Blame for Hybrid Typing

Peter Thiemann

Flanagan and others [7] introduce hybrid type checking as a framework that employs static type checking as much as possible and reverts to dynamic checking when typing constraints cannot be resolved statically. Their language supports refinement types like $x : \mathsf{int}\{x > 0\}$ for the set of positive integers and dependent function types like the type of strictly increasing functions $x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$ along with the standard notion of subtyping for dependent types [1].

The extra ingredient of a language with hybrid type checking is a cast expression that we write as $M : S \Rightarrow T$ for casting the value of M from source type S to target type T. For instance, a cast may further restrict an increasing function F so that it never returns a value greater than twice its input.

$$\begin{array}{rl} G = (F\ : & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}) \\ \Rightarrow & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x \wedge y < 2 * x\})) \end{array}$$

Applying the function G to a suitable argument, say 42, yields a term that applies a cast that is derived from the ranges of the original function type cast to the result of the function application:

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

The implementation of this cast checks the predicate $Q = y > 42 \wedge y < 2 * 42$ on the result $y = (F\,42)$ of the function call. But this check performs more work than strictly necessary because it ignores the static knowledge $P = y > 42$ from the precondition on y. We develop a framework to find a *delta predicate*, which is cheaper to check at run time than Q, but which is equivalent to Q when assuming P. In this particular example, a suitable delta predicate is $y < 84$.

We formalize a dependently typed blame calculus based on the ideas of hybrid typing. This calculus can be seen as an intermediate language in compiling a dependently typed language with refinement types: Subtyping constraints that can be discharged at compile time are eliminated and the remaining ones are reified as run-time type casts. This situation is similar as in the blame calculus considered by Wadler and Findler [14]. However, their base calculus is simply typed and their predicates are boolean-typed terms in the calculus. In contrast, our calculus is dependently typed and we employ a separate language of predicates (first-order predicate logic).

Following Wadler and others [9], we develop the corresponding coercion calculus and define a translation between the calculi that embodies the simplification motivated by the above examples. While our initial development happens in a setting with first order logic with abstract predicates in types, we subsequently instantiate abstract predicates to linear integer constraints and demonstrate that finding the simplified run-time checks amounts to a synthesis problem. We implemented this synthesis procedure using the Rosette system [12, 13] and performed some experiments.