Types and Programming Languages
Practical Exercise CW2

Security Types for a First-order Functional Language (Part 2)

School of Informatics
University of Edinburgh
`http://www.inf.ed.ac.uk/teaching/courses/tpl`

This is an **individual assessed practical exercise**. It will be awarded a mark out of 25. It is one of two assessed exercises in the Types and Programming Languages course. Each exercise is worth 10% of the final result for the module, when combined with the exam mark. You can expect to spend 8–10 hours on this exercise, plus any time required for reading. Please follow the instructions carefully: your work will only be only be marked if it is **submitted electronically** *exactly* according to the instructions below. The deadline for completing this practical is **6pm 16th March 2006**. The final page summarises submission instructions.

The practical work for this course is on a single topic split into two assessed exercises. The topic of the exercises is the design and implementation of a *security type system* for the first-order functional language FOFL which was introduced in lectures.

You may find it helpful to refer to the practical handouts for the first practical.

Part 2: Typechecking security levels

The task in this part is to implement a type checker for SFOFL, i.e., FOFL augmented with the security levels discussed in Part 1. The basis is the FOFL typechecker which you minimally extended in Part 1.

Important note: ideally you should base this work on your solutions to Part 1, which defined SFOFL. In case you had severe difficulty with the definition of the rules for SFOFL, please contact me and I will provide some specimen rules for you to work with. If you are reasonably happy with your own definitions from Part 1, I recommend that you implement those rather than ask for mine; full marks will be awarded here providing your code matches your definition and passes the test cases (or you document and justification any expected variation).

Marking note: your code will be marked primarily on functional correctness rather than issues of style, commenting, efficiency, etc. You are not expected to produce "expert" OCaml code, but should be able to mimic the constructs from the surrounding code in the FOFL implementation, and also use library functions where useful (the HTML manual of OCaml is at `http://www.caml.org`).

Algorithmic reformulation

The first task before implementing is to consider what the evaluation and typechecking algorithms must do. It's likely that both sets of rules will need adjusting for implementation, although hopefully not in a radical way.

Evaluation rules

In the case of the evaluation relation, the evaluation relation suggested had a notion of "target level". Any term inside a program has a defined target level set by its context (ultimately, the return level of the function it is defined inside). However, it is nice to be able to evaluate terms at top-level and have the implementation tell us what the (minimal) security level of the result is, without needing to specify it up front.

For example, if we enter the term[1]

```
  if false[10] then 0[0] else succ(4[5]) !:;
```

then the system might respond:

```
  5[5]
   : ??
```

Here, the evaluation has returned the result `5`, preserving the security level of the argument in `succ`. Recall from the `README` file supplied with the FOFL implementation that the `!:` annotation above means "evaluate without typechecking" — potentially risky, but useful for testing. In particular, we should be able to test cases like this:

```
prog
  def makemysecret(x:Nat[0]):Nat[10] = succ(x)
gorp

makemysecret(makemysecret(3)) !:;
```

Which produces the useful diagnostic output:

```
  Warning: evaluation stuck at function makemysecret, security level too high
  Warning: evaluation reached a stuck (non-value) term!
  makemysecret(makemysecret(3))
    : ??
```

If we try the same thing again, but this time *with* typechecking:

```
makemysecret(makemysecret(3));
```

then we get a static error as expected:

```
/home/da/sfofl/makemysecret.f:6.1:
Security level mismatch in application: expected type Nat but got type Nat[10]
```

This tests the contraposition of safety: any stuck term should be untypable.

[1]Notice in this example that the parser has been extended to allow stamped values, for ease of testing. It is suggested that you also do this, although the evaluator should check that labelled terms are in fact values.

Exercise. Implement your evaluation relation so that it can be tested as above. You will need to alter the same files as in Part 1, to first extend the abstract syntax and parser of the language, and then extend the evaluator. Because the evaluation function sometimes has a target level and sometimes not, one possibility is to add a `int option` argument to the `eval` function. [*15 marks*] (The marks allocated for above include 6 marks for correctly extending the syntax of the language).

Typechecking

The security type system for SFOFL includes a notion of subtyping. In case your definition of typing rules includes the characteristic *subsumption* subtyping rule T-SUB, you will need to reformulate the type system to turn it into an algorithm.

To reformulate your system, you need to consider where subsumption is really needed, and where it can be omitted to give a *minimal type*. Formally, the reformulation consists of giving a new set of rules without a general rule of subsumption, but with instances of subsumption incorporated into the premise of specific rules where necessary (see Chapter 16 of TAPL).[2]

Similarly to the evaluation relation, at certain points we will have to check that the security levels are correct, and at other points, infer the minimal levels. For one of the examples above,

```
  if false[10] then 0[0] else succ(4[5]);
```

the checker might respond

```
 5[5]
  : Nat[10]
```

Although the evaluation has stopped at level 5, the typechecker has assigned the level 10 to the term, because it occured inside the branch of a conditional testing a boolean of level 10. It should always be the case that the resulting security level of the value is dominated by the security level of the type; you should display a diagnostic error in case not, or instead feed the target security level assigned by typing into the evaluator, to get:

```
 5[10]
  : Nat[10]
```

Here's a slightly larger example:

```
prog
  def eq(x:Nat[10],y:Nat[10]):Bool[10] = ?
  def hash(x:Nat[0]):Nat[5] = ?
  def mysecret():Nat[10] = 123
  def comparehash(data:Nat[0],
                  expected:Nat[10]):Bool[10] = eq(hash(data),expected)
gorp

comparehash(97,mysecret());
```

which should display the checked program, then the result of checking and evaluating the term:

```
      eq(hash(97[0])[10],123[10])[10]
        : Bool[10]
```

[2]To be good and honest, one ought to also produce a proof that the reformulated system is equivalent to the original.

Notice that we have extended the notion of value here to include primitive functions (without definitions) applied to values; such values are also stamped during evaluation with the result level of the function, which explains the level 10 stamps outermost and on `hash(97[0])` above.

Exercise. Implement your algorithmic typechecking relation. Assuming you have already adjusted the syntax for programs, you will need to alter the file `core.ml` to change the typechecker, and perhaps the fule `main.ml` to adjust the top level and outer typechecking of programs. It might be also useful to add auxiliary functions to `syntax.ml`. [*10 marks*]

You should check that your typechecker works with some of the example programs shown in Part 1, and devise some tests of your own. The typechecker must:

1. accept all well-typed examples which obey explicit-flow constraints and allow subsumption (i.e. actual arguments passed to functions never have a security level exceeding the declared level, but may have a level below the declared level);
2. reject examples which break explicit-flow constraints with a reasonable error message;
3. behave consistently with your evaluator wrt other examples, i.e., if an example which violates indirect flow is typable, then it should also *not* get stuck in evaluation.

Your typechecker will be tested against a suite of examples to confirm the above!

Submission Instructions

This practical uses the School of Informatics standard electronic submission mechanism. The mechanism is based on the command-line program `submit`, which must be executed from your DICE account. The `submit` program takes a copy of the file you want to submit and puts it into a secure location for later retrieval by the marker.

It is recommend that you begin by making a copy of your submission from Part 1, with the command `cp -pr fofl sfofl`. Alternatively you can fetch a copy of FOFL with unit types from the course web page.

When your online work is ready, you should submit it by typing:

```
cd sfofl
make clean
cd ..
submit cs4 tpl cw2 sfofl
```

Later submissions will override earlier ones.

Please follow these instructions carefully, uniform filenames help us write test and print scripts to assist with marking. In particular, please do not alter any of the file names or function names in the starting code.

The deadline for completing this practical is **6pm 16th March 2006**.

David Aspinall
2nd March 2006
2006/03/01 12:29:39.