The Granule Project

Harley Eades III (PI Augusta University) and Dominic Orchard (PI University of Kent)
Augusta University Students: Preston Keel
University of Kent Students: Vilem-Benjamin Liepelt, Ed Brown, Ben Moon

In his talk, Preston [1] introduced the basic theory underlying graded linear logic and its alternate perspectives. His work is part of a larger project called the Granule Project[1]. The aim of the project is to develop the theory and practical implementation of graded linear type systems both in the external and internal forms. Thus far we have a core implementation of the Granule Language and several subprojects studying advanced extensions of the language underway. In this talk I will introduce the Granule Language, and summarize the subprojects the Augusta University team is currently working on.

1 The Granule Language

In this section we give the reader a quick taste of what Granule has to offer by giving several concrete examples. Each of the examples below are real Granule programs, and the reader is invited to install and try them out[2].

Linearity. Granule is based in linear logic, and by default every variable must be used exactly once. Thus, the following are both well-typed Granule programs:

```
id : ∀ {t : Type} . t → t          swap : ∀ {t : Type, s : Type} . (t, s) → (s, t)
id x = x                           swap (x,y) = (y,x)
```

However, the next two programs fail to typecheck, because they either use their input 0 times (left), or twice (right):

```
✗ drop : ∀ {t : Type} . t → ()      ✗ copy : ∀ {t : Type} . t → (t, t)
  drop x = ()                          copy x = (x,x)
```

In fact, the Granule interpreter `gr` gives us the following errors, respectively:

```
Linearity error:  2:1:              Linearity error:  2:10:
Linear variable x is never used.    Linear variable x is used more than once.
```

One traditional use case for linear types is in handling files, and Granule provides an interface to files, which includes the following operations:

```
openHandle  : forall {m : HandleType} . IOMode m -> String -> (Handle m) <IO>
readChar    : Handle R -> (Handle R, Char) <IO>
closeHandle : forall {m : HandleType} . Handle m -> () <IO>
```

Linearity then enforces that file handles cannot be used more than once. For example, in the program on the left the file handle `h` is only used one time – it appears as if it is used twice, but the variable `h` is being shadowed – however, the program on the right, uses the file handle $\mathtt{h}_1$ twice, and thus, fails to type check:

[1]The project webpage can be found at `https://granule-project.github.io/`
[2]Download and install Granule from here: `https://granule-project.github.io/granule.html`

```
twoChars : (Char, Char) <IO>                 bad : Char <IO>
twoChars = let                               bad = let
  h ← openHandle ReadMode "somefile";          h1 ← openHandle ReadMode "somefile";
  (h, c1) ← readChar h;                    ✗   h2 ← openHandle ReadMode "another";
  (h, c2) ← readChar h;                        () ← closeHandle h1;
  () ← closeHandle h                           (h1, c) ← readChar h1
  in pure (c1, c2)                             in pure c
```

Graded Modalities. We generalize linearity to graded modalities. The following are two examples where the input is allowed to be used any number of times:

```
drop' : ∀ {t : Type} . t [] → ()        copy' : ∀ {t : Type} . t [] → (t, t)
drop' [x] = ()                          copy' [x] = (x, x)
```

Thus, in Granule the linear logic bang-modality `!t` is denoted by `t []`. However, in Granule we can make use of full spectrum control over the structural rules:

```
drop'' : ∀ {a : Type} . a [0] → ()      copy'' : ∀ {a : Type} . a [2] → (a, a)
drop'' [x] = ()                         copy'' [x] = (x, x)
```

Here the type of `drop''` precisely requires that the input `x` be used `0` times, and the type of `copy''` requires that the input `x` be used exactly two times.

This is only a brief introduction to the features of Granule. During the presentation we will also present some more advanced features like data types, pattern matching, and different types of resource algebras apart from the natural numbers.

2 Team AU: Work in Progress

In this talk I will present an in-depth summary of the following subprojects:

Separation Logic. This project builds off of Preston's [1] work on the external/internal views of graded linear logic. Graded linear logic no matter the view is very quantitative, but hiding within the mathematical models of graded linear logic is a means of also supporting non-quantitative substructural logics like the logic of bunched implications. This new formalization of graded linear logic supports utilizing more than one resource-algebra at the same time, and it can be viewed as a merging of the external and internal views of graded linear logic.

Dependent Types. A long standing open problem in dependent type theory is how to reconcile the control of structural rules and dependent types. The combination of those features opens up a lot of interesting problems, for example how are variables managed in specifications? Graded systems overcome these problems by allowing precise control over how variables are managed in both specifications and programs. We will show that naively adding dependent types to linear logic results in a degenerate system, and then discuss how we over come these problems using the internal view of graded linear logic.

References

[1] Preston Keel and Harley Eades III. On the internal and external view of graded linear logic. Presented at the Southeast Regional Programming Languages Seminar (SERPL), 2019.