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/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. The deadline for completing this practical is **6pm 24th February 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 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.tpl` 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:

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

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:

$$
\begin{array}{lcl}
v & ::= & \texttt{true} \mid \texttt{false} \mid nv \\
nv & ::= & 0 \mid \texttt{succ}\ nv
\end{array}
$$

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.

$\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{\texttt{succ}\ t \longrightarrow \texttt{succ}\ t'}$$

$$\frac{}{\texttt{if true then}\ t_1\ \texttt{else}\ t_2 \longrightarrow t_1}$$

$$\frac{}{\texttt{pred}\ 0 \longrightarrow 0}$$

$$\frac{P \text{ contains } \texttt{def}\ f(\vec{x} : \vec{T}) : T = t_f}{f(\vec{v}) \longrightarrow [\vec{x} \mapsto \vec{v}]\, t_f}$$

$$\frac{}{\texttt{pred}\ (\texttt{succ}\ nv) \longrightarrow nv}$$

$$\frac{}{\texttt{if false then}\ t_1\ \texttt{else}\ t_2 \longrightarrow t_2}$$

$$\frac{t \longrightarrow t'}{\texttt{pred}\ t \longrightarrow \texttt{pred}\ t'}$$

$$\frac{t_i \longrightarrow t_i'}{f(\vec{v}, t_i, \vec{t}) \longrightarrow f(\vec{v}, t_i', \vec{t})}$$

$$\frac{}{\texttt{iszero}\ 0 \longrightarrow \texttt{true}}$$

$$\frac{t \longrightarrow t'}{\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \longrightarrow \texttt{if}\ t'\ \texttt{then}\ t_1\ \texttt{else}\ t_2}$$

$$\frac{}{\texttt{iszero}\ (\texttt{succ}\ nv) \longrightarrow \texttt{false}}$$

$$\frac{t \longrightarrow t'}{\texttt{iszero}\ t \longrightarrow \texttt{iszero}\ t'}$$

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:

$\Gamma \vdash t : T$	term t has type T in context Γ
$\vdash P$	P is a well-typed program

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

$\boxed{\Gamma \vdash t : T}$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}$$

$$\frac{}{\Gamma \vdash 0 : \texttt{Nat}} \qquad \frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ t : \texttt{Nat}} \qquad \frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ t : \texttt{Nat}} \qquad \frac{\Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ t : \texttt{Bool}}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{P \text{ contains } \texttt{def}\ f(\vec{x} : \vec{T}) : T = t_f \quad \text{for each } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash f(\vec{t}) : T}$$

$\boxed{\vdash P}$

$$\frac{\text{for each } \texttt{def}\ f(\vec{x} : \vec{T}) : T = t_f \text{ in } P,\ \vec{x} : \vec{T} \vdash t_f : T}{\vdash P}$$

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:

$$\begin{array}{lcl} t & ::= & \cdots \mid \texttt{unit} \\ T & ::= & \cdots \mid \texttt{Unit} \\ v & ::= & \cdots \mid \texttt{unit} \end{array}$$

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:

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

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* $(\lambda x : \texttt{Unit}.\, t_2)t_1$. 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:

$$\mathsf{unclassified} \;\leq\; \mathsf{confidential} \;\leq\; \mathsf{secret} \;\leq\; \mathsf{topsecret}$$

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] = ?
```

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

[1] The 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.

[2] In 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.

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 accidently 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 "accidently 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.

Analysing the Problem

In the next few pages we work towards a specific solution to the problem of how to add security types to FOFL. This solution is restricted and solves only the *explicit flow* case. For an extra challenge, try to find your own solution before reading the suggestions below, or try to extend the given solution to cover implicit flows too.

Understanding security levels

An initial question after considering the examples above should be: *where do security levels arise?* The only place that interesting security levels are introduced in the examples is as a result of applying functions, for example with the definition:

```
def genkey():Nat[10] = ?
```

then `genkey()` stands for a high-security natural number.

Sometimes it's necessary to put secrets into programs; we might like to do that with a function definition such as:

```
def mysecret():Nat[10] = 12345
```

then `mysecret()` is also a high-security number. But now the usual operational semantics is entitled to rewrite `mysecret()` to 12345 and there is a risk of losing its secret status!

For this reason, we shall introduce **stamped values** which are values "stamped" with a security level. Here is an evaluation with a stamped value:

```
succ(mysecret()) --> succ(12345[10]) --> 12346[10]
```

Notice that `succ` preserves the stamp. Stamps are the key to tracking levels dynamically.

A second question is *when are security levels enforced?* Again, the labels on functions are the only points where this happens in the examples. If we stamp values before invoking functions, we can check that the stamps are compatible with the declared levels. Moreover, stamped values should survive being passed as arguments, e.g., consider the program and sample evaluation:

```
def eq(x:Nat[10],y:Nat[10]):Bool[10] = ? /* secure equality test */
def hash(x:Nat[0]):Nat[5] = ?            /* secure hash of argument */

def comparehash(data:Nat[0],
                expected:Nat[10]):Bool[10] = eq(hash(data),expected)

        comparehash(12971238123,mysecret()) -->
        comparehash(12971238123[0],12345[10]) -->
        eq(hash(12971238123[0]),12345[10]) -->
        eq(H[5],12345[10]) -->
        eq(H[10],12345[10]) -->
        false[10]
```

where the final two steps are outwith the language (i.e. relying on the external primitives `eq` and `hash`; here, `H` stands for some number). The evaluation gives a hint about a suitable evaluation process for function application.

Notice that the term

```
comparehash(mysecret(),mysecret())
```

should get stuck, because the first argument reduces to a value stamped with the security level `10` rather than the expected level `0`. On the other hand,

```
comparehash(12971238123,12345)
```

would make progress: unstamped integers can be stamped with an arbitrary level to match the desired argument level.

The final question concerning security levels is, *when do security levels change?* The idea here is that security levels may change whenever information flows, for example, such as when computing a partial result. If we multiply a high-security integer by a low security one, we should get a high-security integer. The simple case of `succ` above showed that already.

However, user-defined functions are different: there is no assumption that information can flow between the inputs,[3] nor is there an assumption that information flows from function input to function output: that must be reflected in the levels which are checked by the type system.

There are two possibilities for information flow. High level inputs can be *disconnected* from the output suggesting that no flow happens, e.g., the constant function:

```
def mynumber(key:Nat[10]):Nat[0] = 5
```

should be type safe, because indeed there is no information flow from the high-security input to the result. Alternatively, the argument may appear in a position where it contributes to the result, such as:

```
def myothernumber(key:Nat[10]):Nat[0] = succ(key)
```

and this function should certainly *not* be type safe! (However, with a return security level of 10 it would be okay). On the other hand, it is safe to pass a low-security source to a a higher security sink: this is "write up", it simply assures us that the value we gave is going to be treated as secure.

```
def makemysecret(x:Nat[10]):Nat[10] = pred(key)

makemysecret(4[4]) --> makemysecret(4[10]) --> pred(4[10]) --> 3[10]
```

This demonstrates a simple form of subtyping: a low security type is a subtype of a higher security one.

The motivating examples in the practical mentioned the side-effecting function `putnum` as an obvious low-security *sink*:

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

Notice that the important low security here is on the input, not on the output level, so it is reasonable to apply the subsumption rule and give applications of `putnum` also a higher type:

```
putnum(3[0]):Unit[10]
```

This is fine because we interpret the label as referring to the security level of the *result* after calling `putnum`. In fact since the unit type contains no information (only one value), security levels for Unit are not obviously meaningful.

These examples should help explain explicit information flow, more explicitly. Implicit flow is (implicitly) more tricky. Both of the following should be disallowed (assuming `key` has a high security level):

[3] that would be a risky assumption in the presence of references and a global state, of course!

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

The type system and evaluation rules we consider here will prevent the second case but not the first. In fact, the first case gets an elevated `Unit[10]` type like that shown above, which we can understand as telling us that a high-security value has contributed to a side-effect; however, dealing with implicit flow in the evaluation rules needs further extensions. So we will follow the advice originally given in the practical exercise and solve the problem just for explicit flow.

Designing the type system

After these further explorations we consider some design decisions for the formal description.

Types. We are asked to add security levels by labelling types with natural numbers:

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

If you have only primitive types like `Nat`, `Bool` and `Unit` in your language, then a type simply becomes one of these annotated with a level. (With structured types you need to consider whether to nest levels — and what does that mean? — or do the simpler thing of keeping one level for the whole type).

The examples shown only include a few levels and any program can only mention a finite number of levels. So it's reasonable to assume levels are a fixed set of natural numbers, say $l \in \{0 \ldots 10\}$. (It might help in some cases to assume a maximal security level; here we don't rely on it).

Terms and values. First we extend the syntax for values:

$$\begin{array}{lcl} v & ::= & \texttt{true} \mid \texttt{false} \mid nv \\ nv & ::= & 0 \mid \texttt{succ}\ nv \\ sv & ::= & v[l] \end{array}$$

The new case of stamped values sv is what was used above. A terminating computation should end with a stamped value.

It is tempting to also introduce a term former to assign security levels to terms, but the examples above show that this isn't necessary, we could simply define dedicated coercion functions, such as:

```
def level10(x:Nat[0]):Nat[10] = x
```

Then writing `level10(5123)` should be good enough to stamp the value in the operational semantics. However, since the safety theorems require that values are a subset of terms, we must extend terms with stamped values as well:

$$t \;::=\; \cdots \mid sv$$

It's not necessary for the programmer to be able to write stamped values directly (compare with the case for references where we add locations to the syntax for terms).

Here's what we need to do next:

(i) Design **evaluation rules** which capture a run-time model of security monitoring. Terms which have a run-time security violation should get stuck.

(ii) Design **typing rules** which guarantee that typable terms do not generate security violations, i.e. they do not get stuck in our evaluation rules.

(iii) Prove the **theorems** which establish type safety formally.

(i) The evaluation relation (abstract machine)

The idea of proving type safety for our language is to show that terms that are ill-behaved operationally cannot be given types. In this setting, "ill-behaved" means having a type error as usual, or making a security violation.

To check for security violations, the abstract machine needs to know about security levels. Notice that this does *not* mean that in a real machine the security levels would necessarily need to be monitored, because the type safety proof will tell us that if we type-check terms before executing them, we do not need to monitor security levels. This is exactly analogous to the situation with integers and booleans: the abstract machine we formalise *does* see the difference between integers and booleans, by the structure of the formal syntax. This is how terms like `succ(false)` can get stuck. On a real machine, the syntactic distinction is not there and (without run-time type tags) we just have a collection of bits, so it is possible to confuse data of one type with data of another.

What does the machine need to know about security levels? To be able to check levels on function calls, it needs to know the levels assigned to argument positions. To be able to stamp the result of evaluating a function (possibly promoting the level), it needs to know the levels assigned to result positions.

Recall the usual evaluation rules for functions:

$$\frac{}{f(\vec{v}) \longrightarrow [\vec{x} \mapsto \vec{v}]\, t_f}$$

$$\frac{t_i \longrightarrow t_i'}{f(\vec{v}, t_i, \vec{t}) \longrightarrow f(\vec{v}, t_i', \vec{t})}$$

where the program P contains `def` $f(\vec{x} : \vec{T}) : T = t_f$. The idea is that once all of the arguments become values, we can substitute into the body; the second rule evaluates the first non-value argument towards a value.[4] For values tagged with levels, we can change the first rule to:

$$\frac{}{f(\vec{v}[\vec{l}]) \longrightarrow [\vec{x} \mapsto \vec{v}[\vec{l}]]\, t_f}$$

where the SFOFL program P contains a definition with levels, `def` $f(\vec{x} : \vec{T}\,[\vec{l}]) : T[l] = t_f$ (this time the vector notation abbreviates a list of arguments of the form $x_i : T_i[l_i]$). The reference to $\vec{l}$ on the left hand side implies that the machine must check that the stamped values have *exactly* the right stamps.

But this rule does not deal with stamping the result. If we evaluate `level10(5123)` then the result is just `5123` again. To handle stamping the result, we will augment the evaluation relation with a *target security level* l, writing $t \longrightarrow_l t'$. The idea is that evaluation progresses at the target level until a value is reached, and then that value gets stamped with the target level. The function rules now become:

$$\frac{}{f(\vec{v}) \longrightarrow_l [\vec{x} \mapsto \vec{v}]\, t_f}$$

$$\frac{t_i \longrightarrow_{l_i} t_i'}{f(\vec{v}, t_i, \vec{t}) \longrightarrow_l f(\vec{v}, t_i', \vec{t})}$$

This works fine for outermost calls like `level10(5123)` (try it), but for nested applications we may be setting the target level higher than the result level of a function. (Consider the `comparehash`

[4] Beware the vector notation shorthand: the vector $\vec{v}$ has different lengths in the two rules, of course.

example earlier). This is safe because we can raise levels; setting it lower would of course be a security violation (`level10(5123)` should evaluate to `5123[10]` or higher, never `5123[4]`). Therefore we let the evaluation of functions take place at any level which dominates the level of the function result.

Similarly, if we evaluate a basic operation like `iszero`, we can make progress at any level which dominates the level of the argument value. (If the argument level exceeds the target level, it is not worth going further). Notice that stamped values now serve as the final result, so the rules that mention values must be adjusted to include stamps. An example for E-IszeroSucc is given in Table 3, along with the specimen function rules, adjusted as described.

$\boxed{t \longrightarrow_l t'}$

$$\frac{k \leq l}{\texttt{iszero}\ 0[k] \longrightarrow_l \texttt{true}[k]} \qquad \text{(E-IszeroSucc)}$$

$$\frac{l \leq l'}{f(\vec{v}[\vec{l}]) \longrightarrow_{l'} [\vec{x} \mapsto \vec{v}[\vec{l}]]\, t_f} \qquad \text{(E-FunBody)}$$

$$\frac{t_i \longrightarrow_{l_i} t_i' \qquad l \leq l'}{f(\vec{v}[\vec{l}], t_i, \vec{t}) \longrightarrow_{l'} f(\vec{v}[\vec{l}], t_i', \vec{t})} \qquad \text{(E-FunArg)}$$

Where $\texttt{def}\ f(\vec{x} : \vec{T}[\vec{l}]) : T[l] = t_f$ is in P

Table 3: Some SFOFL Evaluation Rules (in program P)

In addition to the modified FOFL rules, we need two crucial new rules, E-Lab to stamp unlabelled values v with l to become $v[l]$, and E-Sub to *promote* stamped values $v[k]$ to $v[l]$ when it is safe (the rule requires strict inequality, *not* $k \leq l$, why?). One final subtlety is that we need an extra rule E-SuccLab for `succ` to normalise labels.

Exercise. Complete the definition of the evaluation rules following the above guidelines. You should have 15 rules in total. [*6 marks*]

(ii) Defining the typing relation

The typing rules are more straightforward than the evaluation relation. The judgements are:

$$\Gamma \vdash t : T[l] \qquad\qquad \vdash P$$

Each typing rule is modified to have labels added, and labels are added to types in the context. Obviously, the rule for function applications must check that the argument label for each argument t_i matches the declared label l_i from the program.

One new rule, T-Lab, is needed for typing labelled values, and another, T-Sub, for promoting labels. The latter is a form of subsumption, but without structured types it isn't really worth introducing a special subtyping judgement because we simply have $T[l] <: T[l']$ exactly if $l \leq l'$.

An interesting question is over the typing of the statement `if` t_1 `then` t_2 `else` t_3. With implicit flows in mind, a possible form for the rule is to require the same security level on each position: intuitively this reflects that if we compute some value depending on the guard t_1, then it should have a level at least as high.

Exercise. Complete the definition of the evaluation rules following the above guidelines. You should have 11 rules in total for the main judgement. [*6 marks*]

(iii) Proving the safety theorems

The safety theorems are fairly straightforward modifications. Notice that the Progress theorem captures our intention that evaluation proceeds at level l until we reach a value stamped with l.

Theorem 3 (Progress) *If* $\vdash\ t : T[l]$ *then either* t *is an* l*-labelled value, or there is a* t' *such that* $t \longrightarrow_l t'$.

Theorem 4 (Preservation) *If* $\Gamma \vdash\ t : T[l]$ *and* $t \longrightarrow_l t'$ *then* $\Gamma \vdash\ t' : T[l]$ *too.*

We need the usual lemmas to prove the safety theorems. In addition, to prove Progress we require one auxiliary lemma given below. (The proof of this gives you more hints about the forms of the evaluation rules!)

Lemma 5 (Raised level progress) *If* $t \longrightarrow_l t'$ *and* $l \leq l'$ *then there is a* t'' *such that* $t \longrightarrow_{l'} t''$.

Proof. By induction on the derivation of $t \longrightarrow_l t'$. Consider the last case used:

Case E-Lab**:** then $t = v$ and $t' = v[l]$. By the same rule, we have $v \longrightarrow_{l'} v[l']$.

Case E-Sub**:** then $t = v[k]$ and $t' = v[l]$ for $k < l$. Then by the same rule we have $v[k] \longrightarrow_{l'} v[l']$ since $k < l'$.

Case E-FunBody**,** E-FunArg**:** We have $t \longrightarrow_{l'} t'$ by the same rule again because we may increase l and still satisfy the premise.

Case others: the result follows for all other rules either immediately by raising the target level in the same rule, or by applying the the induction hypothesis and then using the rule again. □

Exercise. Complete the safety theorem proofs. You should write up at least some representative cases (but preferably whole proofs, for practice). Variables and functions are suggested in the practical handout because they are the key constructs of FOFL, and you should also include some cases of values, because they help understand the system, and also cases to test the new typing and evaluation rules in SFOFL. Furthermore, you should state the Canonical Forms and Substitution lemmas in the form you need them (proofs are not necessary but you should convince yourself they hold). [*5 marks*]

Additional resources. To give you more ideas about the topic and 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`.

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 24th February 2006**.

David Aspinall
9th February 2006
2006/02/08 16:25:20.