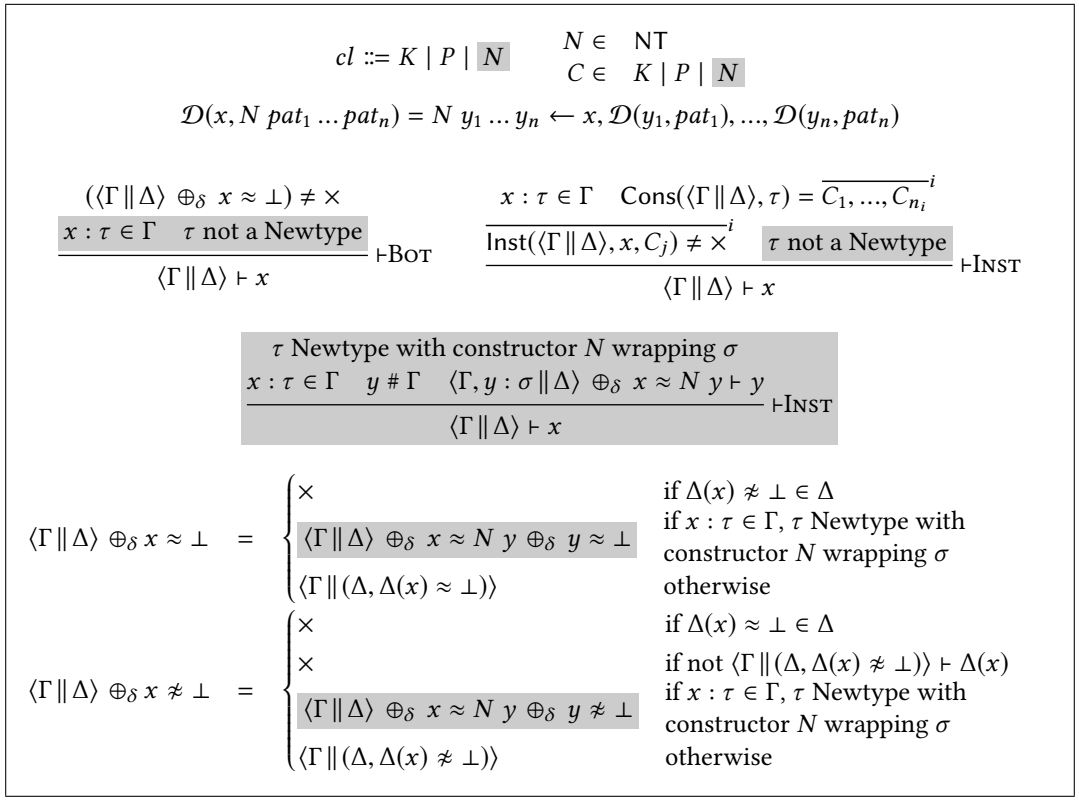

Fig. A.1. Extending coverage checking to handle Newtypes

A APPENDIX

A.1 Literals

The source syntax in fig. 1 deliberately left out literal patterns l. Literals are very similar to nullary data constructors, with one caveat: They don't come with a builtin `COMPLETE` set. Before section 4.5, that would have meant quite a bit of hand waving and complication to the $\vdash$ judgment. Now, literals can be handled like disjunct pattern synonyms (i.e. $l_1 \cap l_2 = \emptyset$ for any two literals l_1, l_2) without a `COMPLETE` set!

We can even handle overloaded literals, but will find ourselves in a similar situation as with pattern synonyms:

```
instance Num () where
  fromInteger _ = ()
n = case (0 :: ()) of 1 → 1; 0 → 2
```

Considering overloaded literals to be disjunct would mean marking the first alternative as redundant, which is unsound. Hence we regard overloaded literals as possibly overlapping, so they behave exactly like nullary pattern synonyms without a `COMPLETE` set.

A.2 Newtypes

Newtypes have strange semantics. Here are three key examples that distinguish it from data types:

newtype $N\ a = N\ a$		$f :: N\ \mathit{Void} \to \mathit{Bool} \to \mathit{Int}$
$g1 :: N\ () \to \mathit{Bool} \to \mathit{Int}$	$g2 :: N\ () \to \mathit{Bool} \to \mathit{Int}$	$f\ \_\ \ \mathit{True} = 1$
$g1\ !(N\ \_)\ \mathit{True} = 1$	$g2\ (N\ !\_)\ \mathit{True} = 2$	$f\ (N\ \_)\ \mathit{True} = 2$
$g1\ (N\ !\_)\ \mathit{True} = 2$	$g2\ !(N\ \_)\ \mathit{True} = 1$	$f\ !\_\ \ \mathit{True} = 3$

The definition of f is subtle. Contrary to the situation with data constructors, the second GRHS is *redundant*: The pattern match on the Newtype constructor is a no-op. Conversely, the bang pattern in the third GRHS forces not only the Newtype constructor, but also its wrapped thing. That could lead to divergence for a call site like $f\ \bot\ \mathit{False}$, so the third GRHS is *inaccessible* (because every value it could cover was already covered by the first GRHS), but not redundant. A perhaps surprising consequence is that the definition of f is exhaustive, because after $N\ \mathit{Void}$ was deprived of its sole inhabitant $\bot \equiv N\ \bot$ by the third GRHS, there is nothing left to match on.

?? outlines a solution (based on that for pattern synonyms for brevity) that handles f correctly. The idea is to treat Newtype pattern matches lazily (so compared to data constructor matches, $\mathcal{D}$ omits the $!x$). The other significant change is to the $\vdash$ judgment form, where we introduce a new rule $\vdash$NT that is specific to Newtypes, which can no longer be proven inhabited by either $\vdash$Inst or $\vdash$Bot.

But *g1* crushes this simple hack. We would mark its second GRHS as inaccessible when it is clearly redundant, because the $x \not\approx \bot$ constraint on the match variable x wasn't propagated to the wrapped (). The inner bang pattern has nothing to evaluate.

We counter that with another refinement: We just add $x \approx Ny$ and $y \not\approx \bot$ constraints whenever we add $x \not\approx \bot$ constraints when we know that x is a Newtype with constructor N (similarly for $x \approx \bot$). Both *g1* and *g2* will be handled correctly.

An alternative, less hacky solution would be treating Newtype wrappers as coercions and at the level of Δ consider equivalence classes modulo coercions. That entails a slew of modifications and has deep ramifications throughout the presentation.

A.3 Strictness

Instead of extending the source language, let's discuss ripping out a language feature, for a change! So far, we have focused on Haskell as the source language, which is lazy by default. Although after desugaring the difference in evaluation strategy of the source language becomes irrelevant, it raises the question of how much our approach could be simplified if we targeted a source language that was strict by default, such as OCaml or Idris.

First off, both languages offer language support for laziness and lazy pattern matches, so the question rather becomes whether the gained simplification is actually worth risking unusable or even unsound warning messages when making use of laziness. If the answer is "No", then there isn't anything to simplify, just relatively more $x \approx \bot$ constraints to handle.

Otherwise, in a completely eager language we could simply drop $!x$ from Grd and Bang from Ant. Actually, Ant and $\mathcal{R}$ could go altogether and $\mathcal{A}$ could just collect the redundant GRHS directly! Since there wouldn't be any bang guards, there is no reason to have $x \approx \bot$ and $x \not\approx \bot$ constraints either. Most importantly, the $\vdash$Bot judgment form has to go, because $\bot$ does not inhabit any types anymore.