Gradual Typing for Annotated Type Systems

Luminous Fennell, Peter Thiemann

Department of Computer Science
University of Freiburg
Georges-Köhler-Allee 79
D-79110 Freiburg, Germany
{fennell,thiemann}@informatik.uni-freiburg.de

Abstract: Refinement type systems have been proposed by a number of researchers to sharpen the guarantees of existing type systems. Examples are systems that distinguish empty and non-empty lists by type, taint tracking and information flow control, dimension analysis, and many others. In each case, the type language is extended with annotations that either abstract semantic properties of values beyond the capabilities of the underlying type language (e.g., empty and non-empty lists) or they express extrinsic properties that are not locally checkable (e.g., taintedness, dimensions).

Gradual typing emerged as an approach to combine static and dynamic typing in a single language. Recent work considered a number of variations on gradual typing that are not directly related to dynamic typing, like gradual information flow, gradual typestate, and gradual effect systems. Instead of considering entire types as static or dynamic, these systems focus on gradualizing type refinements.

This proliferation of gradual systems begs the question if there is a common underlying structure. In this work, we give a partial answer by outlining a generic approach to "gradualize" existing annotated type systems that support annotations on base types.

We illustrate the usefulness of gradual annotation typing with the example of gradual dimension annotations. Type systems with dimensions prevent the programmer to mix up measurements of different dimensions that are represented with a common numeric type. For illustration we consider an ML-like language with simple types where numbers carry a dimension annotation. The following function, calculating an estimated time to arrival, is well-typed in this language.

```
fun eta (dist : float[m]) (vel:float[m/s]) : float[s] =
    dist / vel
```

The annotated type `float[u]` represents an integer of dimension `u` where `u` ranges over the free abelian group generated by the SI base dimensions: `m`, `s`, `kg`, and so on.

Each gain in safety costs flexibility. For example, a straightforward definition of the power function on meters fails to type-check in a system based on simple types.[1]

```
fun pow_m (x : float[m]) (y : int[1]) =
    if y == 0 then 1[S(1)] else x * pow_m x (y - 1)
```

[1] The annotation `S(1)` indicates a statically checked dimensionless number.

Polymorphism over dimensions does not help, because the dimension of the result depends on the parameter y as in `float[m^y]`. A gradual annotation for such functions avoids the complexity of a dependent type system and preserves some guarantees about the annotation. In our system, an implementation of the function `pow_m` could be provided with the gradual type `float[m] →int[1] →float[?]`. The annotation "`?`" marks the dimension of the result type as *dynamic* and indicates that the run-time system needs to check the consistent use of the dynamic dimension of the value. The programmer has to insert casts of the form $e : t \rightsquigarrow t'$, where t is the type of e and t' is the destination type. Casts only switch type annotations from static to dynamic or vice versa. They do not modify the underlying structure of the type. Here is the gradualized implementation of `pow_m`:

```
1 fun pow_mg (x : float[m]) (y : int[1]) =
2     if y == 0 then 1[D(1)]
3     else (x : float[m] ⇝ float[?]) * pow_mg x (y - 1)
```

The cast `x : float[m] ⇝float[?]` in line 3 converts `x` of type `float[m]` to destination type `float[?]` with a dynamic dimension. At run time, values of dynamic dimension are marked with a `D`, as illustrated in line 2. The dynamically annotated result can be reintegrated into statically verified code by casting the dynamic annotation to a static one:

```
fun volume (d : float[m]) : float[m^3] =
  (pow_mg 3 d) : float[?] ⇝ float[m^3]
```

For example, the expression `(pow_mg 3 2[m]) : float[?] ⇝ float[m^2]` evaluates to `8[D(m^3)] : float[?] ⇝float[m^2]`. As the computed dimension `D(m^3)` is incompatible to the expected dimension `m^2`, the cast fails and stops a computation with a potentially flawed result.

In our ESOP paper [FT14], we generalize the approach sketched above to arbitrary annotations $a \in A$ by giving an annotation Algebra $\mathcal{A}$ with carrier A that provides an operation $\oplus_A$ for each operation $\oplus$ on base types. The type system checks static annotations using $\oplus_A$ and enforces that values typed with dynamic annotations carry a value annotation $D(a)$ that can be checked at run time. We further extend the system described in the paper to annotations on type constructors like arrows, sums and products, and to polymorphic annotations.

For lack of space, we refer to our original paper on gradual annotation typing for a detailed discussion of prior work on gradual typing and annotated type systems, respectively [FT14].

References

[FT14] Luminous Fennell and Peter Thiemann. Gradual Typing for Annotated Type Systems. In Zhong Shao, editor, *ESOP'14*, Lecture Notes in Computer Science, Grenoble, France, April 2014. Springer.