Exercise 8.1 Type coercions

Adding and comparing integers and reals can be allowed by introducing implicit conversions: Adding an integer and a real results in a real value, comparing an integer and a real can be done by first converting the integer into a real. Implicit conversions like this are called coercions.

1. Modify, in the theory Types, the inductive definitions of taval and tbval such that implicit coercions are applied where necessary.

2. Adapt all proofs in the theory Types accordingly.

Hint: Isabelle already provides the coercion function real_of_int (int ⇒ real).

Exercise 8.2 Security type system: bottom-up with subsumption

Recall security type systems for information flow control from the lecture. Such a type systems can either be defined in a top-down or in a bottom-up manner. Independently of this choice, the type system may or may not contain a subsumption rule (also called anti-monotonicity in the lecture). The lecture discussed already all but one combination: a bottom-up type system with subsumption.

1. Define a bottom-up security type system for information flow control with subsumption rule (see below, add the subsumption rule).

2. Prove the equivalence of the newly introduced bottom-up type system with the bottom-up type system without subsumption rule from the lecture.

\textbf{inductive} sec\_type2' :: "com ⇒ level ⇒ bool" ("(\_ ⇒ \_)") \([0,0] 50\) \textbf{where}

\textbf{Skip2'}: "\_ ⇒ SKIP : l"

\textbf{Assign2'}: "sec x ≥ sec a ⇒ \_ ⇒ x ::= a : sec x"

\textbf{Semi2'}: "[\_ ⇒ c₁ : l; \_ ⇒ c₂ : l] ⇒ \_ ⇒ c₁ ;; c₂ : l"

\textbf{If2'}: "[sec b ≤ l; \_ ⇒ c₁ : l; \_ ⇒ c₂ : l] ⇒ \_ ⇒ IF b THEN c₁ ELSE c₂ : l"

\textbf{While2'}: "[sec b ≤ l; \_ ⇒ c : l] ⇒ \_ ⇒ WHILE b DO c : l"
General homework instructions

All proofs in the homework must be carried out in Isar style.

Homework 8.1  Security type systems: bottom-up vs. top-down

Submission until Tuesday, December 12, 10:00am.

Prove the equivalence of the bottom-up system (\(\vdash \) : ) and the top-down system ( : ) without subsumption rule. Carry out a direct correspondence proof in both directions without using the \(\vdash'\) system.

lemma bottom_up_impl_top_down: “\(\vdash c : l \Rightarrow l \vdash c\)”
lemma top_down_impl_bottom_up: “\(l \vdash c \Rightarrow \exists l' \geq l. \vdash c : l'\)”

Homework 8.2  Explicit type coercions

Submission until Tuesday, December 12, 10:00am.

In the tutorial, we have introduced implicit coercions in the typing of arithmetic and boolean expressions. Here, we want to use explicit coercions. In particular, we want to

- add a new constructor \(\text{Real} :: \text{aexp} \Rightarrow \text{aexp}\) to \(\text{aexp}\),
- extend \(\text{taeval}\) to support this new constructor,
- extend \(\text{atyping}\) and \(\text{btyping}\) to return an expression with coercions inserted (which we will call an elaborated expression), and
- treat addition and comparison of different types as runtime errors in the semantics.

Copy and modify \(\text{Types}\) as necessary, including all proofs below (you don’t have to implement nor prove soundness for command elaboration).

Note: It is not recommended to keep the existing introduction and elimination rules, as they might make the automated tactics loop.

datatype \(\text{aexp} = \text{Ic} \; \text{int} | \text{Rc} \; \text{real} | \text{Real} \; \text{aexp} | \text{V} \; \text{vname} | \text{Plus} \; \text{aexp} \; \text{aexp}\)

Note that coercing an arithmetic expression of type \(\text{Rty}\) using \(\text{Real}\) should be considered a type error. Your elaboration implementation should add coercions where possible and necessary, but you’re free to insert them where they fit. For example, the expression \(\text{Plus} \; (\text{Rc} \; 0) \; (\text{Plus} \; (\text{Ic} \; 1) \; (\text{Ic} \; 2))\) could be elaborated to \(\text{Plus} \; (\text{Rc} \; 0) \; (\text{Real} \; (\text{Plus} \; (\text{Ic} \; 1) \; (\text{Ic} \; 2)))\).
**datatype** bexp = Bc bool | Not bexp | And bexp bexp | Less aexp aexp

**inductive** aelab :: “tyenv ⇒ aexp ⇒ aexp ⇒ ty ⇒ bool”
(“(1/ ⊢ (, → /))” [50,0,0,50] 50)

**inductive** belab :: “tyenv ⇒ bexp ⇒ bexp ⇒ bool” (“(1/ ⊢ (, → ))” [50,0,50] 50)

*Syntax examples:* \(\Gamma \vdash t \rightsquigarrow t' : \tau\) and \(\Gamma \vdash t \rightsquigarrow t' : \tau\)

You have to come up with the following lemma statements yourself.

**lemma** apreservation:

**lemma** aprogress:

**lemma** bprogress:

Is your \(aelab\) predicate deterministic? If yes, give an informal proof sketch, if no, give a counterexample.