Implementation of IF in Cooma

Table of Contents

Model

Information flow is moddelled using the idea of information at two levels: public and secret. Secret variables (containing secret information) are denoted with a trailing !. For example, `var x : Int! = 1` declares a varible `x` of type `Int` which is classified as secret.

This notion extends to function arguments, return types, record fields and variant options. For example the following function `iden` takes a secret string as it's only argument and returns it:

```
def iden(x : String!) String! = x
```

We can't leak the value of the secret string using the following alternate version of `iden` as it is ill-typed:

```
def iden(x : String!) String = x
                               ^
error : Expected a public String but got a secret String
```

When extending this idea to records and variants, we are faced with a choice between allowing the whole type itself to also be classified as secret or keeping them as strictly public types (see Records and Variants for further dicussion on design choices). In the current implementation, we have chosen to allow records and varaints types to also be classified secret. With this decision comes additional caveats to the way records and variants may be declared:

- Public records and variants can contain a mixture of public and secret fields.
- Secret records and varaints can **ONLY** contain secret fields.
    - Functions within secret records must have both **ONLY** secret arguments and a secret return type. Enforcing argument types to be secret doesn't add any further restriction on what can be passed to them since public data can be classified as private. The secret return type does however enforce that whatever the function produces be secret. These rules extend to functions that may be returned from functions.

```
// Records
val mixedRec : { x : Int, y : Int! }
val secretRec : { x : Int!, y : Int! }!

// Variants
val mixedVar : < x : Int!, y : Int >
val secretVar : < x : Int!, y : Int! >!

// Secrets can't contain public fields
val errorRec : { x : Int }!
                     ^
var errorVar : < x : Int >!
                     ^
error : Secret record/variant can't contain public fields
```

Since the underlying representation of capabilites (`Reader`/`Writer`/`ReaderWrtier`) are as records, the same rules apply then those of records. However, due to the fact that these capabilities are pre-defined, we have slightly more control over there underlying representation. We chose to implement them as secret records (see [Capabilities](#) for further discussion on design choices). An additional restriction is that either the data they consume (`Writer`) or produce (`Reader`) must also be secret to abide by the rules of secret records. This means the `ReaderWriter` does not have differing information levels for its `read` and `write` methods. This is inetended to be the case as it makes more sense that a single capability exists at either a public or secret level, not a mix. If a mix is required, this can be done by declaring seperate `Reader` and `Writer` capabilities. The underlying type of secret capabilities are:

```
type Reader! = { read : () => String! }!
type Writer! = { write : String! => () }!
type ReaderWriter! = { read : () => String!, write : String! => () }!
```

Subtype Relation

We use the subtyping relation to enforce the notion of a "no write-down" policy. This does not allow secret variables to be used in place of public variables. Since we don't use any additional construct, other then `!`, the subtype is sufficient to enforce this. The relation $T <: T!$ states that a type T is a subtype of $T!$, it's secret alternate. This allows public values to be made secret in order to perform computions with secret values, while disallowing secret values to become public.

Contravariance and Covariance

The following describes the relationship between subtypes and arguments (contravariance) and subtypes and return types (covariance). It states that given a function $S_1 \to S_2$ we can provide an argument T_1 such that T_1 is a subtype of the argument type S_1. The second part states that we can return an value of type T_2 provided that it is a supertype of the expected return type S_2 .

$$\frac{T_1 \ <: \ S_1 \qquad S_2 \ <: \ T_2}{S_1 \to S_2 \ <: \ T_1 \to T_2}$$

The below example was adapted from [Covariance and Contravariance (computer science)](#).

```
/*
Assues the existance of the following relation Cat <: Animal
*/

void contra(Animal a){ ... }
contra(new Cat()); // Ok - T1 <: S1 (Cat <: Animal)

Animal co(Cat a){
  return a;        // Ok - S2 <: T2 (Cat <: Animal)
}
```

Using our subtype relation we can sketch some examples showing that we can classify public information but cannot declassify secrets:

```
// Using subtype realtion Int <: Int! as an example

// Contravariance
{
  def contra(x : Int!) ...
  val x : Int = 10
  contra(x)        // Ok - Int <: Int!
}
{
  def contra(x : Int) ...
  val x : Int! = 10
  contra(x)        // Error - Int! <\: Int
}

// Covariance
{
  def co(x : Int) Int! = x
  val x : Int = 10
  co(x)            // Ok - Int <: Int!
}
{
  def co(x : Int!) Int = x
  val x : Int! = 10
  co(x)            // Error - Int <\: Int
}
```

Implementation

This section will provide various implementation details of our information flow model in Cooma.

Syntax

Currently in the Cooma parser, all types are hard coded as expressions. This affords us two ways of implementing secret variants of these types:

- Add an analgous SecT for every type.
- Group types together and add another higher level Sec type:

```
Expression {paren} =
  ...
  | PredefType '!'                                            {SecT}
  | PredefType
  ...

PredefType : Expression =
    'Unit'                                                    {UniT}
  | '{' nest(FieldType ++ ',') \n '}'                         {RecT}
  | '<' nest(FieldType ++ ',') \n '>'                         {VarT}
  | 'Boolean'                                                 {BoolT}
  | 'Int'                                                     {IntT}
  | 'Reader'                                                  {ReaderT}
  | 'ReaderWriter'                                            {ReaderWriterT}
  | 'Writer'                                                  {WriterT}
  | 'String'                                                  {StrT}
  | 'Type'                                                    {TypT}
```

We have implemented the syntax for secret types using the second alternative in the `Cooma.syntax` file. This means that a the secret type is just a special wrapper around existing types. The form of secret types is this `SecT(Type)`. This is in contrast to an analagous type such as `SecIntT`. We have excluded function types `FunT` from the list as we do not feel that it's necessary for whole functions to be classified as secret. We feel that it's sufficent enough to only deal with the data being passed into the funciton.

Semantic Analysis

This section is used to collect various implementataion notes on `SemanticAnalyser.scala`.

Subtype

We can add our subtype realtion, which enforces our information flow, to the `subtype` function. Since secret types ares wrappers over normal types, we can easily identify when we are checking some type `t` against a secret type `u` and pull the underlying type `s` out for comparison.

```
def subtype(t : Expression, u : Expression) : Boolean =
  ...
  case (_, SecT(s)) =>
    subtype(t, s)
  ...
```

Record and Variant Checks

Due to our design choice to allow records or variants to be classified as secret, we need to add additional semnatic checks for SecT(RecT) and SecT(VarT). The check needs to confirm that every field in the corresponding record or variant is of the form SecT(_). If a field is public (not wrapped with a `SecT`) it will be an error.

```
def checkSecretFields(fs : Vector[FieldType], t : Expression) : Messages = {
    fs.flatMap(f => f.expression match {
        case SecT(_) => noMessages
        case _       => error(f, s"public field ${f.identifier} found in secret ${show(t)}")
    })
}
```

An interesting case arises when a record contains a field which is also a record that is calssified as public with **only** secret fields `val x : { a : { b : Int! } }!`. In this case, whilst all fields in `r` must be secret due to the trailing `!`, `a` is a public field. However, since all of the fields in `a` are secret, `a` is a direct subtype of `{ b : Int! }!`. Currenlty, this is classfieid as an ill-typed secret record, but may be allowed with more research on the potential side-effects or this use. We could either wrap these records in a secret type (`SecT`) or perform an additional check in the `checkSecRecord` function:

```
def checkSecRecord(e : Expression) : Messages =
  ...
  case RecT(fs) =>
    fs.forall {
      case SecT(r : RecT) =>
        checkSecRecord(r)
      case SecT(_) => true
      case _ => false
    }
  ...
```

Actually it turns out that nested fields are captured by the original check since it's performed on all `SecT(RecT)` and `SecT(VarT)` AST nodes. Since all fields must be public, when checking inner-records, it's not possible to have a public field of the form `SecT` and thus is safe to not add additional checks.

Primitive Operations

We also need to consider the additional changes that we need to make to primitive operations. Our model dictates that we should be able to perform primitive operations between secret and public data as long as the result is also secret i.e. `Int! + Int = Int!`. This would mean special expections need to be made for primitive operations when dealing with an operand of type `SecT`.

Aliasing

Due to the way capabilities (`Reader`/`Writer`/`ReaderWriter`) are aliased, we need to add corresponding secret capabilites to the `alias` function, corresponding to there underlying implementation in `SymbolTable.scala`. The alias of the secret capabilites are the same as there public counterparts, prefixed with 'sec':

```
def alias(e : Expression) : Expression =
  ...
  case `secReaderT`       => SecT(ReaderT())
  case `secReaderWriterT` => SecT(ReaderWriterT())
```

```
  case `secWriterT`        => SecT(WriterT())
  ...
```

This means we also need to add corresponding cases to `unalias` to allows us to 'move' the other way:

```
def unalias(n : ASTNode, t : Expression) : Option[Expression] =
  ...
  case SecT(ReaderT())       => Some(secReaderT)
  case SecT(ReaderWriterT()) => Some(secReaderWriterT)
  case SecT(WriterT())       => Some(secWriterT)
  ...
```

Basic Types

This section contains notes on the implementation of `Int`, `String` and `Boolean`.

Int and String

No additional changes were needed other then the addition of the subtyping relation.

Boolean

Due to the underlying implementation of `Boolean` the following additional were necessary:

- An additional case added to `subtype` function. This is due to the fact that `boolT` representes a boolean which unfolds to the variant `< False = (), True = ()>`. Therefore when checking if a `boolT <: SecT(BoolT())` we need to catch this case and instead check `subtype(BoolT(), s)`. This is due to the fact that we cannot get the alias for `boolT` since the alias and unalias methods do not exist in the `SemanticAnalysis` object.

Records and Variants

Only slight modifications, mainly to do with field selection on secret records were necessary. They include:

- Added extra case to the `Sel` case of the `tipe` function. This is neccessary to make sure that when checking for a record in a `SecT(RecT())` that the actual `RecT()` AST node is checked instead of throwing an error since `SecT` doesn't contain the records fields.
- Added an extra case in the `checkFieldUse` function so that if we have a secret record `SecT(RecT())`, the fields of the `RecT()` are extracted and checked isntead of throwing a "selection of X field from non-record type Y" error.

Design Notes

The discussion here centers whether or not we should allow records or variants to be considered secret i.e. `val x : { a : Int! }!` and what that means in terms of addiditonal checks for a program. As of its current implementation, Cooma allows records and varaints to be defined as secret (as opposed only to fields). We feel this makes more sense then the alternative as records and variants themselves can be types. It also makes sense that a record or variant may be considered as secret as to limit the places that it can flow to. It also allows for a more idomatic implementation of the built-in capabilites. We believe this decision makes the secret/public typing system slightly more expressive then its alternate option. The additional caveat of this however, is that additional checks must be in place, namely for secret records (`SecT(RecT())`) and secret variants (`SecT(VarT())`). We restirct secret records to conating **only** seceret fields as we feel this best fits with the ideas of a secret record. The same rule also applies to variants.

Capabilities

Building on the changes required by records and variants, capabilites required a number of additions, namely:

- Seperate secret capability variants defined in `SymbolTable.scala`. These are similiar to there public counterparts but are prefixed with `sec`. The only changes from the public versions are:
  - The record is now wrapped in a `SecT` AST node.
  - The argument and return types for the pre-defined functions (`read` and `write`), excluding `Unit` have been wrapped in a `SecT` AST node also. This means that now secret capabilites only consume (`Writer.write`) and produce (`Reader.read`) secret values.
- Extra alias cases were added to the `alias` and `unalias` function as outlined in Aliasing.

Design Notes

Much of the previous discussion on public vs secret records and variant applies to capabilites also, since there undlying representation is as a record. This is covered in Records and Variants - Design Notes. Another topic of interest is whether the `Unit` types in the `read` and `write` functions should be typed as `SecT(UniT)` or just `UniT`. Currenlty we are not convinced it makes much of a difference since we shouldn't need to worry since we don;t need to check subtypes between capabilities i.e. only a`Reader` can be used in place of a `Reader`, not another capability. Since these capabilities can only be assigned by the run-time (users cannot create them), we don;t need to worry about user defined ones either. Currently said function only use the `UniT` but a shift to `SecT(UniT())` would be easy if need be.

Other Additions

This section contains a number of additional changes.

An additional change was required in the `tipe` function to determine the type of `SecT` nodes:

```
lazy val tipe : Expression => Option[Expression]
  ...
  case _ : SecT =>
    Some(TypT())
  ...
```