresh name for each identifier, then associates the name given by the user to the wrapped version of the value supplied to that fresh name. The following macro implements exactly this:

```
(define-syntax (define/contract stx)
  (syntax-case stx (::)
    [(_ (f (id :: c) ...) b)
     (with-syntax ([(new-id ...) (generate-temporaries #'(id ...))])
       #'(define f
           (lambda (new-id ...)
             (let ([id (guard c new-id)]
                   ...)
               b))))]))
```

With conveniences like this, designers of contract languages can improve the readability, efficiency, and robustness of contract use.

16.5 Extending to Compound Data Structures

As we have already discussed, it appears easy to extend contracts to structured datatypes such as lists, vectors, and user-defined recursive datatypes. This only requires that the appropriate set of run-time observations be available. This will usually be the case, up to the resolution of types in the language. For instance, as we have discussed [REF], a language with datatypes does not require *type* predicates but will still offer predicates to distinguish the *variants*; this is case where type-level "contract" checking is best (and perhaps must) be left to the static type system, while the contacts assert more refined structural properties.

However, this strategy can run into significant performance problems. For instance, suppose we built a balanced binary search-tree to perform asymptotic logarithmic time (in the size of the tree) insertions and lookups. Now say we have wrapped this tree in a suitable contract. Sadly, the mere act of checking the contract visits the entire tree, thereby taking linear time! Ideally, therefore, we would prefer a strategy whereby the contract was already checked—incrementally—at the time of construction, and does not need to be checked again at the time of lookup.

Worse, both balancing and search-tree ordering are recursive properties. In principle, therefore, they attach to every sub-tree, and so should be applied on every recursive call. During insertion, which is a recursive procedure, the contract would be checked on every visited sub-tree. In a tree of size t, the contract predicate applies to a sub-tree of $\frac{t}{2}$ elements, then to a sub-sub-tree of $\frac{t}{4}$ elements, and so on, resulting—in the worst case—in visiting a total of $\frac{t}{2} + \frac{t}{4} + \cdots + \frac{t}{t}$ elements...making our intended logarithmic-time insertion process take linear time.

In both cases, there is ready mitigation available in many cases. Each value needs to be associated (either intrinsically, or by storage in a hash table) with the set of contracts it has already passed. Then, when a contract is ready to apply, it first checks whether the value has already been checked and, if it has, does not check again. This is essentially a form of memoization of contract checking and can thus reduce the algorithmic complexity of checking. Again, like memoization, this works best when the values are immutable. If the values can mutate and the contracts perform arbitrary computations, it may not be sound to perform this optimization.

There is a subtler way in which we might examine the issue of data structures. As an example, consider the contract we wrote earlier to check that all values in a numeric list are even. Suppose we have wrapped a list in this contract, but are interested only in the first element of the list. Naturally, we are paying the cost of checking all the values in the list, which may take a very long time. More importantly, however, a user might argue that reporting a violation about the second element of the list is itself a violation of our expectation about contract-checking, since we did not actually use that element.

This suggests deferring checking even for some values that could be checked immediately. For instance, the entire list could be turned into a wrapped value containing a deferred check, and each value is checked only when it is visited. This strategy might be attractive, but it is not trivial to code, and especially runs into problems in the presence of *aliasing*: if two different identifiers are referring to the same list, one with a contract guard and the other without, we have to ensure both of them function as expected (which usually means we cannot store any mutable state in the list itself).

16.6 More on Contracts and Observations

A general problem for any contract implementation—which is exacerbated by complex data—is a curious one. Earlier, we complained that it was difficult to check function contracts because we have insufficient power to observe: all we can check is that a value is a function, and no more. In real languages, the problem for data structures is actually the opposite: we have too much ability to observe. For instance, if we implement a strategy of deferring checking of a list, we quite possibly need to use a structure to hold the actual list, and modify `first` and `rest` to get their values through this structure (after checking contracts). However, a procedure like `list?` might now return `false` rather than `true` because structures are not lists; therefore, `list?` needs to be re-bound to a procedure that also returns `true` on structures that represent these special deferred-contract lists. But the contract system author needs to also remember to tackle `cons?`, `pair?`, and goodness knows how many other procedures that all perform observations.

In general, one observation is essentially impossible to "fix": `eq?`. Normally, we have the property that every value is `eq?` to itself, even for functions. However, the wrapped value of a function is a new procedure that not only *isn't* `eq?` to itself but probably *shouldn't* be, because its behavior truly is different (though only on contract violations, and only after enough values have been supplied to observe the violation). However, this means that a program cannot surreptitiously guard itself, because the act of guarding can be observed. As a result, a malicious module can sometimes detect whether it is being passed guarded values, behaving normally when it is and abnormally only when it is not!

16.7 Contracts and Mutation

We should rightly be concerned about the interaction between contracts and mutation, and even more so when we have contracts that are either inherently deferred or have been implemented in a deferred fashion. There are two things to be concerned about. One is storing a contracted value in mutable state. The other is writing a contract *for* mutable state.

When we store a contracted value, the strategy of wrapping ensures that contract checking works gracefully. At each stage, a contract checks as much as it can with the value at hand, and creates a wrapped value embodying the residual check. Thus, even if this wrapped value is stored in mutable state and retrieved for use later, it still contains these checks, and they will be performed when the value is eventually used.

The other issue is writing contracts for mutable data, such as boxes and vectors. In this case we probably have to create a wrapper for the entire datatype that records the intended contract. Then, when a value inside the datatype is replaced with a new one, the operation that performs the update—such as `set-box!`—needs to retrieve the intended contract from the wrapper, apply it to the value, and store the wrapped value. Therefore, this requires changing the behavior of the data structure mutation operators to be sensitive to contracted values. However, mutation does not change the point at which violations are caught: right away for immediate contracts, upon (in)appropriate use for deferred ones.

16.8 Combining Contracts

Now that we've discussed combinators for all the basic datatypes, it's natural to discuss combining contracts. Just as we saw unions [REF] and intersections [REF] for types, we should be considering unions and intersections (respectively, "or"s and "and"s), ; for that matter, we might also consider negation. However, contracts are only superficially like types, so we have to consider these questions in their own light for contracts rather than try to map the meanings we have learned from types to the sphere of contracts.

As always, the immediate case is straightforward. Union contracts combine with disjunction—indeed, being predicates, their results can literally be combined with `or`—and intersection contracts with conjunction. We apply the predicates in turn, with short-circuiting, and either generate an error or return the contracted value. Intersection contracts combine with conjunction (`and`). And negation contracts are simply the original immediate contract applied and the decision negated (with `not`).

Contract combination is much harder in the deferred, higher-order case. For instance, consider the negation of a function contract from numbers to numbers. What exactly does it mean to negate it? Does it mean the function should *not* accept numbers? Or that if it does, it should not produce them? Or both? And in particular, how do we enforce such a contract? How, for instance, do we check that a function does not accept numbers—are we expecting that when given a number, it produces an error? But now consider the identity function wrapped with such a contract; since it clearly does not result in an error when given a number (or indeed any other value), does that mean we should wait until it produces a value, and if it does produce a number, reject it? But worst of all, note that this means we will be running functions on domains on which they are *not* defined: a sure recipe for destroying program invariants, polluting the heap, or crashing the program.

Intersection contracts require values to pass all the sub-contracts. This means re-wrapping the higher-order value in something that checks all the domain sub-contracts as well as all the range sub-contracts. Failing to meet even one sub-contract means the value has failed the entire intersction.

Union contracts are more subtle, because failing to meet any one sub-contract is not grounds for rejection. Rather, it simply means that that one sub-contract is no longer a candidate contract representing the wrapped value; the other sub-contracts might still be candidates, and only when no others are left must be reject the value. This means the implementation of union contracts must maintain memory of which sub-contracts have and have not yet passed—memory, in this case, being a sophisticated term for the use of mutation. As each sub-contract fails, it is removed from the list of candidates, while all the remaining ones continue to applied. When no candidates remain, the contract system must report a violation. The error report would presumably provide the actual values that eliminated each part of each sub-contract (keeping in mind that these may be nested multiple functions deep).

In a multi-threaded language like Racket, this also requires locks to avoid race conditions.

The implemented versions of contract constructors and combinators in Racket place restrictions on the acceptable forms of sub-contracts. These enable implementations that are both efficient and yield useful error messages. Furthermore, the more extreme situations discussed above rarely occur in practice—though now you know how to

implement them if you need to.

16.9 Blame

Let's now return to the issue of reporting contract violations. By this I don't mean what string to we print, but the much more important question of *what* to report, which as we are about to see is really a semantic consideration.

To illustrate the problem recall our definition of `d/dx` above, and assume we were running it without any contract checking. Suppose now that we apply this function to the entirely inappropriate `string-append` (which neither consumes nor produces numbers). This simply produces a value:

```
> (define d/dx-sa (d/dx string-append))
```

(Observe that this would succeed even if contract checking were on, because the immediate portion of the function contract recognizes `string-append` to be a function.) Now suppose we apply `d/dx-sa` to a number, as we ought to be able to do:

```
> (d/dx-sa 10)
string-append: contract violation
  expected: string?
  given: 10.001
```

Notice that the error report is deep inside the body of `d/dx`. On the one hand, this is entirely legitimate: that is where the improper application of `string-append` occurred. However, the *fault* is not that of `d/dx` at all—rather, it is the fault of whatever body of code supplied `string-append` as a purportedly legitimate function from numbers to numbers. Except, however, the code that did so has long since fled the scene; it is no longer on the stack, and is hence outside the ambit of traditional error-reporting mechanisms.

This problem is not a peculiarity of `d/dx`; in fact, it routinely occurs in large systems. This is because systems, especially with graphical, network, and other external interfaces, make heavy use of *callbacks*: functions (or methods) that register interest in some entity and are invoked to signal some status or value. (Here, `d/dx` is the moral equivalent of the graphics layer, and `string-append` is the callback that has been supplied to (and stored by) it.) Eventually, the system layer invokes the callback. If this results in an error, it is the fault of *neither* the system layer—which was given a callback of purportedly the right contract—*nor* of the callback itself, which presumably has legitimate uses but was improperly supplied to the function. Rather, *the fault is of the entity that introduced these two entities*. However, at this point the call stack contains only the callback (on top) and the system (below it)—and the only guilty party is no longer present. These kinds of errors can therefore be extremely difficult to debug.

The solution is to extend the contract system to incorporate a notion of *blame*. The idea is to effectively record the introduction that resulted in a pair of components coming together, so that if a contract violation occurs between them, we can ascribe the failure to the expression that did the introduction. Observe that this is only really interesting in the context of functions, but for consistency we will extend blame to immediate contracts as well in a natural way.

For a function, notice that there are two possible points of failure: either it was *given* the wrong kind of value (the pre-condition), or it *produced* the wrong kind of

value (the post-condition). It is important to distinguish these two cases because in the former case we should blame the environment—in particular, the actual parameter expression—whereas in the latter case (assuming the parameter has passed muster) we should blame the function itself. (The natural extension to immediate values is that we can only blame the value itself for not satisfying the contract, which is akin to the "post-condition".)

For contracts, we will introduce the terms *positive* and *negative* position. For a first-order function, the negative position is the pre-condition and the positive one the post-condition. Therefore, this might appear to be needless extra terminology. As we will soon see, however, these terms have a more general meaning.

We will now generalize the parameters consumed by contracts. Previously, immediate contracts consumed a predicate and function contracts consumed domain and range contracts. This will still be the case. However, what they each return will be a function of two arguments: labels for the positive and negative positions. (These labels can be drawn from any reasonable datatype: abstract syntax nodes, buffer offsets, or other descriptions. For simplicity, we will use strings.) Thus function contracts will close over the labels of these program positions, to later blame the provider of an invalid function.

The `guard` function is now responsible for passing through the labels of the contract application locations:

```
(define (guard ctc val pos neg) ((ctc pos neg) val))
```

and let us also have `blame` display the appropriate label (which we will pass to it from the contract implementations):

```
(define (blame s) (error 'contract s))
```

Suppose we are guarding the use of `add1`, as before. What are useful names for the positive and negative positions? The positive position is post-condition: i.e., any failure here must be blamed on the body of `add1`. The negative position is the pre-condition: i.e., any failure here must be blamed on the parameter to `add1`. Thus:

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1
                  "add1 body"
                  "add1 input"))
```

Had we provided a non-function to guard, we would expect an error at the "post-condition" location: this is not really a failure of the post-condition, but surely the parameter cannot be blamed if the application failed to be a function. (However, this shows that we are really stretching the term "post-condition", and the terms "positive" provides a useful alternative.) Because we trust the implementation of `add1` to only produce numbers, we would expect it is impossible to fail the post-condition. However, we would expect an expression like `(a1 "x")` to trigger a pre-condition error, presumably signaling a contract error at the location `"add1 input"`. In contrast, had we guarded a function that violates the post-condition, such as this,

```
(define bad-a1 (guard (function (immediate number?)
                                (immediate number?))
                      number->string
                      "bad-add1 body"
                      "bad-add1 input"))
```

we would expect blame to be ascribed to `"bad-add1 body"`.

Let us now see how to implement these contract constructors. For immediate contracts, we have seen that blame should be ascribed to the positive position:

```
(define (immediate pred?)
  (lambda (pos neg)
    (lambda (val)
      (if (pred? val) val (blame pos)))))
```

For functions, we might be tempted to write

```
(define (function dom rng)
  (lambda (pos neg)
    (lambda (val)
      (if (procedure? val)
          (lambda (x) (dom (val (rng x))))
          (blame pos)))))
```

but this fails to work in a very fundamental way: it violates the expected signature on contracts. That is because all contracts now expect to be given the labels of positive and negative positions, which means `dom` and `rng` cannot be used as above. (As another hint, we are using `pos` but not `neg` anywhere in the body, even though we have seen examples where we expect the position bound to `neg` to be blamed.) Instead, clearly, we somehow instantiate the domain and range contracts using `pos` and `neg`, so that they "know" and "remember" where a potentially violating function was applied.

The most obvious reaction would be to instantiate these contract constructors with the same values of `dom` and `rng`:

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom pos neg)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

Now all the signatures match up, and we can run our contracts. But when we do so, the answers are a little strange. For instance, on our simplest contract violation example, we get

```
> (a1 "x")
contract: add1 body
```

Huh? Maybe we should expand out the code of `a1` to see what happened.

```
   (a1 "x")
 = (guard (function (immediate number?)
                    (immediate number?))
          add1
          "add1 body"
          "add1 input")
 = (((function (immediate number?) (immediate number?))
     "add1 body" "add1 input")
    add1)
 = (let ([dom-c ((immediate number?) "add1 body" "add1 input")]
         [rng-c ((immediate number?) "add1 body" "add1 input")])
     (lambda (x) (rng-c (add1 (dom-c x)))))
 = (let ([dom-c (lambda (val)
                  (if (number? val) val (blame "add1 body")))]
         [rng-c (lambda (val)
                  (if (number? val) val (blame "add1 body")))])
     (lambda (x) (rng-c (add1 (dom-c x)))))
```

Poor `add1`: it never stood a chance! The only blame label left is `"add1 body"`, so it was the only thing that could ever be blamed.

We will return to this problem in a moment, but observe how in the above code, there are no real traces of the function contract left. All we have are immediate contracts, ready to blame actual values if and when they occur. This is perfectly consistent with what we said earlier [REF] about being able to observe only immediate values. Of course, this is only true for first-order functions; when we get to higher-order functions, this will no longer be true.

What went wrong? Notice that only the contract bound to `rng-c` ought to be blaming the body of `add1`. In contrast, the contract bound to `dom-c` ought to be blaming the input to `add1`. It's almost as if, in the domain position of a function contract, the positive and negative labels should be...swapped.

If we consider the contract-guarded `d/dx`, we see that this is indeed the case. The key insight is that, when applying a function taken as a parameter, the "outside" becomes the "inside" and vice versa. That is, the body of `d/dx`—which was in positive position—is now the caller of the function to differentiate, putting that function's body in positive position and the caller—the body of `d/dx`—in negative position. Thus, on the domain side of the contract, every nested function contract causes positive and negative positions to swap.

On the range side, there is no need to swap. Consider again `d/dx`. The function it returns represents the derivative, so it should be given a number (representing the point at which to calculate the derivative) and it should return a number (the derivative at that point). The negative position of this function is indeed the client who uses the derivative function—the pre-condition—and the positive position is indeed the body of `d/dx` itself—the post-condition—since it is responsible for generating the derivative.

As a result, we obtain an updated, and correct, definition for the function constructor:

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom neg pos)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

Exercise

> Apply this to our earlier example and confirm that we get the expected blame. Also expand the code manually to see why this happens.

Suppose, further, we define `d/dx` with the labels `"d/dx body"` for its positive position and `"d/dx input"` for its negative. Say we supply the function `number->string`, which patently does not compute derivatives, and apply the result to `10`:

```
((d/dx (guard (function (immediate number?)
                        (immediate string?))
              number->string
              "n->s body"
              "n->s input"))
 10)
```

This correctly indicates that the blame should be ascribed to the expression that fed `number->string` as a supposed numeric function to `d/dx`—not to `d/dx` itself.

Exercise

> Hand-evaluate `d/dx`, apply it to *all* the relevant violation examples, and confirm that the resulting blame is accurate. What happens if you supply `d/dx` with `string->number` with a function contract indicating it maps strings to numbers? What if you supply the same function with no contract at all?

17 Alternate Application Semantics

Long ago [REF], we considered the question of what to substitute when performing application. Now we are ready to consider some alternatives. At the time, we suggested just one alternative; in fact there are many more. To understand this, see whether you can answer this question:

Which of these is the same?

- `(f x (current-seconds))`
- `(f x (current-seconds))`
- `(f x (current-seconds))`