Subtypes for Specifications*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. Specification languages are best used in environments that provide effective theorem proving. Having such support available, it is feasible to contemplate that typechecking can use the services of the theorem prover. This allows interesting extensions to the type systems provided for specification languages. I describe one such extension called "predicate subtyping" and illustrate its utility as mechanized in PVS.

1 Introduction

For programming languages, type systems and their associated typecheckers are intended to ensure the absence of certain undesirable behaviors during program execution [2]. The undesired behaviors generally include untrapped errors such as adding a boolean to an integer, and may (e.g., in Java) encompass security violations. If the language is "type safe," then all programs that can exhibit these undesired behaviors will be rejected during typechecking.

Execution is not a primary concern for specification languages, but typechecking can still serve to reject specifications that are erroneous or undesirable in other ways. A minimal expectation for specifications is that they should be consistent: an inconsistent specification is one from which some statement and its negation can both be derived; such a specification necessarily allows any property to be derived and thus fails to say anything useful at all. The first systematic type system (now known as the "Ramified Theory of Types") was developed by Russell [15] to avoid the inconsistencies in naive set theory, and a simplified form of this system (the "Simple Theory of Types," due to Ramsey [13] and Church [4]) provides the foundation for most specification languages based on higher-order logic. If a specification uses no axioms, typechecking with respect to such a type system guarantees consistency. The consistency of axioms cannot be checked algorithmically in general, so the best that a typechecker can do in the presence of axioms is to guarantee "conservative extension" of the non-axiomatic part of the specification (i.e., roughly speaking, that it does not introduce any new inconsistencies).

Since their presence weakens the guarantees provided by typechecking, it is desirable to limit the use of axioms and to prefer those parts of the specification

* This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.
language for which typechecking ensures conservative extension. Unfortunately, those parts are usually severely limited in expressiveness and convenience, often being restricted to quantifier-free (though possibly recursive) definitions that have a strongly constructive flavor; such specifications may resemble implementations rather than statements of required properties, and proofs about them may require induction rather than ordinary quantifier reasoning. Thus, a very worthwhile endeavor in design of type systems for specification languages is to increase the expressiveness and convenience of those constructions for which typechecking can guarantee conservative extension, so that the drawbacks to a definitional style are reduced and resort to axioms is needed less often.

In developing type systems for specification languages, we can consider some design choices that are not available for programming languages. In particular, a specification language will usually be part of an environment that includes an effective theorem prover, so it is feasible to contemplate that typechecking can rely on general theorem proving, and not be restricted to the trivially decidable properties that are appropriate for programming languages.

“Predicate subtypes” are one example of the opportunities that become available when typechecking can use theorem proving. I am an enthusiastic user of predicate subtypes—I consider them the most useful innovation I have encountered in type systems for specification languages—and the purpose of this paper is to share my enthusiasm. I will do so using simple examples to explain what predicate subtypes are, and to demonstrate their utility in a variety of situations.

2 Predicate Subtypes

A predicate subtype is, as its name suggests, a subtype characterized by some predicate or property. For example, the natural numbers are a subtype of the integers characterized by the predicate “greater than or equal to zero.” Predicate subtypes can help make specifications more succinct by allowing information to be moved into the types, rather than stated repeatedly in conditional formulas. For example, instead of

\[ \forall (i, j: \text{nat}): i \geq 0 \text{ and } j \geq 0 \implies i + j \geq i \]

we can say

\[ \forall (i, j: \text{nat}): i + j \geq i \]

because \( i \geq 0 \) and \( j \geq 0 \) are recorded in the types for \( i \) and \( j \).

Theorem proving can be required in typechecking some constructions involving predicate subtypes. For example, if \( \text{half} \) is a function that requires an even number (defined as one equal to twice some integer) as its argument, then the formula

\[ \forall (i: \text{int}): \text{half}(i+i+2) = i+1 \]

2 Another is consistency checking for tabular specifications [11].