Totality Checker for a Dependently Typed Language

MARTY STUMPF

1 INTRODUCTION

This is the reference document for the termination-checker repository, which is a totality checker for a dependently typed language implemented in Haskell. The totality checker checks for:

(1) strict positivity
(2) pattern matching coverage
(3) termination

To support these checks, the type checker has to support data type (hence the strict positivity) and function (hence the pattern matching coverage) declarations. I first describe the type checker without totality checks in section 2. Then I describe the mechanism for checking strict positivity in section 3. After that, I describe the mechanism for checking termination in section 4. Finally, I describe the mechanism for checking the patterns of a function cover all cases in section 5.

2 PRELIMINARIES (TYPE CHECKING WITHOUT TOTALITY CHECKS)

The type checker checks *data type* and *function* declarations. These declarations consist of *expressions.*

2.1 Expressions

An expression e is one of the following (as in ‘Types.hs‘):

$$
\begin{array}{rcrr}
e & = & \star & \text{universe of small types} \\
 & | & x & \text{variable} \\
 & | & c & \text{constructor name} \\
 & | & D & \text{data type name} \\
 & | & f & \text{function name} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & (x : A) \to B & \text{dependent function type} \\
 & | & e e_1 \ldots e_n & \text{application}
\end{array}
$$

2.2 Evaluation and Values

Because of dependent types, computation is required during type-checking. An expression is *evaluated* during type-checking to a *value*. (See ‘Evaluator.hs‘.)

Values may contain *closures*. A closure e^{ρ} is a pair of an expression e and an *environment* ρ. Environments provide bindings for the free variables occurring in the corresponding e.

Author’s address: Marty Stumpf, thealmartyblog@gmail.com.

2.2.1 *Values.*

$$
\begin{array}{rrr}
v = & v\ (v_1 \dots v_n) & \text{application} \\
\mid & Lam\ x\ e^{\rho} & \text{abstraction} \\
\mid & Pi\ x\ v\ e^{\rho} & \text{dependent function space} \\
\mid & k & \text{generic value} \\
\mid & \star & \text{universe of small types}
\end{array}
$$

$v\ (v_1 \dots v_n)$, $Lam\ x\ e^{\rho}$, and $Pi\ x\ v\ e^{\rho}$ can be evaluated further (see more below) while k, $\star$, c, f, and D are atomic values which cannot be evaluated further. A generic value *k* represents the computed value of a variable during type checking. More on this below.

The closures in $Lam\ x\ e^{\rho}$, and $Pi\ x\ v\ e^{\rho}$ do not have a binding for x. If there is no concrete value, a fresh generic value k would be the binding for x so that the closures can be evaluated.

2.2.2 *Evaluations.*

An expression e in environment ρ are evaluated as follows (as in Evaluator.hs):

$$
\begin{array}{rr}
eval\ (\lambda x.e)^{\rho} = & Lam\ x\ e^{\rho} \\
eval\ ((x : A) \to B)^{\rho} = & Pi\ x\ v_A\ B^{\rho} \\
 & \text{where } v_A = eval\ A^{\rho} \\
eval\ (ee_1 \dots e_n)^{\rho} = & app\ v\ v_1 \dots v_n \\
 & \text{where } v = eval\ e^{\rho}, v_i = eval\ e_i^{\rho} \\
eval\ (\star)^{\rho} = & \star \\
eval\ c^{\rho} = & c \\
eval\ f^{\rho} = & f \\
eval\ x^{\rho} = & \text{value of } x \text{ in } \rho
\end{array}
$$

Applications can be evaluated further as follows:

$$
\begin{array}{rr}
app\ u\ () = & u \\
app\ (u\ c_{11} \dots c_{1n})\ (c_{21} \dots c_{2n}) = & app\ u\ (c_{11} \dots c_{1n}, c_{21} \dots c_{2n}) \\
app\ (Lam\ x\ e^{\rho})\ (v, (v_1 \dots v_n)) = & app\ v'\ (v_1 \dots v_n) \\
 & \text{where } v' = eval\ e^{\rho, x=v} \\
app\ f\ (v_1 ... v_n) = & app_{fun}\ f\ (v_1 \dots v_n) \\
 & \text{if } f \text{ is a function} \\
app\ v\ (v_1 \dots v_n) = & v\ (v_1 \dots v_n)
\end{array}
$$

3 STRICT POSITIVITY CHECKS

4 TERMINATION CHECKS

4.1 Syntactic Checks

4.2 Type-based Checks

5 PATTERN MATCHING COVERAGE CHECKS