KØBENHAVNS UNIVERSITET

Assignment 3 - PLD

Mikkel Willén

29. marts 2023

Indhold

1 Records

a)

I would suggest a record type that specified the name, data type and order of each field in the record. In general, if we have a type

```
record = {f_1 : e_1, ... , f_k : e_k}
```

a subtype of the record would contain all fields and maybe some additional fields.

```
subrecord = {f_1 : e_1, ... , f_k : e_k, f_i : e_i}
```

Specific fields in the record could then be accessed with the dot notation eg. person.name. Then if we try to access a field, that does not exist in the record such as person.weight, we would get an error message. This way we can prevent invalid operations on the record, and the program can then deal with the error as they see fit.

b)

A record type T_1 is a subtype of another record type T_2 if every field in T_2 is also in T_1 and that the respective fields have the same type. T_1 can then have additional fields, and i would then be safe to replace a type T_2 with its subtype T_1 since all accesible fields in T_2 is also in T_1.
This definition of subtyping is suitable because it is intuitive. It allows us to reason about subtyping in a natural way and to use subtyping to define more complex types, eg. we can use subtyping to define a type for a function that takes a record as an argmuent and returns a record that is a subtype of the argument record.

c)

The subtype ordering should be a pertial ordering and not a total ordering. A partial ordering is a binary relation that is reflexive, antisymmetric and transitive. A totalt ordering is a partial ordering that is also connex, meaning that for any two elements a and b either $a \leq b$ or $b \leq a$. In the case of record types, it is not always possible to compare two record types for subtyping, since we might want records for very different things.

d)

The hierarchy of types under the subtype ordering that we have defined has a greatest type and a smallest type. The greatest type is the empty record type, which has no fields. The empty record type is a subtype of every other record type, because it has no fields that could conflict with the fields of other record types. The smallest type is the record type that has all the fields of all other record types. For example

```
T1 = {name : string, age : int}
T2 = {age : int, employed : bool}
```

is the record type

```
SmallestType = {name : string, age : int, employed : bool}
```

2 Language constructs

The uniconditional construct is type-safe because it is equivalent to an if-then-else statement with an empty else branch. The declaration-as-statement construct is also type-safe because the declaration is treated as a statement and the scope of the declaration is limited to the block in which it appears. This means that the variable is only visible within that block and cannot be accessed outside of it.

3 Modules

When two modules contain identifiers with the same name, we can eg. use dot notation or something similar to specify which module to use. If for example both the arithmetic and statistics modules have a function called average, we can use the following to specify which module to use:

```
use Arithmetic
use Statistics
Arithmetic.average(x, y)
Statistics.average(x, y)
```

An advantages of this, is that we always know which module and function we use, and it is also a lot easier for readers of the code, to know what exactly is going on. A disadvantages that it is a bit more code, but most programming invironments these days have autocomplete functionality, so it does not really make a difference.
Another option is to use the function that is last defined. In

```
use Arithmetic
use Statistics
average(x, y)
```

it would be the average function from the Statistics module that is used. This a little bit faster to write, but the coder must always know which functions are in the modules and which is lastest defined. This also makes it a lot more dificult for readers to read, and they would in a lot of cases have to read up on the conventions of the programming language to be sure what module is used.

4 Haskell

a)

The function *tjop* takes a tuple of a list and an integer as inpout. It returns a list that contains the first n elements of the input list. The function uses the cons-operator : to prepend the first element of the input list to the result of the recursive call of the function with the tail of the input last and $n - 1$. The recursion stops when the second element of the tuple is 0 or the input list is empty. In either case the function returns an empty list. Some examples:

```
$ tjop ([1, 2, 3, 4, 5], 3)
[1, 2, 3]
$ tjop ([1, 2, 3, 4, 5], 0)
[]
```

b)

1. In the given code, the constants are [] and 0, the operator is : and the function is *tjop*. The type of [] is

   ```
   list<a>
   ```

   the type of 0 is

   ```
   int
   ```

   the type of : is

   ```
   a -> list<a> -> list<a>
   ```

   and the type of *tjop* is not yet know se we use a new type variable b to represent the output type.

2. We can propagate the types of the constants and the operator to the subexpressions of *tjop*. The first argument of *tjop* is a list of type a, where a is the same type variable as the one used for []. The second argument of *tjop* is an integer. The recursive call of *tjop* has the same type as *tjop*, which is $list < a > -> int -> list < b >$. The result of the cons opreation is a list of type $list < a >$, which is the same as the type of the input.

3. We can now use unification to find the most general type of *tjop*. We can use the following equations to represents the types:

```
list<a> = list<a>
int = int
tjop (list<a>, int) = list<b>
tjop (list<a>, n - 1) = list<b>
a = b
```

We can solve these equations by substituting b with a in the 3rd and 4th equations and we get:

```
forall a : (list<a>, int) -> list<a>
```

4. The resulting type of *tjop* does not contain any type variables, so we do not need to generealise the type, and therefore the most general type of *tjop* is

```
forall a : (list<a>, int) -> list<a>
```

5 Prolog

a)

istree is a predicate and *leaf* is a function symbol. Predicates are used to define relationships between terms in Prolog.

b)

```
T = node(bingo, node(abacus, leaf(minnie), ()), node(boing, (), node(fedtmule,
    leaf(plop). leaf(hello))))
```

c)

```
rightmost(leaf(X), X) :- istree(leaf(X))
rightmost(istree(node(Y, L, R), X)):- istree(leaf(R)), rightmost(R,X).
rightmost(istree(node(Y, L, empty)), X):-istree(leaf(L)), rightmost(L,X)
```

6 Hindley-Milner

$$T_1 = \text{real} \to \alpha_4 \text{ and } T_2 = (\text{string} \to (\text{int} \times \text{int list})) \to \alpha_5$$

Unification is not possible because the two types have different structures.

$$T_1 = \alpha_1 \to (\alpha_2 \to \text{real}) \text{ and } T_2 = \text{int} \to ((\alpha_3 \text{ list}) \to \text{real})$$

Unification is possible and is unified to:

$$\alpha_1 \to \text{int},\ \alpha_2 \to (\alpha_3 \text{ list})$$

$$T_1 = (\alpha_9 \times \alpha_8) \times \alpha_8 \text{ and } T_2 = (\alpha_7 \text{ list} \times (\alpha_{10} \text{ list})) \times \alpha_9$$

Unification is possible and is unified to:

$$\alpha_7 \to (\alpha_9 \times \alpha_8),\ \alpha_{10} \to \alpha_8$$

$$T_1 = \alpha_6 \text{ list and } T_2 = (\alpha_6 \text{ list}) \text{ list}$$

Unification is not possbile because the two types have different structures. The first type is a list of α_6 while the second type is a list of lists of α_6.

7 Fig. 9.2

a)

a_2 points to c, which is not the class descriptor of B nor is it null. c poinrts to C, which is not the class descriptor of B nor is it null. C points to B which is the class descriptor of B, so the call returns true.

b)

a_2 points to A which is not the class descriptor of B, and since A's point descriptor is null, the function returns false.