Type Systems and Functional Programming

Type unification

The objective of this activity is the implementation of a **type unification** mechanism, which is an essential part of the type inference process. The latter will be implemented in the following activity.

Unification

As a first example, let us assume that we wish to identify a **common representation** for the types `(a, int)` and `([c], b)`, where the meaning of the parantheses are taken from Haskell, and `int` is the type of integers. Notice that, by **binding** the type variable `a` to type `[c]`, and the type variable `b` to type `int`, we get the common type `t = ([c], int)`, which still contains type variables. Here, `t` is the ***most general type*** resulting from the unification of the two expressions.

Thus, the unification process aims at determining a **set of bindings** of type variables to other types; this set is called a ***substitution***. In the example above, the result is `S = {[c]/a, int/b}` (`a` is bound to `[c]`, etc.). The substitution that leads to the most general common type of two expressions is called the ***most general unifier*** of the two expressions. We say that the types `a` and `b` **unify** under the substitution `S` iff, by performing the **replacements** dictated by the bindings, within both types, we obtain the **same type**: `a unify(S) b ⇔ a/S = b/S`, where `a/S` is the type obtained by performing the replacements from `S` within `a`.

Keep in mind that is it is possible to build **binding chains**, such as `{b/a, c/b, d/c, (e → f)/d, g/e}`, where the final binding of `a` is to `(g → f)`. Thus, in order to correctly unify types, and **distinguish** between their actual forms (free type variable, function type, etc.), the chain **ends** need to be explored. For example, the end of the chain of `a` is `(e → f)` — this is enough to point out that `a` is bound to a function type. The exploration may stop when either of the following is reached:

- a **variable free** within the substitution
- a **function type**.

The possible **situations** encountered during unification are enumerated below. The discussion holds w.r.t. chain **ends**.

- Two **free variables**, `a` and `b`, **always** unify (they may even be identical).
- A **free variable** and a **function type**, `a` and `(b → c)`, unify if `a` does **not** occur within either `b` or `c` (actually, within the types to which they are bound). Otherwise, **infinite** types would result. For avoiding this issue, an ***occurrence check*** must be performed.
- Two **function types**, `(a → b)` and `(c → d)`, unify if `a` unifies with `c`, and `b` with `d`.

Examples

The statement `{type/var}` reads: *the type variable `var` is bound to the type `type`*.

Type 1	Type 2	Initial substitution	Result	Final substitution	Comments
a	a	{}	True	{}	same variable
a	b	{}	True	{b/a} or {a/b}	2 free variables
a	b	{c/a}	True	{b/c, c/a} or {c/b, c/a}	2 free variables
a	(b → c)	{}	True	{(b → c)/a}	free variable and function type
a	(a → c)	{}	False	-	failed occurrence check

Type 1	Type 2	Initial substitution	Result	Final substitution	Comments
a	(b → a)	{ }	False	-	failed occurrence check
a	(b → c)	{a/b}	False	-	failed occurrence check

Implementation

Use your **monadic evaluator** as a starting point. The following are recommended **steps** for implementing the unification mechanism.

1. `Typing.Unification`: Implement the functions in in the order in which they are provided.

---

labs/unification.txt · Last modified: 2017/01/11 19:10 by Mihnea Muraru