

Algorithmic subtyping for (uniform) rate types

LUCAS DU, University of California, Davis, USA

DOOFENSCHMIRTZ INGRAM, The Couch, Catland, USA

1 A cleaner subtyping system (for uniform rates)

In implementing a prototype subtype checker, several things seem to be clear:

- We need to figure out how to handle *multiple refinements* properly.
- We should have a unified way to check the order between two rate refinements, regardless of window size. Relatedly, we need to figure out how to take the max, min, and sum of rate refinements that don't have a consistent window size.
- Our current system for handling crossovers for concatenation does not compose well and needs to be reworked. Specifically, simply taking the sum of the two rates in a concatenated pair of streams means that, for example, when we concatenate this concatenation with a third stream, we are summing *all three rates*, instead of *just the maximum of the crossover points*. The latter option seems to be much better and make more sense, at least from the perspective of what these rates are actually modelling.

In service of these goals, we propose a number of ideas:

- (1) We unify subtype checking on individual uniform rate refinements into one rule.
- (2) We give some idea of a semantics for streams with multiple refinements. In particular, we view such sets of refinements as *logical conjunctions* (in the spirit of Liquid Types, and other such refinement type systems).
- (3) We define what `max` and `min` mean on these sets of multiple refinements, and give a definition of a complete lattice for these types.
- (4) We define what `sum` means on these sets of multiple refinements. This `sum` operation, along with the set of rate refinements, trivially forms a commutative monoid. We also add these summed refinement types to our complete lattice.
- (5) We define decision procedures for subtyping over these rate refinements.
- (6) We suggest an idea for handling crossovers in concatenation in a composable way.

1.1 A unified rule for subtyping single rates

$$\frac{\text{LEQ-RATE} \quad (t_2 \leq t_1 \wedge n_1 \leq n_2) \vee (t_1 \leq t_2 \wedge n_1 \leq n_2 / (\lceil t_2/t_1 \rceil))}{n_1/t_1 \leq n_2/t_2}$$

$$\frac{\text{S-RELATIVE-RATE} \quad S_1 <: S_2 \quad n_1/t_1 \leq n_2/t_2}{(S_1)@n_1/t_1 <: (S_2)@n_2/t_2}$$

1.2 Handling multiple refinements

We treat multiple refinements as logical conjunctions composed of single rates. Our rate refinement types are all multiple refinements; single rate refinements are just a special case. There is a special Top type, `T`, which is the case where there are no rate limits and anything goes, and a special Bot

type, 0, which is the case where there is no rate at all (i.e. no events are allowed through). For example, a stream with rates @10/1s and @12/2s has rate refinement [$\text{@10/1s} \wedge \text{@12/2s}$].

Formally, we define this sort of “merging” as:

$$\frac{\begin{array}{c} \text{MULTIPLE-REFINE-MERGE-SUB} \\ n_1/t_1 \leq n_2/t_2 \end{array}}{((S)@[n_1/t_1])@[n_2/t_2] \doteq (S)@[n_1/t_1]}$$

$$\frac{\begin{array}{c} \text{MULTIPLE-REFINE-MERGE-NOSUB} \\ n_1/t_1 \not\leq n_2/t_2 \end{array}}{((S)@[n_1/t_1])@[n_2/t_2] \doteq S@[n_1/t_1 \wedge n_2/t_2]}$$

We also list the \top (which is just the absence of rate limits) and 0 subtyping rules:

$$\frac{\begin{array}{c} \text{S-BOT-SUB} \\ (S_1)@0 <: (S_2)@r \end{array}}{\text{S-TOP-SUP}}$$

where r is any rate refinement type. (There’s a bit of a change of terminology here, but r is just any rate refinement type as defined before: a logical conjunction of single rates n_i/t_i .) Note that S-BOT-SUB allows arbitrary bare stream types S_1 and S_2 without an explicit subtyping relation, as a rate refinement of 0 subtypes *any* other type (regardless of whether S_1 subtypes S_2).

Subtyping on these refinements is defined as follows (note how subtyping is like logical implication and subtyping between multiple rate refinements is like implication between logical conjunctions):

$$\frac{\begin{array}{c} \text{S-MULTIPLE-REFINE} \\ S_1 <: S_2 \quad \forall i : 1 \leq i \leq m, \exists j : 1 \leq j \leq n \text{ where } r_{1j} \leq r_{2i} \end{array}}{(S_1)@[r_{11}, r_{12}, r_{13}, \dots, r_{1n}] <: (S_2)@[r_{21}, r_{22}, r_{23}, \dots, r_{2m}]}$$

We will expand on this a bit later when we get to full decision procedures on types that include summed refinements.

1.3 max, min, and the rate refinement lattice

We now define `max` and `min` on these rate refinements. `max` is essentially the least upper bound of two refinements on the rate refinement lattice, while `min` is the greatest lower bound of two refinements (again, on the lattice). Refinements higher on the lattice are supertypes of refinements lower on the lattice. The idea for `max`: for all comparable pairs of single rates between both refinements, take the supertype; for each single rate (across both refinements) in the remaining single rates (that are, by definition, incomparable with any other single rates), convert all other remaining rates to that window size (using *supertype* conversions), and take the minimum of those (i.e. lowest event count); add each of these minimums to the resulting conjunction.

For `min`, things are a bit simpler. For all comparable single rates between both refinements, take the subtype; for all remaining incomparable rates, just add each of them directly to the resulting conjunction.

These are also the definitions of `glb` and `lub`. \top and 0 are, respectively, the top element and bottom element of the lattice.

1.4 Adding sums

Basically: for single rates that don’t have a common window size, just keep the `+` around. Only collapse that information when we need to, i.e. at the final step of subtype checking. When we do

the final subtyping check, we first check piecewise if each summand in the LHS (i.e. the possible subtype) subtypes a unique summand in the RHS. If not, then we must actually collapse: to do this, we try every possible case, based on common window sizes. Specifically, we iterate over every unique window size of rates on both the LHS and RHS—for each window size, we convert *all* single rates to that window size (taking the supertype conversion on the LHS and the subtype conversion on the RHS), add them up naturally (which we can do once we have a common window size), and then do the normal comparison. If any check passes, then the whole check passes; otherwise, if all checks fail, the whole check fails.

Alternatively, we can do this independently for the LHS and RHS, in the sense that we only take common window sizes on the LHS and RHS independently, sum for each of those window sizes, then try subtyping between all resulting pairs. For example, if the window sizes on the LHS are 1, 3, 6 and the window sizes on the RHS are 5, 7, 8, then we take overall sums with window size 1, 3, 6 on the LHS and overall sums with window size 5, 7, 8 on the RHS, then do subtyping comparisons on each possible combination (i.e. 1 vs. 5, 1 vs. 7, 1 vs. 8, 3 vs. 5, and so on). It is an interesting question (at the moment) to see which alternative is better, or if it doesn't matter. It appears to me, at first glance, that the first approach is always better, since we're doing the conversions individually for each single rate, instead of doing the conversion at the end, after we've already combined all the individual rates. But this can be proven.

We also need to add these sums to our complete lattice.

1.5 Decision procedures for subtyping on multiple, summed refinements

NOTE: This formulation of rate types still doesn't totally work out as we would like. Caleb put it really well: the reason we need to extend our rate types to *lists of rate types* is that this helps with the **composition problem**—during the implementation of subtyping, it turned out that checking subtypes required a kind of “global” information captured by this list of all the rates that were associated with a type.

NOTE: However, this still doesn't solve all our problems. Indeed, we still have problems with **composition**. Specifically, our rates as they stand don't form a complete lattice, which means we can't nicely take max/min (or least upper bound, greatest lower bound). We also don't have any sense of how to do addition (our “solution” above that just keeps the `+` around just delays the inevitable, since we still don't know exactly how to check subtyping between types with `+` in them), nor do we have any sense of how to do other possibly useful operations like concatenation. So the problem is that our rates don't compose well. But: all these operations we want, and that we want to compose well, look somewhat like an *algebra*—perhaps a boolean or a Kleene algebra. And that would be a very interesting direction to investigate, since modeling our rates as an algebra like this, with nice compositional properties and a rich tradition of various algorithmic and/or decidability results would be very, very helpful. (As an aside: if Kleene algebra does indeed end up being useful to us, wouldn't KARate be a funny name...?) In particular: some kind of (unary) timed automata?

NOTE: So: what we really need is a *an algebraic theory* of rate limits. At least, that's the goal. (Maybe they do actually form some kind of monoid!)