ATS: A Language That Combines Programming with Theorem Proving

Sa Cui, Kevin Donnelly, and Hongwei Xi

Computer Science Department,
Boston University
{cuisa, kevind, hwxi}@cs.bu.edu

Abstract. ATS is a language with a highly expressive type system that supports a restricted form of dependent types in which programs are not allowed to appear in type expressions. The language is separated into two components: a proof language in which (inductive) proofs can be encoded as (total recursive) functions that are erased before execution, and a programming language for constructing programs to be evaluated. This separation enables a paradigm that combines programming with theorem proving. In this paper, we illustrate by example how this programming paradigm is supported in ATS.

1 Introduction

The framework Pure Type System (PTS) [1] offers a simple and general approach to designing and formalizing type systems. However, PTS makes it difficult, especially, in the presence of dependent types to accommodate many common realistic programming features, such as general recursion [7], recursive types [11], effects [10] (e.g., exceptions [9], references, input/output), etc. To address such limitations of PTS, the framework Applied Type System (ATS) [14] has been proposed to allow for designing and formalizing (advanced) type systems in support of practical programming. The key salient feature of ATS lies in a complete separation of the statics, in which types are formed and reasoned about, from the dynamics, in which programs are constructed and evaluated. With this separation, it is no longer possible for programs to occur in type expressions as is otherwise allowed in PTS.

Currently, ATS, a language with a highly expressive type system rooted in the framework ATS, is under active development. In ATS, a variety of programming paradigms are supported in a typeful manner, including functional programming, object-oriented programming [3], imperative programming with pointers [16] and modular programming. There is also a theorem proving component in ATS [4] that allows the programmer to encode (inductive) proofs as (total recursive) functions, supporting a paradigm that combines programming with theorem proving [5]. This is fundamentally different from the paradigm of extracting programs from proofs as is done in systems such as Coq [2] and NuPrl [6]. In ATS, proofs are completely erased before execution, while proofs in Coq, for example, are not. In addition, ATS allows the construction of programs involving
ATS: A Language That Combines Programming with Theorem Proving

Some formal syntax of ATS is shown in Figure 1. The language ATS has two components: the static component (statics) which includes types, props and type indices and the dynamic component (dynamics) which includes programs and proof terms. The statics itself is a simply typed language and a type in it is referred to as a `sort`. For instance, we have the following base sorts in ATS: `addr`, `bool`, `int`, `prop`, `type`, `view`, `viewtype`, etc. Static terms `L`, `B`, `I` of sorts `addr`, `bool` and `int` are referred to as static address, boolean and integer terms, respectively. Static terms `T` of sort `type` are types of program terms, and static terms `P` of sort `prop` referred to as props, are types of proof terms. Proof terms exist only to show that their types are inhabited (in order to prove constraints on type indices). Since the type system guarantees that proof functions are total, we may simply erase proof terms after type-checking. We also allow linear proof terms, which are assigned a view `V`, of sort `view`. Since it is legal to use non-linear proofs as linear ones, we have that `prop` is a subsort of `view` and `type` is a subsort of `viewtype`.

Types, props and views may depend on one or more type indices of static sorts. A special case of such indexed types are singleton types, which are each a type for only one specific value. For instance, `int(I)` is a singleton type for the integer equal to `I`, and `ptr(L)` is a singleton type for the pointer that points to the address (or location) `L`.

We combine proofs with programs using `proving types` of the form `(V | T)` where `V` and `T` stand for static terms of sort `view` and `type`, respectively. A proving type formed with a view is assigned the sort `viewtype`; if `V` can be assigned a prop then we can assign the proving type the sort `type`. We may