Gram: A linear functional language with graded modal types (extended abstract)

Dominic Orchard, Vilem-Benjamin Liepelt
School of Computing, University of Kent, UK

Many modern programs are *resource sensitive*, that is, the amount of resources (*e.g.*, energy, bandwidth, time, memory), and their rate of consumption, must be carefully managed. Furthermore, many programs handle *sensitive resources*, such as passwords, location data, photos, and banking information. Ensuring that private data is not inadvertently leaked is as important as the functional input-output behaviour of a program.

Various type-based solutions have been provided for reasoning about and controlling resources. A general class of program behaviours called *coeffects* has been proposed as a unified framework for capturing different kinds of resource analysis in a single type theory [6, 7, 2, 3, 4, 1]. Recently it has been shown how coeffect types can be integrated with effect types for resource reasoning with effects [3].

To gain experience with such type systems for real-world programming tasks, and as a vehicle for further research, we are developing **Gram**[1], a functional programming language based on the linear λ-calculus augmented with *graded modal types*, inspired by the coeffect-effect calculus [3].

Graded modal type theory A graded modality is an indexed family of modalities with some additional structure on the indices which mirrors the structure of the axioms/proof rules. For example, the exponential modality of linear logic ! has a graded counterpart in Bounded Linear Logic [5], where ! is replaced with a family of modalities $!_n$ indexed by the natural numbers (the reuse bound). The operations of the usual natural number semiring are then used in the axioms/rules of the logic *e.g.*, the transitivity axiom is $!_{n*m}A \rightarrow !_n!_mA$. There are various different examples in the literature under the name of *coeffects* which provide fine-grained analysis of resources and context-dependence via graded necessity modalities.

The goal with **Gram** is to support arbitrary, user-customisable graded modalities to enable fine-grained, quantitative program reasoning. At the moment, there are three built-in modalities: BLL-style resource-bounded graded necessity, a security-lattice graded necessity, and an effect-graded possibility modality for I/O. Type checking is based on a bidirectional algorithm, interfacing with the Z3 SMT solver to discharge constraints.

Example 1: Reuse bounds The following is a valid **Gram** program:

```
dub : |Int| 2 -> Int
dub |x| = x + x

trip : |Int| 3 -> Int
trip |x| = x + x + x

twice : forall c . |(|Int| c -> Int)| 2 -> |Int| (2 * c) -> Int
twice |g| |x| = g |x| + g |x|

main : Int
main = twice |dub| |1| + twice |trip| |1|
```

The first definition specifies a function `dub` on the integers (type `Int`) whose first parameter is used non-linearly, exactly twice, as captured by the resource bound 2 indexing the modality. The type `|Int| n` can be read as $!_n$`Int` in Girard *et al.*'s notation. The pattern match `|x|` discharges the incoming modality and binds `x` as a non-linear variable. Looking at the type signature for `twice`, we can deduce that it is a higher-order function: its first parameter is a unary function whose parameter is used non-linearly exactly `c` times and which returns an `Int`—a good fit for `dub` and `trip`. The second parameter of `twice` is used non-linearly exactly

[1] `http://github.com/dorchard/gram_lang`

`2 * c` times, since `g` uses `c` copies of its first parameter and is applied twice. Thus, `main` will produce the value `10`. This example shows **Gram**'s support of coeffect polymorphism.

Example 2: Security levels Another modality available in **Gram** is indexed by a two-point security lattice with levels: `Lo` and `Hi`. For example:

```
secret : |Int| Hi                                -- specified as Hi security
secret = |42|

dub : forall (l : Level) . |Int| l -> |Int| l    -- at any level...
dub |x| = |(x + x)|                              -- ...double an int

main : |Int| Hi
main = dub secret                                -- double the secret
```

Here `main` is marked as a high-security value via its modal type. The `dub` function appears again, but its type now tracks security levels and is level-polymorphic. It takes an integer at any level `l`, returning a value at the same level. Crucially, the following program is ill-typed:

```
leak : |Int| Hi -> |Int| Lo                      -- fails to type check
leak |x| = |x|
```

However, we can define a well-typed constant function that discards its high-security value to produce a low-security value by combining resource bounds with security levels:

```
notALeak : ||Int| Hi| 0 -> |Int| Lo
notALeak x = |0|
```

Example 3: Effects A graded possibility modality provides tracking of side effects in the style of a graded monad and effect system. A type `<t> f` describes a computation returning a value of type `t` and producing side effects `f`.

In the following code, input (read) and output (write) operations to the `stdio` are tracked:

```
echo : <Int> [R, W]
echo = let <x : Int> = read in write x
```

The following shows both reuse bound coeffects and I/O effects coming together, explaining the side-effects of twice applying some integer function which has a read effect:

```
doTwice : |(Int -> <Int> [R])| 2 -> |Int| 2 -> <Int> [R, R]
doTwice |f| |x| = let <a : Int> = f x in
                  let <b : Int> = f x in <a + b>
```

Future work We are currently working on making the language more featureful (*e.g.*, adding recursion, algebraic data types). We are exploring various avenues of further work: (1) combining different modalities smoothly, including compositional coeffects and interaction between different coeffects and effects; (2) supporting user-definable modalities, *e.g.*, via a type-class-like mechanism with optional user-defined semantics and solver plug-ins; (3) combining dependent types with graded modalities; and (4) integrating indexed modalities for guarded recursion.

References

[1] Flavien Breuvart and Michele Pagani. Modelling coeffects in the relational semantics of linear logic. In *CSL*, 2015.

[2] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *ESOP*, pages 351–370, 2014.

[3] Marco Gaboardi, Shinya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining Effects and Coeffects via Grading. In *ICFP*. ACM, 2016.

[4] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *ESOP*, pages 331–350, 2014.

[5] Jean-Yves Girard, Andre Scedrov, and Philip J Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97(1):1–66, 1992.

[6] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *ICFP*, pages 123–135, 2014.

[7] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified Static Analysis of Context-Dependence. In *ICALP (2)*, pages 385–397, 2013.