Types and Programming Languages
Practical Exercise CW1

Security Types for a First-order Functional Language HINTS FOR PART 1

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

This document contains some hints for solving Part 1 of the first assessed coursework exercise. It guides you towards a particular solution of a simplified form of the exercise; other solutions and design choices are also possible. The simplification is to only deal with explicit information flow.

If you have an alternative solution you should give details and carefully motivate its operation back to the original specification; since this is a rather challenging problem, incomplete solutions will be accepted (but as a minimum, they must solve the explicit flow case).

The original marking guide will be applied if you follow these hints, except that because the hints provide the informal explanation asked for in the practical document, 12 marks will be allocated for the definitions (6 marks each for steps 1 and 2 below).

The **submission deadline** is the same as for the second exercise, namely **5pm 17th March 2004**. Submit your work as described in Part 1.

1 Understanding security levels

Before reading this document you should review the motivation and examples given in the practical exercise. An initial question after reading the examples through 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,[1] 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
```

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

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 ruled out (assuming `key` has a high security level):

```
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.

2 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 given 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:

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

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

3. Prove the **theorems** which establish type safety formally.

2.1 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.[2] 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` 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 1, along with the specimen function rules, adjusted as described.

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*]

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

$\boxed{t \longrightarrow_l t'}$

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

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

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

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

Table 1: Some SFOFL Evaluation Rules (in program P)

2.2 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 $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$. 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*]

2.3 Proving the safety theorems

The safety theorems are 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 1 (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 2 (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 3 (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). Variables and functions were suggested in the practical handout because they are constructs of FOFL; 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*]

David Aspinall
3rd March 2005
2005/03/03 10:14:41.