Improving Haskell Types with SMT

Iavor S. Diatchki
Galois Inc., USA
iavor.diatchki@gmail.com

Abstract

We present a technique for integrating GHC's type-checker with an SMT solver. The technique was developed to add support for reasoning about type-level functions on natural numbers, and so our implementation uses the theory of linear arithmetic. However, the approach is not limited to this theory, and makes it possible to experiment with other external decision procedures, such as reasoning about type-level booleans, bit-vectors, or any other theory supported by SMT solvers.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Constraints

Keywords Type systems, SMT, constraint solving

1. Introduction

For a few years now, there has been a steady push in the Haskell community to explore and extend the type system, slowly approximating functionality available in dependently typed languages [16–18]. The additional expressiveness enables Haskell programmers to maintain more invariants at compile time, which makes it easier to develop reliable software, and also makes Haskell a nice language for embedding domain specific languages [15]. The Haskell compiler is not just a translator from source code to executable binaries, but it also serves as a tool that analyzes the program and helps find common mistakes early in the development cycle.

Unfortunately, the extra expressiveness comes at a cost: Haskell's type system is by no means simple. Writing programs that make use of invariants encoded in the type system can be complex and time consuming. For example, often one spends a lot of time proving simple theorems about arithmetic which, while important to convince the compiler that various invariants are preserved, contribute little to the clarity of the algorithm being implemented [18].

Given that in many cases the proofs constructed by the programmers tend to be fairly simple, we can't help but wonder if there might be a away to automate them away, thus gaining the benefits of static checking, but without cluttering the program with trivial facts about, say, arithmetic. This paper presents a technique to help with this problem: we show one method of integrating an SMT solver with GHC's type checker. While we present the technique in the context of Haskell and GHC, the technique should be applicable to other programming languages and compilers too.

The approach presented in this paper uses similar tools—namely SMT solvers—to Liquid Haskell [14]. However, the design and overall goals of the two techniques are somewhat different and, indeed, it makes perfect sense to use both systems in a single project. Liquid Haskell enables Haskell programmers to augment Haskell definitions with refinements of the types, which may be used to verify various properties of the program. The refinement language is separate from the language of Haskell types—it is quite expressive, and is really aimed at formal verification. As such, a function will have its ordinary type and, in addition, a refinement type, which serves to specify some aspect of the behavior of the function. The checks performed by Liquid Haskell are entirely separate from the ones performed by GHC's type checker. This is quite different from the technique described in this paper, which shows how to integrate an SMT solver directly into GHC's type checker. The benefit of of extending GHC's type checker is that we don't need to introduce a separate language for specifications, instead we may reuse types directly. Ultimately, however, the two approaches are not at all mutually exclusive. The next section gives a brief flavor of what can be achieved using the techniques from this paper.

1.1 Examples

We illustrate the utility of the functionality provided by our algorithm with a few short examples. A very common example in this area is to define a family of singleton types that links type-level natural numbers to run-time constants that represent them:

```
data UNat :: Nat -> * where
  Zero :: UNat 0
  Succ :: UNat n -> UNat (n + 1)
```

Here we've used a unary representation of the natural numbers, and each member of the family, `UNat n`, has exactly one inhabitant, namely the natural number `n` represented in unary form. Because we are using a GADT, we can pattern match on the constructors of the type and gradually learn additional information about the value being examined. The kind `Nat` is inhabited by types corresponding to the natural number (e.g., `0`, `1`, ...), and `(+)` is a type-level function for adding natural numbers.

Next, we define a function to add two such numbers:

```
uAdd :: UNat m -> UNat n -> UNat (m + n)
uAdd Zero     y  = y
uAdd (Succ x) y  = Succ (uAdd x y)
```

While this is a simple definition, and we are unlikely to have gotten it wrong, it is nice to know that GHC is checking our work! Had we made a mistake, for example, by mis-typing the recursive call as `uAdd x x`, we would get a type error:

```
  Could not deduce (((n1 + n1) + 1) ~ (m + n))
    from the context (m ~ (n1 + 1))
```