The Problem

Higher-order contracts defer contract checking. These checks build up during recursive calls and across module boundaries.

Example: Factorial

Here is an (almost) tail-recursive version of everyone's favorite recursive function:

```
: factorial (x:int | x>=0) -> (x:int | x>=0) -> (x:int | x>=0)
define factorial acc x =
  if x < 2
  then acc
  else factorial (acc * x) (x - 1)
```

In practice, tail-call optimization is ruined because each function call creates a deferred contract with pre and post-conditions to check.

Example: Mutually recursive modules

We could just write an iterative `factorial` and avoid the problem altogether. A similar fix will not work for mutally-recursive modules—you cannot collapse those boundaries.

```
module A
  : fact1 (...)
  define fact1 acc x =
    if x < 2
    then acc
    else B.fact2 (acc * x)
                 (x - 1)
```

```
module B
  : fact2 (...)
  define fact2 acc x =
    if x < 2
    then acc
    else A.fact1 (acc * x)
                 (x - 1)
```

Related work confirmed space-efficiency for gradual types. This paper addresses space efficiency for predicate contracts like those needed to write `factorial`: refinement types of the form $\{x\,{:}\,\tau \mid f(x)\}$ where $f(x)$ depends on a single variable of base type.

Eidetic space efficiency

The key idea of this paper is that if we have a chain of casts, we can collapse them. Let f represent a function value (a lambda). Let T_1, T_2, and T_3 represent different function types that we want to assert that f has. Also let l_1 and l_2 be blame labels—if an assertion fails, the label shows who to blame. Now the chain of casts:

$$\langle T_2 \Rightarrow T_3\rangle^{l_2}(\langle T_1 \Rightarrow T_2\rangle^{l_1}\ f)$$

ultimately converts the type of f from T_1 to T_3. So we can essentially reduce the casts to be:

$$\langle T_1 \Rightarrow T_3\rangle f$$

The problem is, we still need to assert T_2 and properly assign blame. We can do that by changing the fat arrow from $\langle T_1 \Rightarrow T_3\rangle$ to $\langle T_1 \Rightarrow^c T_3\rangle$ and defining a semantics for this strange new check, c.

Coercions

The c stands for "coercion".

- Coercions are either arrows $c_1 \mapsto c_2$ or stacks of assertions r.
- The stacks preserve the order of assertions and the original blame labels.
- The stacks also eliminate duplicates (the stack r is really an ordered set).
  If T_1 and T_2 are the same assertion, you can drop T_2 because you will never ever reach it.
- The stacks have 5 elements: $r = (\{x : B \mid e_1\}, s, r, k, e_2)$.
  - $\{x : B \mid e_1\}$ is the target type.
  - s is a status bit. It indicate whether we have reached the target.
  - r is a list of unchecked assertions.
  - k is the constant we are checking.
  - e_2 is either k or the current active check on k.

Basically, instead of blowing up the call stack with deferred contracts, we build up a new data structure and check as we go. The data structure is a tree of stacks: we make a pair of stacks for each arrow coercion $c_1 \mapsto c_2$. This is what all the symbols in the paper explain.

Soundness

Section 7 and the appendix show that collapsing contracts behaves the same as the Classic space-inefficient method of wrapping and deferring checks. Programs in both models either both terminate successfully or both blame the same label. The proof goes by logical relations—see Figure 11 for an overview. This means that the results only apply to strongly normalizing programs. Also interesting: type equality works by encoding each possible using $\log_2(n)$ bits. This works because the number of types n in any program in this language is finite. But I am not sure if the types $\{x : \tau \mid x \geq 0\}$ and $\{x : \tau \mid 0 \leq x\}$, for example, are identified as equal.

Alternatives

The first half of the paper presents two design alternatives: Heedful and Forgetful contracts. Together with the Eidetic and Classic calculii, we have 4 possibilities for manifest contracts.

Heedful space efficiency

The Heedful calculus (described in Section 5) naïvely collects a set of assertions instead of building the more sophisticated coercion stack. This is a simple design and it co-terminates with Classic and Eidetic; however, it does not always assign blame the same way Classic does.

Forgetful space efficiency

The Forgetful calculus (described in Section 4) is extremely simple: the coercion $\langle T_2 \Rightarrow T_3\rangle^{l_2}(\langle T_1 \Rightarrow T_2\rangle^{l_1}\ f)$ is cast directly to $\langle T_1 \Rightarrow T_3\rangle f$. No more T_2! This design may terminate on programs that Classic rejects, but it is both very simple to implement *and* type sound. Soundness follows because even though we assert T_2, no operation depends on f actually having type T_2 because the T_3 happens immediately afterwards.

Bounds for space efficiency

Table 1, copied here from the paper, shows the space bounds for each design alternative. The third column is the important one: Forgetful, Heedful, and Eidetic contracts bound the size of pending casts to the syntactic size of the program e. The second column needs a little more explaining. First off, L is the size of a blame label (i.e. a small, constant number of bits). The symbol W_h represents the number of bits required to encode a type of height h. The height h of a type is (approximately) the number of arrows in it. Finally, s is the number of positions in a type (there are two positions per arrow).

The figure says that each cast in Classic and Forgetful mode consist of two types (the "before" and "after") and a label. Casts in Heedful mode are offset by the worst-case constant 2^{W_h} because each cast is associated with a set of intermediate casts. The Eidetic coercion stacks contain only flat (non-arrow) types of height 1, but there may be up to s of them in a single cast.

Mode	**Cast Size**	**Pending Casts**
Classic	$2W_h + L$	∞
Forgetful	$2W_h + L$	$\lvert e\rvert$
Heedful	$2W_h + 2^{W_h} + L$	$\lvert e\rvert$
Eidetic	$s2^{L+W_1}$	$\lvert e\rvert$

Table 1: Space efficiency of λ_H

Conclusion

This paper shows that space efficiency is possible; future work will determine whether such a system is feasible. Other open questions are whether can we devise space-efficient stateful (i.e. effectful) or dependent contracts.