

DIFT - Dependent Information Flow Types Typechecker Prototype Draft Release Notes (v1.10)

Luisa Lourenço Luis Caires
CITI and NOVA Laboratory for Computer Science and Informatics
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

September 2014

Contents

1	Introduction	1
2	Installation and Usage	1
3	Programming Language (Concrete Syntax)	2
3.1	Security Types	3
3.2	Expressions	4
3.3	A Conference Manager System: Example	5

1 Introduction

This short note documents the type checker prototype implementation associated to our POPL'15 paper “Dependent Information Flow Types”. Our current type checker implementation (version 1.10), based on the algorithm described in the paper above, is still under development (check for updates), but already covers all features of the language and type system, and can be used e.g. to check all examples in the paper and experiment with the type system.

2 Installation and Usage

To install the DIFT typechecker prototype, simply unzip the .zip file. The .zip contains the following files:

- `conf_examples` file : some of the examples of a conference manager system
- `conf_examples_abrv` : same as `conf_examples` but using global type and identifiers definitions
- `simple_examples` : illustrates some basic type system features
- `dropbox_examples` : illustrates a toy dropbox service example

- `LambdaDIFT.jar` : jar file with the typechecker prototype code
- `z3_linux` , `z3_mac`, and `z3_win` : binaries of Z3 SMT Solver, version 4.3.2
- `dift` script file and `dift.bat` (Windows) script file: template script file to run the typechecker, you may change it if needed
- `default_lattice` : file containing the description of a sample security lattice

Synopsis:

```
sh dift [-help] [-input <file>] [-output <file>] [-debug <file>] [-lattice <file>]
```

Running the DIFT Typechecker prototype in interactive mode:

1. Run `sh dift` to execute the typechecker with the default options, using security lattice defined in `default_lattice`.
2. Execute command `#exit` to exit the typechecker

Running the DIFT Typechecker prototype from a source file:

1. To check (an) example(s) from a source file, run `sh dift -input <file>` to execute the typechecker with the default lattice, using the examples from file `<file>`. E.g., `sh dift -input simples_examples` will check code in file `simple_examples`.
2. Programs always terminate with `;;`, so a file can contain multiple programs separated by `;;`.

3 Programming Language (Concrete Syntax)

The concrete syntax of our core programming language is given by Figures 1 and 2 (at the end of this document). For now, we do not include primitives to define a security lattice, the lattice specification is written in a separate file, at given as a parameter input to the type checker. By default, we give a predefined security lattice, automatically imported by the typechecker via file `default_lattice`.

Once we execute the typechecker in interactive mode (`sh dift`), a session is started with the default security lattice and we can input programs, one at a time separated by `;;`, and obtain the resulting type. We also have definitions of programs and types to be added to a global environment, stored during a session. In order to do so, we use the following syntax:

global declarations

```
let identifier = expression ;;
```

and *global type definitions*

```
typedef type_name = type ;;
```

Defined types and terms can then be used subsequently anywhere, e.g., one may define types that depend on other type definitions

```
typedef typeA = type ;;
typedef typeB = typeA;;
```

3.1 Security Types

Security types are base types annotated with a security label.

$$\text{security_type} ::= \text{base_type} \wedge \text{security_label}$$

Security Labels

Security labels can be standard or indexed

$$\begin{aligned} \text{security_label_noindex} &::= \text{identifier}, \text{"BOT"}, \text{"TOP"} \\ \text{security_label} &::= \text{security_label_noindex} \wedge (\text{label_index}_1, \dots, \text{label_index}_n) \end{aligned}$$

Label indexes are of the form

$$\text{label_index} ::= \text{"BOT"}, \text{"TOP"}, \text{numericLit}, \text{"true"}, \text{"false"}, \text{identifier}$$

For instance, we can declare security label `users` as the security compartment for all users in a system. Or, instead, we can index the security label with the identifier of user 42 and declare security label `users(42)`, stating the security compartment of user 42's data.

Type declarations

A *type declaration* can be a security type or a type declaration identifier

$$\text{type_decl} ::= \text{security_type}, \text{identifier}, \{ \text{type_decl} \}$$

A particular case of a type declaration, however, is the *collection type* which is declared as `{ type_decl }` to describe the security type of its elements and whose security label, as expected, matches that of its elements.

A *base type* (`base_type`) can be a *basic type* (`int`, `bool`, `cmd`), a *reference type*

$$\text{ref}(\text{type_decl})$$

a *dependent sum type*

$$\text{Sigma}[\text{id}_1: \text{type_decl}_1, \dots, \text{id}_n: \text{type_decl}_n]$$

a *dependent product type*

$$(\text{Pi}(\text{id}_1: \text{type_decl}_1, \dots, \text{id}_n: \text{type_decl}_n). \text{type_decl})$$

or a *function type*

$$(\text{type_decl}_1, \dots, \text{type_decl}_n \Rightarrow \text{type_decl})$$

So, for instance, to declare the type a table containing all users in a system, we write the following type

$$\{ \text{ref} (\text{Sigma}[\text{uid}: \text{int}^{\text{BOT}}, \text{name}: \text{int}^{\text{U}(\text{uid})}, \text{univ}: \text{int}^{\text{U}(\text{uid})}, \text{email}: \text{int}^{\text{U}(\text{uid})}]^{\text{BOT}})^{\text{BOT}} \}$$

which corresponds to a collection type whose elements are references. The reference type has security label `BOT` and its elements are of dependent sum type. The dependent sum type, in turn, has four fields: `uid`, `name`, `univ`, and `email`. Each field is typed with base type `int` and, with the exception of field `uid`, have security label `U(uid)`. So the security label is indexed and depends on a previous field, `uid`.

3.2 Expressions

Expressions have the following forms:

local declarations

```
let identifier = expression in expr
```

local type definitions

```
typedef type_name = type in expr
```

sequence

```
expr ; expr
```

conditional

```
if expr then expr [else expr]
```

where the else branch may be omitted if the conditional is a command (does not return a value).

conditions

```
expr and expr, expr or expr, not expr, expr == expr
```

corresponding to boolean operations.

operations

```
expr + expr, expr - expr, expr * expr, expr / expr
```

corresponding to numeric operations.

list operations

```
foreach(identifier in expr) with identifier = expr do expr  
first(expr)  
expr cons expr
```

where the `foreach` computes the accumulated value of a list's elements, `first` retrieves the first element of a list, and `cons` adds an element to a list.

application

```
expr(expr1, ..., exprn)
```

is the application of a function.

field access

```
expr.identifier
```

projects the field of a record.

assign

```
expr := expr
```

is the assignment operation

dereference

`! expr`

retrieves the contents of a reference.

a *upcast* operator

`[type_decl] expr`

that up-classifies (raising their security level) expressions.

and a *downcast* operator

`]security_label[expr`

that down-classifies (lowering their security level) the security level of expressions's context (*pc*).

As values we have integer, strings, booleans, empty lists {} and

references

`ref expr`

lambdas

`fun identifier1 : type_decl1, ..., identifiern : type_decln => expr`

records

`[identifier1 : type_decl1, ..., identifiern : type_decln]`

lists

`{identifier1, ..., identifiern}`

We illustrate some primitives of our language in the following section using a conference manager system as example.

3.3 A Conference Manager System: Example

In this scenario, a user of the system can be either a registered user, an author user, or a programme committee (PC) member user. The system stores data concerning its users' information, their submissions, and the reviews of submissions in "database tables" which we will represent as lists of (references to) records (e.g., mutable lists). So we start by defining the types for each "database table":

```
typedef usr_type = { ref (Sigma[uid: int^BOT, name: int^U(uid),
                           univ: int^U(uid), email: int^U(uid) ]^BOT)^BOT } ;;
```

```
typedef sub_type = { ref (Sigma[uid: int^BOT, sid: int^BOT,
                              title: int^A(uid, sid), abst: int^A(uid, sid),
                              paper: int^A(uid, sid) ]^BOT )^BOT } ;;
```

```
typedef rev_type = { ref (Sigma[uid: int^BOT, sid: int^BOT,
                              PC_only: int^PC(uid, sid), review: int^A(TOP, sid),
                              grade: int^A(TOP, sid)]^BOT)^BOT } ;;
```

And then we declare the “database tables” as empty collections of the previously defined types.

```
let Users = {}: usr_type ;;
let Submissions = {}: sub_type ;;
let Reviews = {}: rev_type ;;
```

Notice that the declared types are defining the following security policy: a registered user’s information is *only* observable *by himself*; the content of a paper can be seen by *its author as well as its reviewers*; and regarding a submission’s review, we have that comments to the PC can *only* be observable to the other members that are *also reviewers of the submission*, and that comments and grade of the submission can be seen by *its author only*.

So security level $U(uid)$ represents registered users with id uid ; $A(uid, sid)$, stands for author of submission with id sid and whose user id is uid ; and $PC(uid, sid)$, stands for PC members assigned to review submission with id sid and whose user id is uid . Also $A(TOP, sid)$ represents the security compartment of (all) authors of submission with id sid .

The (default) security lattice is defined by the following axioms (quantifiers ranging over natural numbers):

$$\forall uid, sid. U(uid) \leq A(uid, sid)$$

$$\forall uid1, uid2, sid. A(uid1, sid) \leq PC(uid2, sid)$$

As well as by the general axioms for any security lattice in our system

$$\ell(\bar{v}, u, \bar{w}) \leq \ell(v, \top, w) \text{ and } \ell(\bar{v}, \perp, \bar{w}) \leq \ell(\bar{v}, u, \bar{w}).$$

So, e.g, for all uid we have $U(\perp) \leq U(uid) \leq U(\top)$; we can see $U(\top)$ as the approximation (by above) of any $U(uid)$, e.g, standing for the standard label U .

Going back to our example, suppose we want to retrieve all the assigned papers of a given reviewer

```
typedef sub_elem = Sigma[uid: int^BOT, sid: int^BOT, title: int^A(uid, sid),
                        abst: int^A(uid, sid), paper: int^A(uid, sid) ]^BOT ;;

typedef sub = { sub_elem } ;;

let viewAssignedPapers = fun uidr: int^BOT =>
  ( foreach(x in Reviews) with res_x = {}: sub do
    let tuple_rev = !x
    in if(tuple_rev.uid == uidr ) then
      ( foreach(y in Submissions) with res_y = {}: sub do
        let tuple_sub = !y
        in if(tuple_sub.sid == tuple_rev.sid) then
          tuple_sub::res_y else res_y )
        else res_x ) ;;
```

Now assume we want to offer an operation, say `addCommentSubmission`, that allows PC members to add comments to other PC members during the evaluation of a given submission. Such operation can be written as

```
let comment = fun u: int^BOT, s: int^BOT, r: sub_elem =>
  [ int^A(u, s) ] (if(r.uid == u and r.sid == s) then r.paper else 1)

in let addCommentSubmission = fun uid_r: int^BOT, sidr: int^BOT =>
```

```

    ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
      if(p.sid == sidr ) then
        ( foreach(y in Reviews) with dummy2 = skip do
          let trev = !y in
            if(trev.sid == p.sid ) then
              ( let up_rec = [uid: int^BOT = trev.uid,
                             sid: int^BOT = trev.sid,
                             PC_only: int^PC(uid,sid) = comment(p.uid, p.sid, p),
                             review: int^A(TOP, sid) = trev.review,
                             grade: int^A(TOP, sid) = trev.grade ]
                in y := up_rec ) ) )
    in addCommentSubmission;;

```

Notice that `viewAssignedPapers` was previously declared in another program using a global declaration, which allows us to use it in the code of `addCommentSubmission` operation, and has type

```

( Pi(uidr: int^BOT).{ Sigma[uid: int^BOT, sid: int^BOT,
                        title: int^A(uid, sid), abst: int^A(uid, sid),
                        paper: int^A(uid, sid)]^BOT } )^BOT

```

Also, we use an upcast operator on the definition of `comment` function to raise the function result's security level to $A(u, s)$, in order to obtain the dependent product type

```

( Pi(u: int^BOT, s: int^BOT,
   r: Sigma[uid: int^BOT, sid: int^BOT, title: int^A(uid, sid),
            abst: int^A(uid, sid), paper: int^A(uid, sid)]^BOT).int^A(u, s))^BOT

```

when typechecking `comment`. Notice that its return type in the call `comment(p.uid, p.sid, p)` has security label $A(p.uid, p.sid)$. Additionally, we know `t_rev` has the type of the collection's references, `rev_type`'s elements type:

```

Sigma[uid: int^BOT, sid: int^BOT, PC_only: int^PC(uid, sid),
      review: int^A(TOP, sid), grade: int^A(TOP, sid)]^BOT

```

So, in order to type check the assignment expression, `y := up_rec`, we need to check that `up_rec` has the same type as `t_rev`'s. Namely, we have to check if `comment(p.uid, p.sid, p)` has type $PC(t_rev.uid, p.sid)$.

As we have seen, the type for `comment(p.uid, p.sid, p)` has security label $A(p.uid, p.sid)$ but since it has field dependencies, we need to infer values for them. In this case, we cannot infer a value for field `uid` so we approximate to `TOP` obtaining $A(TOP, p.sid)$. However, because we know by the conditional that `t_rev.sid = p.sid`, we can index the security level by field `sid` instead, which allows us to type the assignment operation since field `sid` is bounded by the dependent sum type of the record being used for the assignment.

Then we can type `comment(p.uid, p.sid, p)` with type $A(TOP, p.sid)$ and thus, due to $A(TOP, p.sid) \leq PC(BOT, p.sid)$, we can up-classify `comment(p.uid, p.sid, p)` with $PC(t_rev.uid, p.sid)$. Meaning we can type the record `up_rec` with the dependent sum type

```

Sigma[uid: int^BOT, sid: int^BOT, PC_only: int^PC(uid, sid),
      review: int^A(TOP, sid), grade: int^A(TOP, sid)]^BOT

```

So when typechecking the program above, the typechecker outputs

```

Type: ( Pi(uid_r: int^BOT, sidr: int^BOT).(cmd^BOT) )^BOT

```

However, if we remove the line 'if(trev.sid == p.sid) then' from the definition of addCommentSubmission:

```
let comment = fun u: int^BOT, s: int^BOT, r: sub_elem =>
  [ int^A(u,s) ] (if(r.uid == u and r.sid == s) then r.paper else 1)

in let addCommentSubmission = fun uid_r: int^BOT, sidr: int^BOT =>
  ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
    if(p.sid == sidr ) then
      ( foreach(y in Reviews) with dummy2 = skip do
        let trev = !y in
          ( let up_rec = [uid: int^BOT = trev.uid,
                        sid: int^BOT = trev.sid,
                        PC_only: int^PC(uid,sid) = comment(p.uid, p.sid, p),
                        review: int^A(TOP, sid) = trev.review,
                        grade: int^A(TOP, sid) = trev.grade ]
            in y := up_rec ) ) )
  in addCommentSubmission;;
```

Then our typechecker outputs

```
Wrong type: Expected declared type Sigma[uid: int^BOT, sid: int^BOT,
PC_only: int^PC(uid, sid), review: int^A(TOP, sid),
grade: int^A(TOP, sid)]^BOT
but found type Sigma[uid: int^BOT, sid: int^BOT,
PC_only: int^A(TOP, sidr), review: int^A(TOP, TOP),
grade: int^A(TOP, TOP)]^BOT
```

detecting an insecure flow in the assignment since field PC_only requires security level PC(uid,sid) but the result of comment(p.uid, p.sid, p) cannot be raised to PC(uid,sid). This expresses that because we are not filtering properly the results of the iteration of collection Reviews (we removed the conditional), we cannot guarantee confidentiality of the PC_only field. Indeed, we could be adding a comment of a submission to the register of a PC member who is not a reviewer of that submission.

We refer to our technical report <http://ctp.di.fct.unl.pt/~luisal/resources/TR-DIFT.pdf> for more technical details on dependent information flow types.


```

<program>      ::= (<expr> | <gtypedef> | <glet> | "clear") ";;"

<expr>         ::= <seq> | <ifthenelse> | <let> | <operations>
                  | <foreach> | <typedef>

<gtypedef>     ::= "typedef" ident "=" type_decl
<glet>         ::= "let" ident "=" <expr>

<seq>          ::= <expr> ";" <expr>
<ifthenelse>   ::= "if" <operations> "then" <expr> ["else" <expr>]
<let>          ::= "let" ident "=" <operations> "in" <expr>
<foreach>      ::= "foreach" "(" ident "in" <application> ")"
                  "with" ident "=" <operations> "do" <expr>
<typedef>      ::= "typedef" ident "=" <type_decl> "in" <expr>
<operations>   ::= <assign> | <cons> | <conditions>

<assign>       ::= <values> ":" <operations>
<cons>         ::= <operations> ":" <operations>
<conditions>   ::= <and> | <or> | <compare>

<and>          ::= <compare> "and" <conditions>
<or>           ::= <compare> "or" <conditions>
<compare>      ::= <sum> "==" <compare> | <sum>

<sum>          ::= <add> | <sub> | <product>
<add>          ::= <product> "+" <sum>
<sub>          ::= <product> "-" <sum>
<product>      ::= <mul> | <div> | <application>

<mul>          ::= <application> "*" <product>
<div>          ::= <application> "/" <product>
<application> ::= <fieldAcc> "(" (expr ",")* ")" | <fieldAcc> | <unary>

<unary>        ::= <not> | <deref> | <first>

<not>          ::= "not" <application>
<deref>        ::= "!" <application>
<first>        ::= "first" "(" <application> ")"

<fieldAcc>     ::= <values> "." <ident> | <values>
<values>       ::= "true" | "false" | "skip" | ident | <pargs> | <lambda> | <downcast>
                  | <record> | <collection> | numericLit | <ref> | <empty> | <upcast>

<pargs>        ::= "(" <expr> ")"
<lambda>       ::= "fun" ( ident ":" <type_decl> ",")+ "=" <expr>
<record>       ::= "[" (ident ":" <type_decl> "=" <expr> ",")+ "]"
<collection>   ::= "{" (<expr> ",")+ "}"
<ref>          ::= "ref" <expr>
<empty>        ::= "{" "}" ":" <type_decl>
<upcast>       ::= "[" <type_decl> "]" <values>
<downcast>     ::= "]" (<security_label> | <security_label_noindex>) "[" <values>

```

Figure 1: Concrete Syntax (expressions)

```

<type_decl> ::= ident | <security_type> | "{" <type_decl> "}"

<security_type> ::= <base_type> "^" (<security_label> | <security_label_noindex>)

<base_type> ::= "int" | "Int" | "bool" | "Bool" | "cmd" | "Cmd"
              | <refType> | <sumType> | <prodType> | <functionType>

<refType> ::= "ref" "(" <type_decl> ")"
<sumType> ::= "Sigma" "[" (ident ":" <type_decl> ",")+ "]"
<prodType> ::= "(" "Pi" "(" (ident ":" <type_decl> ",") ")" "." <type_decl> ")"
<functionType> ::= "(" (<type_decl> ",")+ "=>" <type_decl> ")"

<security_label_noindex> ::= ident | "BOT" | "bot" | "TOP" | "top"
<security_label> ::= (ident | "BOT" | "bot" | "TOP" | "top")
                    "(" (label_indexes ",")+ ")"

<label_indexes> ::= "BOT" | "bot" | "TOP" | "top" | numericLit | "true" | "false" | ident

```

Figure 2: Concrete Syntax (types declarations)