Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Compiler Engineering (2018)

Assignment 2 — Type Checking

Due: midnight Sunday 30 September

Overview

This assignment will extend the WHILE language compiler with *union types*. Union types have a long history in the research literature (see e.g. [1, 2, 3, 4]), and are now making their way into mainstream programming languages. For example, the Ceylon language developed by Redhat supports union types, as does Microsoft's TypeScript and the Whiley research language. Note also that, whilst union types appear superficially similar to sum types (e.g. as found in Haskell or ML), they are fundamentally different and strictly more powerful.

You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment. *We recommend that you start from a fresh version of the compiler, rather than your extension from assignment 1.*

Part 1 — Null (20%)

The `null` type is used to show the absence of something. It is distinct from `void`, as variables can hold the special `null` value (where as there is no special `void` value). The set of values defined by the type `null` is the singleton set containing exactly the `null` value. Values of `null` type support equality comparisons.

```
NullType ::= null
```

Example. The following illustrates a simple example of the `null` type:

```
null Null() {
    return null;
}
```

Although this method seems rather useless, it illustrates a key point: that `null` is both a type and a value in its own right.

What to do. This part of the assignment is a simple warm up exercise. You need to extend the parser, type checker and interpreter to support both the type `null` and the constant `null`. At this point, you should find the test cases provided in `NullTests` from the supplementary code now pass.

Part 2 — Unions (20%)

A union type is constructed from two or more component types and contains any value held in any of its components. For example, the type `null|int` is a union which holds either an integer or null value. The set of values defined by a union type `T1|T2` is exactly the *union* of the sets defined by `T1` and `T2`. Values of union type support equality comparisons.

```
UnionType ::= TermType ( | TermType )+
```

Example. The following example illustrates a union type:

```
// Return lowest index of matching item, or null if none
int|null indexOf(int[] items, int value) {
    int i = 0;
    while(i < |items|) {
        if(items[i] == value) {
            return i;
        }
        i = i + 1;
    }
    // item not found
    return null;
}
```

Here, a union type is used to construct a more expressive return value. If no matching element is found, `null` is returned (rather than e.g. `-1`).

What to do. Your goal now is to further extend the parser to support union types as given by the syntax above. This will require modifying the parser and adding a new subclass of `Type` to the Abstract Syntax Tree (AST). *You do not need to modify any other part of the compiler at this stage.* Having completed this, you should find that all tests in `UnionParsingTests` now pass.

Part 3 — Subtyping I (20%)

The addition of union types to WHILE presents a challenge because it introduces a distinction between the *syntactic* description of types and their underlying *semantic* meaning. In most programming languages (e.g. Java), this gap is either small or non-existent and, hence, there is little to worry about. The following illustrates this gap between the syntax and semantics of types:

```
int|null id(null|int x) {
    return x;
}
```

In this function we see two distinct *type descriptors* expressed in the program text, namely `int|null` and `null|int`. Type descriptors occur at the source-level and describe types which occur at the semantic level. In this case, we have two distinct type descriptors which describe the *same* underlying semantic type. We will often refer to types as providing the semantic (i.e. meaning) of type descriptors.

Equivalences. Since types are defined in terms of the values they represent, it is possible for two distinct type descriptors to describe the same underlying type. For example, `int|null` is considered equivalent to `null|int`. Whilst this case is fairly easy to spot, others are not so obvious. Some examples are given here to illustrate:

- `int|int` is equivalent to `int`
- `{int f,int g}|{int f}` is equivalent to `{int f}`
- `{int|null f}` is equivalent to `{int f}|{null f}`

You should think carefully about what other equivalences between types might exist and be sure to add test cases for them.

Subtyping Rules (Naive). The simplest approach to implementing subtyping with union types is based on the following subtyping rules, where $S \vee T$ is the mathematical notation for the WHILE type `S|T`:

$$\frac{T_1 \leq T_2}{T_1 \leq T_2 \vee T_3}\ [\text{S-UNION1}] \qquad \frac{T_1 \leq T_3}{T_1 \leq T_2 \vee T_3}\ [\text{S-UNION2}] \qquad \frac{T_1 \leq T_3 \quad T_2 \leq T_3}{T_1 \vee T_2 \leq T_3}\ [\text{S-UNION3}]$$

Whilst these rules are easy to implement, they are not *complete*. For example, they cannot be used to show that `{int|bool f}` is a subtype of `{int f}|{bool f}`.

What to do. The goal here is to extend the `TypeChecker` to implement subtyping of union types. You should begin by implementing the naive subtyping rules in the method `isSubtype()`. These are relatively straightforward, *though you will need to think carefully about the order in which they are applied.* Having implemented these rules, you should find that all but one of the tests in `UnionSubtypeTests` now pass.

Part 4 — Subtyping II (20%)

The goal now is to implement a *complete* notion of subtyping for union types. There are several ways to achieve this. The essential goal is to transform every type into a *normal form*. For example, record types can moved into a normal form by *unfactoring* them, that is by moving unions *outside* records. Thus, the normal form of `{int|bool f}` is `{int f}|{bool f}`. Note, however, that we cannot unfactor array types in this way since `{int|bool f}[]` is *not* equivalent to `{int f}[]|{bool f}[]`.

Having completed this component, you should find that all tests in `UnionSubtypeTests` now pass. *We strongly recommend that you add your own test cases to be sure you have consider all possibilities.*

Part 5 — Casts (20%)

The final part of this assignment is necessary to make union types actually useful. This is because, up to this point, there is no sensible way to use a union type. Specifically, we need a mechanism to convert from a union type (e.g. `int|null`) into a base type (e.g. `int`). Although there are variety of different ways we could do this, the simplest is to introduce the notion of a *cast*.

A *cast operator* accepts a value of one type $\mathtt{T}_1$ and returns a value of a different type $\mathtt{T}_2$, where either $\mathtt{T}_1 \leq \mathtt{T}_2$ or $\mathtt{T}_2 \leq \mathtt{T}_1$. This may result in a change of the underlying representation.

```
CastExpr ::= ( Type ) Expr
```

Example. The following illustrates a cast operator being used:

```
int extract(int|null x) {
    if(x != null) {
      return (int) x;
    } else {
      return 0; // default
    }
}
```

This examines a value of union type to see whether or not it contains the `null` value and, if not, returns the integer value, otherwise it returns a default value.

What to do. The goal here is to extend the parser, type checker, the definite assignment, the unreachable code checker and the interpreter to support cast expressions. Having done this, you should find that the tests in `CastTests` all now pass.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be based partly on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. In particular, they may constitute a more comprehensive set of test cases than given in the supplementary code.

You will also be required to submit a brief report outlining what you did, as in Assignment 1. Further details will be added later.

References

[1] Franco Barbanera and Mariangiola Dezani-Cian Caglini. Intersection and union types. In *Proceedings of the Conference on Theoretical Aspects of Computer Software (TACS)*, pages 651–674, 1991.

[2] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[3] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2), 2007.

[4] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–354, 2013.