

Types and Programming Languages

Practical Exercise CW1

Security Types for a First-order Functional Language (Parts 0 & 1)

School of Informatics

University of Edinburgh

<http://www.inf.ed.ac.uk/teaching/courses/tp1>

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. The deadline for completing this practical is **6pm 18th February 2004**. The final page summarises submission instructions.

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

- Part 0 reviews some material covered in lectures, and introduces a supplied OCaml implementation of FOFL. The implementation uses the same framework as the typecheckers accompanying the course textbook (*TAPL*). To help understand the implementation, you are asked to extend it in an easy way. This stage is preparatory and should be regarded as part of the reading requirement for the course. Nevertheless, some marks are awarded for the extended program (electronic submission).
- Part 1 introduces the topic of the exercise, which is to invent a type system for ensuring *secure information flow* in programs. In Part 1 you are asked to design the rules of the type system and prove the central type safety property for the new language (paper submission).
- Part 2 will be issued in week 8. This will ask you to implement your design by extending your version of the FOFL typechecker from Part 0. Some further questions will be posed based on improvements to the system. This part will have a wholly electronic submission.

As usual, you should read through the whole document carefully before beginning work. Furthermore, although Parts 1 and 2 are separated into different exercises, it may help to experiment with implementation during Part 1 to help get the definitions right. But do not spend overly long on a polished program, since most marks are awarded for the written work in this exercise.

In case of questions or difficulties, please use the course newsgroup `eduni.inf.course.tp1` in the first instance.

Part 0. Background: FOFL and its implementation

Recall the language FOFL introduced in lectures. It is a first-order functional language with a succinct definition. As usual, we build up the language definition in three stages: (1) the abstract syntax; (2) the operational semantics, and (3) the typing rules. The typing rules are designed to rule out certain ill-behaved programs.

Syntax

The abstract syntax of FOFL is given by the grammar:

$$\begin{aligned} t & ::= \text{true} \mid \text{false} \mid 0 \mid \text{succ } t \mid \text{pred } t \mid \text{iszero } t \mid \text{if } t \text{ then } t \text{ else } t \\ & \quad \mid x \mid f(t_1, \dots, t_n) \\ T & ::= \text{Nat} \mid \text{Bool} \\ d & ::= \text{def } f(x_1 : T_1, \dots, x_n : T_n) : T = t \\ P & ::= \text{prog } d^* \text{ gorp} \end{aligned}$$

Where t is used for the syntactic category of *terms*, T is used for *types*, d for *function definitions* and P for programs. As usual, x ranges over a set of variables. We also use another disjoint category of identifiers, the function names, ranged over by f . By convention, a program is not allowed to define the same function name more than once.

From the grammar we can see that basic FOFL has a trivial, flat type structure. Every expression either denotes a natural number or a boolean value, so there are only two kinds of types. A FOFL program consists of a sequence of definitions for functions which may be freely (perhaps mutually) recursive. For example, here is a program defining functions which compute whether a given number is even or odd, by subtracting until we reach zero:

```
prog
  def iseven(x:Nat):Bool = if iszero x then true else isodd(pred x)
  def isodd(x:Nat):Bool = if iszero x then false else iseven(pred x)
gorp
```

There are more example programs included with the supplied implementation of FOFL.

Small-step operational semantics

To give a small-step semantics to FOFL, we need to define the syntactic category of *values*, a subset of the category of terms, ranged over by v :

$$\begin{aligned} v & ::= \text{true} \mid \text{false} \mid nv \\ nv & ::= 0 \mid \text{succ } nv \end{aligned}$$

The rules giving the operational semantics for FOFL appear in Table 1. Notice that for a given term, at most one evaluation rule can apply. The rule for evaluating the application of a function ensures that all of the arguments to the function are reduced to values first, from left to right. Once values are obtained for all arguments, the abstract machine simulates a function call by simultaneously substituting the argument values for the formal parameters in the function's definition.

$t \longrightarrow t'$	$\frac{t \longrightarrow t'}{\text{succ } t \longrightarrow \text{succ } t'}$	
$\frac{}{\text{if true then } t_1 \text{ else } t_2 \longrightarrow t_1}$	$\frac{}{\text{pred } 0 \longrightarrow 0}$	$\frac{P \text{ contains def } f(\vec{x} : \vec{T}) : T = t_f}{f(\vec{v}) \longrightarrow [\vec{x} \mapsto \vec{v}] t_f}$
$\frac{}{\text{if false then } t_1 \text{ else } t_2 \longrightarrow t_2}$	$\frac{t \longrightarrow t'}{\text{pred } (\text{succ } nv) \longrightarrow nv}$	$\frac{t_i \longrightarrow t'_i}{f(\vec{v}, t_i, \vec{t}) \longrightarrow f(\vec{v}, t'_i, \vec{t})}$
$\frac{t \longrightarrow t'}{\text{if } t \text{ then } t_1 \text{ else } t_2 \longrightarrow \text{if } t' \text{ then } t_1 \text{ else } t_2}$	$\frac{t \longrightarrow t'}{\text{pred } t \longrightarrow \text{pred } t'}$	$\frac{}{\text{iszero } 0 \longrightarrow \text{true}}$
	$\frac{}{\text{iszero } (\text{succ } nv) \longrightarrow \text{false}}$	
	$\frac{t \longrightarrow t'}{\text{iszero } t \longrightarrow \text{iszero } t'}$	

Table 1: FOFL Evaluation Rules (in program P)

Typing

The typing rules for FOFL are shown in Table 2. Because terms may contain variables, we need to perform type-checking in a *context* which declares the types of variables. Contexts are lists of bindings of types to variables:

$$\Gamma ::= \emptyset \mid \Gamma, x : T$$

As usual, we follow the convention that any context that appears in a valid typing judgement must not bind the same variable more than once.

The rules define two typing judgements:

$$\begin{array}{ll} \Gamma \vdash t : T & \text{term } t \text{ has type } T \text{ in context } \Gamma \\ \vdash P & P \text{ is a well-typed program} \end{array}$$

Notice that the second definition depends on the first, but the definition of $\Gamma \vdash t : T$ assumes that a particular program P has been given to extract the type information from function definitions.

Properties of FOFL

Recall from lectures that FOFL satisfies the syntactic characterisation of type safety. Given a well-typed FOFL program, we want to show that evaluating a closed, typable term will end with a value. This is given by the combination *SAFETY* = *PROGRESS* + *PRESERVATION*.

Theorem 1 (Progress) *If $\vdash t : T$ then either t is a value, or there exists a t' such that $t \longrightarrow t'$.*

Theorem 2 (Preservation) *If $\vdash t : T$ and $t \longrightarrow t'$ then $\vdash t' : T$ too.*

You should make sure that you can reproduce the detailed proofs of these theorems as sketched in lectures. For Progress, the *canonical forms* lemma is useful. For Preservation, the *substitutivity* of the typing judgement is useful. If Preservation is generalised to hold in a context Γ (which

<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\Gamma \vdash t : T$</div> $\frac{}{\Gamma \vdash \text{true} : \text{Bool}}$ $\frac{}{\Gamma \vdash \text{false} : \text{Bool}}$ $\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$ $\frac{}{\Gamma \vdash 0 : \text{Nat}}$ $\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{succ } t : \text{Nat}}$ $\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{pred } t : \text{Nat}}$ $\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{iszero } t : \text{Bool}}$	$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$ $\frac{P \text{ contains } \text{def } f(\vec{x} : \vec{T}) : T = t_f \text{ for each } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash f(\vec{t}) : T}$ <div style="border: 1px solid black; padding: 2px; margin: 10px auto; width: 50px; text-align: center;">$\vdash P$</div> $\frac{\text{for each } \text{def } f(\vec{x} : \vec{T}) : T = t_f \text{ in } P, \vec{x} : \vec{T} \vdash t_f : T}{\vdash P}$
---	--

Table 2: FOFL Typing Rules (for program P)

is necessary for the induction to go through when binding terms such as λ -abstraction or `case` constructs are included), the *thinning* lemma (or *weakening* and *permutation* separately) are useful. In this case, for the typing judgement which checks FOFL programs, a *renaming* lemma may be used to make sure that the variables x_i are disjoint from variables already in the context.

Implementation of FOFL

You are supplied with an implementation of FOFL, available from the course web page:

<http://www.inf.ed.ac.uk/teaching/courses/tpl/fofl.tar.gz>

You should download this file and unpack it with

```
tar xpf fofl.tar.gz
```

This will make a directory `fofl` which contains the OCaml code. On a Linux system with OCaml installed you should be able to type `make` to compile the typechecker. To run it on one of the example files use a command like this:

```
./f example-evenodd.f
```

You should examine the implementation in the files `syntax.ml` and `core.ml` carefully, especially if you have not studied the typechecker code accompanying the course textbook TAPL so far. It will be useful to read (the short) Chapter 4, and in case of difficulties with the language, to refer to OCaml resources on the Web at <http://www.caml.org>.

You should be able to relate the implementation to the formal description of the FOFL language above. Can you find where the conventions about non-duplication of variables and function names are enforced? How is simultaneous substitution defined and where is it used? How is the FOFL program declared in a file made into a parameter for the typechecker?

Warm-up exercise

Extend the FOFL language by adding some *structured types* chosen from those studied in lectures. At a bare minimum, you should add a type constructor for *unit* types (see Chapter 11 of TAPL). Recall that for unit types, the syntax is extended thus:

$$\begin{array}{l} t ::= \dots \mid \mathbf{unit} \\ T ::= \dots \mid \mathbf{Unit} \\ v ::= \dots \mid \mathbf{unit} \end{array}$$

There are no new evaluation rules, **unit** is already a value; it is the unique and only value of the type **Unit** so it has no destructor. The single new typing rule is:

$$\frac{}{\Gamma \vdash \mathbf{unit} : \mathbf{Unit}}$$

To make this simple extension, you will need to change several files. Specifically, you should

1. Modify the file `syntax.ml` which defines the abstract syntax, including the auxiliary functions to manipulate it. Notice that the datatypes are revealed in the interface file `syntax.mli` so you must repeat your extensions there.
2. Modify the files `lexer.mll` and `parser.mly` which are inputs to the parser generator. In `lexer.mll` you need to add new keywords following the pattern for the others, and in `parser.mly` you need to add new productions for `ATerm` and `AType`.
3. In `core.ml` you should extend the evaluation function and the type-checking function.

If you have difficulty doing this, you can cheat by referring to the full implementation of simply-typed lambda calculus which is supplied with TAPL (the type checker is called `fullsimple`). Try to resist the temptation to copy the code without having thought about it first!

Your work on this warm-up exercise should be submitted electronically using the `submit` command. Inside your working `fofl` directory you should issue the command `make clean` to remove generated files and backup files. Then submit the entire directory directly under the name `fofl`. See the final page for the actual command you should type.

NB: if you add further structural types beyond unit types, it might be sensible to make a copy of the implementation with just unit types to avoid making your job harder in Parts 1 and 2.

[8 marks]

At the same time as adding unit types, it would be useful to extend the language with a mechanism for *sequencing*. The term $t_1; t_2$ is evaluated by evaluating t_1 followed by t_2 . The result of evaluating t_1 is discarded, so its type should be **Unit**. In TAPL (p.120), Pierce shows that this can be treated as a new construct in the language, or as the *abbreviation* $(\lambda x : \mathbf{Unit}. t_2)t_1$. FOFL doesn't have λ -terms, but it is still possible to encode $t_1; t_2$. Can you see how? The answer is revealed further below.

Part 1: Designing Security Types

When writing security-critical applications, it is wise to keep in mind which pieces of data should be regarded as confidential and which it is safe to reveal. In *multi-level security systems* a series of levels are used to classify data and owners. Typical levels (e.g. used by the military) are:

$$\text{unclassified} \leq \text{confidential} \leq \text{secret} \leq \text{topsecret}$$

A person with only confidential clearance cannot read secret or top secret documents; this is an example of the *no read up* (or simple-security) condition. To prevent information flowing in the wrong direction, we should also prevent somebody with top-secret clearance writing a secret document, lest they let some top-secret facts escape unwittingly. This is an example of the *no write down* condition (also known as the **-property*). Of course, the no write down condition is rather too strong in practice, so some controlled method of releasing information or *declassification* is needed.

Multi-level security has typically been implemented using run-time monitoring to enforce access control and prevent information leakage. But we can do better by programming within a simple, efficient and static type system.

Adding security levels to FOFL

We will add security levels by labelling types with natural numbers. A new syntactic category of *security-annotated types* has the single production:

$$S ::= T[l]$$

where l is a natural number, the security level. Larger numbers represent higher security levels. As a shorthand, level 0 types (which classify public data) can have their labels omitted, so we may write just `Nat` instead of `Nat[0]`, for example.

We want to prevent information flowing from high levels to lower ones. In FOFL, information is passed around by calling and returning from functions. Function arguments are *data sinks* and function results are *data sources*. For example, some basic IO functions might be given unclassified typings:¹

```
def putnum(x:Nat[0]):Unit[0] = ?
def getnum():Nat[0] = ?
```

(writing the level 0 just to be explicit). The function `putnum` acts as a data sink, and the function `getnum` is a data source.²

We can give some more interesting typings to functions that should be aware of security levels. For example,

```
def genkey():Nat[10] = ?
def encrypt(key:Nat[10],plaintext:Nat[2]):Nat = ?
def decrypt(key:Nat[10],ciphertext:Nat):Nat[2] = ?
```

¹The supplied implementation of FOFL allows function bodies to be omitted by giving `?` as above. This allows us to assume external or *foreign* functions might be invoked with the types that we have declared.

²In a *pure* functional language, `getnum` would have to be a rather boring constant function; in impure languages like OCaml, it could refer to some global state, for example, the current position in the input stream, and return a different result on successive invocations. From the point of view of our type system, we can be pure or impure.

Keys are very sensitive so have a high security level. The plaintext passed to the encryption function is also sensitive, so has a non-zero security level, but it is perhaps less critical than the key. The result of encryption is public: after all, it should encode the data so that it may be safely transmitted across an unsecured (public) channel

The following terms should be typable: `decrypt(genkey(),3762)` (okay since the levels and types match exactly); `encrypt(genkey(),1234)` and even `encrypt(7546,1234)` (it is permitted to use low security values where high ones are required: this is “reading down”).

Programs with security flaws

Some terms should not be typable. For example, this function

```
def stupidshowsecret (key:Nat[10]):Unit = putnum(key)
```

type-checks without security levels but should be prohibited for violating security: it passes a high security data value to a public sink. This is an example of an *explicit* information flow which violates the security policy on levels.

Here is a rather more sophisticated attempt to reveal a key:

```
def seq(x:Unit,y:Unit):Unit = unit

def showsecret (key:Nat[10]):Unit =
  if iszero(key) then putnum(0) else
    seq(if isodd(key) then putnum(1) else putnum(0),
        showsecret(divide(key,2)))
```

This time `showsecret` prints out the key in (reversed) binary. We should prevent this! However, the calls to `putnum` now take ordinary numbers as arguments. This is an example of an *implicit* information flow which occurs because the low-security function calls occur inside the branch of a test on a high-security value.

How might the type system reject this example? First, notice that to type `showsecret` we expect that `isodd` will reveal some information flow. In the new language we must give security levels to every function argument and result. Since `isodd` leaks information from input to output, its result type should have the same level as its input argument type, for example:

```
def isodd(x:Nat[10]):Bool[10] = ...
```

(whereas `isodd` with a result level of `Bool[0]` should be prohibited). In general, even a zero test on a high-security value reveals partial information, so the term

```
if iszero x then putnum(0) else putnum(1)
```

should not be allowed in a context where `x` has a higher security level than the argument of `putnum`. So to prevent this, the type system needs to bear in mind the level of the guard when the branches of a conditional are examined.

Discussion Of course, an honest programmer isn't likely to accidentally write the kind of code like the `showsecret` function! If we can show that deliberately clever attempts like this are caught by the type system, perhaps we increase confidence that even “accidentally clever” attempts are also caught. More importantly, if we are using our type system to provide *certification* for mobile code executed elsewhere, we have to suppose a malicious agent will try to be clever and defeat the type system by supplying specially designed attack code. But in general, as well as thinking hard about

code examples which should be rejected, it may be useful to have a proof that characterises that a type system is “good enough” and rejects enough of the bad programs we want it to.

Exercise

Based on the sketched requirements above, design an extension to the formal definition of FOFL to encode security levels. Complete the following steps to define the new language SFOFL (secure FOFL):

1. Give a complete definition of a suitable syntax, operational semantics and type system for SFOFL. The operational semantics might keep track of the current security level of the term that is being evaluated, monitoring for security violations and getting stuck if one occurs. The typing system should reject the bad examples shown above, but accept programs which do not contain illegal flows. (**Hint:** first consider explicit flows, then implicit flows; you may need special extensions for the latter). [10 marks]
2. Justify why your typing rules allow the good examples shown above but reject the bad ones. Give an informal explanation of rules which take special account of the security levels. [2 marks]
3. In terms of your operational semantics, give an outline proof of the type safety of SFOFL, proving progress and preservation theorems. In particular, you should show representative cases in each theorem, and include the cases for function applications and variables, and any special rules you introduce. [5 marks]

Here is an optional extra question:

4. Consider the function `maybeshowsecret` defined thus:

```
def maybeshowsecret (key:Nat [10]):Unit =
  if iszero(key) then putnum(0) else
  seq(if isodd(key) then putnum(1) else unit,
      maybeshowsecret(divide(key,2)))
```

In what way may this function sometimes reveal the secret? Ideally it should be rejected by the SFOFL typing scheme. If your system forbids this example, explain with reference to your operational semantics for SFOFL why it can be that a type system can be considered to be *more accurate* than dynamic checking.

Additional resources. To give you ideas about the answers, you may like to study TAPL Chapter 15, or find a research paper on security typing. Two good starting places for research on security typing are:

Chapter 6 of *Secure Communicating Systems*, Michael R A Huth, CUP 2001.

Language-Based Information-Flow Security, Andrei Sabelfeld and Andrew C. Myers, IEEE Journal on Selected Areas in Communications, Vol 21, No 1. January 2003.

The second reference is available at <http://www.cs.chalmers.se/~andrei/jsac.pdf>. Some additional hints may be made available on the course web page.

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.

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

```
cd fofl
make clean
cd ..
submit cs4 tpl cw1 fofl
```

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.

Your offline submission should be handed in to the ITO, clearly labelled with *Types and Programming Languages Practical 1* and with your name and matriculation number.

The deadline for completing both parts of this practical is **6pm 18th February 2004**.

David Aspinall
4th February 2005
2005/03/02 20:50:31.