Implementation of the History Aware Programming Language through Translation into Scala

Md. Shamim Ashik
sashik@cs.utsa.edu

Technical Report CS-TR-2012-012

Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249

Abstract

Many organizations handle information about individuals that those individuals consider private. In many cases, these organizations are required to comply with many regulations that protect this information. This sensitive information is often processed electronically, and there is frequently no rigorous basis for claiming that these software systems comply with regulations. As part of a larger framework, the History Aware Programming Language (HAPL) has been designed to facilitate the development of programs that can be formally verified against formalized privacy policies. This report provides a preliminary description of the HAPL language and its implementation. Using the ANTLR parser generator, we created a parser that can recognize any syntactically correct HAPL program. Moreover, we implemented a source-to-source compiler that translates the correct HAPL programs into Scala programs. Since, HAPL has specialized primitive constructs, the generated Scala code uses an additional Scala library against which provides the necessary facilities; we describe a partial implementation of this library.

Key words. Programming Language, Source-to-source translation.

1 Introduction

Many organizations need to follow the business norms or the policy regulations imposed either internally or by a third party such as the government. Laws and regulations, such as the Health Insurance Portability and Accountability Act (HIPAA) and the Gramm-Leach-Bliley Act (GLBA) [5], contain many rules that constrain the handling private information by organizations. However, these days, organizations rely heavily on electronic information systems to manage and/or transfer private information. Currently, there is no satisfactory way for an organization to ensure that such systems satisfy regulatory requirements and the organization's own policies. In order to enforce the policies, organizations must depend on the programmers who write the system. The current state-of-the-art in software development, however, means that defect-free software cannot be expected.

To overcome this situation, we proposed a framework [19] for the design and implementation of information systems that provably enforce privacy policies [1].

[1] This work was supported, in part, by the National Science Foundation under grant CNS-0964710

At the lowest level of the framework, we have the History Aware Programming Language (HAPL). HAPL has several unique characteristics that make it particularly suitable for developing programs that are formally verified against privacy policies. HAPL is designed around the actor paradigm with primitives for sending and receiving messages. HAPL is designed with primitive constructs for dealing with personal information about individuals, has a construct for taking actions explicitly conditioned upon past history, and has static semantics (in the form of a type system and static analysis) that ensure that only programs that comply with the declared policy are considered valid. Two of these are particularly important to be able to understand the implementation of HAPL described in this report. The verification of a HAPL program against HAPL's static semantics are beyond the scope of the work described in this report.

HAPL respects the Actor Model of computation. In the Actor Model, a program, at runtime, consists of a set of concurrently executing actors that communicate via asynchronous message passing. In HAPL, each Actor is an instance of an Actor class that may contain actor methods. After receiving a message, a HAPL actor will execute the code given in the action method for that message type. Only one action method of an actor can execute at a time, and it can be coded to send messages to itself or other actors.

HAPL also provides a novel **ltlquery** construct for conditioning the execution of a block of code on the actor's past history as described by a temporal logic formula. The implementation of this construct requires the generation of runtime code that maintains compact data structures that are able to determine the truth of such temporal logic formula at runtime [12]. As an example of where such a construct might be useful is, consider the situation where, in order to electronically send a patient's health information to any health insurance provider system, the hospital's information system needs to check whether or not the patient previously gave consent for this. Without the construct, the actor would need to represent this fact about history using explicit state variable, but by providing ltlquery as a language primitive, programs become both more concise and easier to verify. In this report, we describe the source-to-source compiler which generates code that passes the appropriate information to this runtime support. The implementation of the actual runtime support for this mechanism is left as future work.

Thus, this report describes how we implemented a source-to-source compiler that can translate valid HAPL programs into semantically equivalent Scala programs. We chose to implement HAPL with a source-to-source compiler because it will give us the opportunity to debug the program with existing tools and because Scala has a widely used Actor library that can be used to implement the actor primitives.

Road map Section 2 presents background. Section 3 presents our contributions in the areas of program structure and programming languages features. Section 4 describes the goal of this work. Also it talks about the scope of the work. Section 5 talks about the required software's installation process and important technical details of the work. Section 6 concludes.

2 Background

To date, most of the work in the area of privacy policies has focused on creating framework that can express privacy policies [7, 8, 9]. But these generally do not propose any tools or techniques to enforce privacy policies. A notable exception is the work that has been done implementing runtime monitors [15, 13, 10] to enforce security policies, these techniques are inadequate for

```
               prg = msgDecls actorDecls
          msgDecls = { ‘message’ ident typeParamDec ‘(’ typeArguments ‘)’ ‘;’}
        actorDecls = actorSignature ‘(’{ var ident ‘:’ type [,] }‘)’ ‘representing’ TLPV_ident ‘{’
                     varDecls actionDecls ‘}’
    actorSignature = ‘actor’ ident typeParamDec
          varDecls = { ‘var’ ident ‘=’ exp ‘;’}
          valDecls = { ‘val’ ident ‘=’ exp ‘;’}
       actionDecls = { ‘action’ ident typeParamDec ‘(’ formals ‘)’ ‘{’ valDecls stmtSeq ‘}’ }
              type = bool | ‘P’ ‘[’TLPV_ident‘]’ | ‘ProtectedInformation’ ‘[’TLPV_ident‘]’ |
                     ‘Actor’[‘[’ TLPV_ident ‘]’] | ‘List’ ‘[’type‘]’ | existType | recordType
         existType = ∃ ‘[’ident; inrole‘]’.type
        recordType = ‘(’ type [‘,’] ‘)’
      typeParamDec = ‘[’ { ident [,] } [;] { inrole } ‘]’
            inrole = ‘inrole’ ‘(’ TLPV_ident, ident[, TLPV_ident] ‘)’
         arguments = { exp [,] }
           formals = { ident ‘:’ type [,] }
     typeArguments = { typeName typeParamDec [,] }
           stmtSeq = { asgn‘;’ | send‘;’ | ifHistory | ifStmt | unpack | foreach }
           foreach = ‘foreach’ ident ‘in’ ‘(’ ident ‘)’ ‘{’ stmtSeq ‘}’
            unpack = ‘unpack’ exp ‘into’ ‘[’ TLPV_ident ‘]’ and ident ‘{’ stmtSeq ‘}’
         ifHistory = ‘ltlquery’ ‘(’ LTL-Query ‘)’ ‘{’ stmtSeq ‘}’ [‘else’ ‘{’ stmtSeq ‘}’ ]
              send = ‘send’ ‘(’ exp, exp, message ‘)’
           ltlsend = ‘send’ ‘(’ ident, ident, ident[‘(’ ‘{’ ident [,] ‘}’ ‘)’] ‘)’ | ‘send’ ‘(’ ident, ident, ident ‘)’
        ltlreceive = ‘receive’ ‘(’ ident, ident, ident[‘(’ ‘{’ ident [,] ‘}’ ‘)’] ‘)’ | ‘receive’ ‘(’ ident, ident, ident ‘)’
           message = ident ‘(’ arguments ‘)’
            ifStmt = ‘if’ ‘(’ exp ‘)’ ‘{’ stmtSeq ‘}’ [‘else’ ‘{’ stmtSeq ‘}’]
               exp = ident | exp.‘head’ ‘(’ ‘)’ | exp.‘tail’ ‘(’ ‘)’ | exp :: exp | pack | inrole | exp ‘==’ exp |
                     exp.‘isEmpty’ ‘(’ ‘)’ | list | recordProjection | ‘(’ arguments ‘)’
  recordProjection = ‘#’ int ‘(’ ident ‘)’
              asgn = ident ‘=’ exp
              list = ‘new’ ‘List’ ‘[’ type ‘]’ ‘(’ ‘)’
              pack = ‘pack’ ‘[’ typeParamDec ‘]’ exp : existType
         LTL-Query = ltlsend | ltlreceive | ‘contains’ ‘(’ ident, ident, ident‘)’ | T | F | ¬LTL-Query |
                     LTL-Query (∧ | ∨ | ⟹ ) LTL-Query | LTL-Query 𝒮 LTL-Query |
                     ∃ident.LTL-Query | ∀ident.LTL-Query | ⟠ LTL-Query | ⊟ LTL-Query
```

Figure 1: **HAPL Syntax in Extended BackusNaur Form (EBNF)**

enforcing privacy policy. Moreover, monitoring seeks to separate concerns for basic functionality from security concerns.

Our enforcement framework [19] is inspired by the work of the Contextual Integrity (CI) paper [8]. In the CI paper, authors proposed a framework which can express privacy norms in linear temporal logic (LTL) [14]. In their framework, two entities can communicate by sending and receiving messages where each message contains a set of agent, attribute pairs, and information about principals. This information can be protected information, transmission of which is restricted by the privacy norms and regulations. In their work, they showed that their framework is capable of expressing norms such as those found in the Health Insurance Portability and Accountability Act (HIPAA) and the Gramm-Leach-Bliley Act (GLBA). They accomplished this by separating these regulations into positive and negative norms, such that a send of message containing protected information is allowed only if it satisfies at least one positive norm and all of the negative norms.

In order to verify the correct enforcement of the privacy policies, our aim is to statically check all the send actions in the HAPL program (we will formally introduce HAPL in section 3) and determine that each send actions satisfies the required norms. This static analysis can be accomplished by using an abstract variant of the technique presented by Krukow *et al.* [12]. That work presents a dynamic programming approach to determine whether at each stage in a trace, which is being generated by a running system, a past-only temporal logic formula is satisfied. They call this *dynamic model checking*. The technique involves inductively calculating, at each state in the trace, a representation of environments under which each subformula is satisfied.

HAPL provides concurrency by supporting the Actor paradigm. This paradigm is also taken by several programming languages, including industrial languages such as Erlang, Scala.

In the actor model [11, 6], a system has several actors. These actors communicate to each other via asynchronous message passing. An actor can send a message to another actor, can receive a message from any other actors. Actors can receive multiple messages at the same time. Each actor has has a mailbox through which it receives those messages. Messages queued in the mail box are taken care of by the actor one at a time fashion. Actions taken by the actor is based on the signature/content of the messages. This is called the behavior of the actor. Upon receiving a message one actor can also change its previous behavior.

Although we do not present the semantic of HAPL in this report, it is worth noting that the development of the semantics of the actor portion is greatly inspired from the work of Agha *et al.* [6] and Talcott [17].

One important aspect of the static analysis would be to track any principal variables of the program with information about the same principal at static time. This is possible through an adaptation of Tse and Zdancewic's work on Runtime Principals [18]. In this work, runtime principals have singleton types which are constructed using principals or type-level principal variables that statically represent principals. These principals and type-level principal variables are used to construct types for data variables. Thus, one can associate a principal with variables containing data about that principal statically.

3 HAPL (History-Aware Programming Language)

In this section we will present the formal syntax of HAPL. Moreover, we will present example programs written in HAPL and describe the usage of constructs found in HAPL syntax. Before

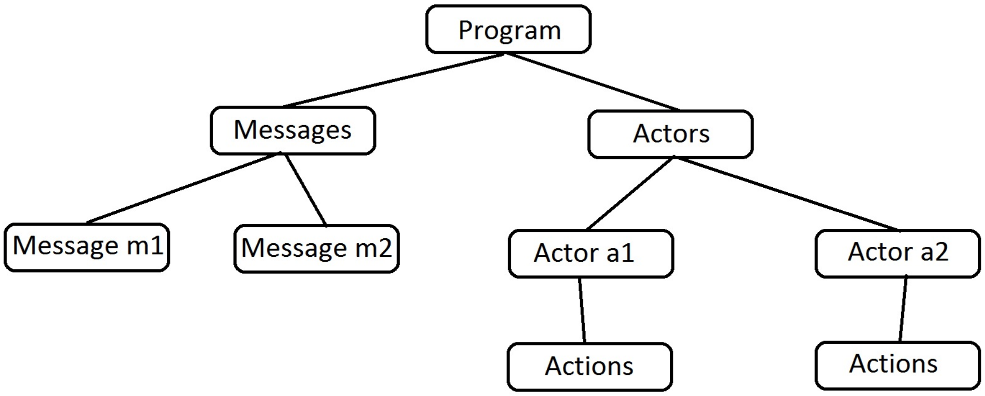

Figure 2: **Program Structure**

going to the detail, we can see figure 2 to have an understanding on the high level tree structure of any program.

HAPL Syntax. The formal syntax of HAPL is given in figure 1. From the figure we see that we can have multiple actors in a program. Actors can take arguments. Also, variables can be defined within the body of the actor before the start of any action method. Each actor can have multiple action methods. Action methods are triggered when an actor receives a message (whose signature is same as the action method) from any actor. Within the action method an actor can perform necessary computation. It can change its local states and can send messages to other actors via the send construct of the language. This is called the behavior of the actor. In HAPL, actors can only send messages inside of action methods.

If we look at the sample HAPL program of figure 3a, we can see that the program starts with 3 message definitions. Next we have $Pings$ and $Pongs$ actor. Actor $Pings$ takes a type parameter β as an input, which is actually a principal variable. It also takes an actor as an input. The actor itself represents a principal X. The actor has only one action method namely $pong$. This action method takes β as a type parameter input and within the body of this action method the actor sends a $ping$ message to an instance of the $Pongs$ actor. $object\ test$ is not a part of HAPL. This portion of the code is appended in order to run the output in Scala.

As we are converting HAPL into Scala, in order to run any code written in HAPL, we need to have a code snippet appended with the HAPL program. A sample Scala code is shown in figure 3b. In this portion of code, we are allowed to declare variables, instantiate actors and make them run. The code remains same in the ouput Scala program.

Ltlquery. As we discussed earlier, HAPL interacts with a runtime system which supports the query mechanism. In order to make this interaction, HAPL introduces a construct *ltlquery*. Right below we can see how it can be used in a program.

```
ltlquery(E act.O send(p2, p1, childInfo()) AND actsOnBehalfOf(p2, x)){
   send(p1, p2, replyDoc(x, y));
}
```

```
message ping[](Actor);
message pong[](Actor);

actor Pings[α]() representing α{
   action pong [δ](ponger:Actor){
      send(self, ponger, ping());
   }
}
actor Pongs[β]() representing β{
   action ping[γ](pinger:Actor){
      send(self, pinger, pong());
   }
}
```

Figure 3a: **A sample HAPL program**

```
object test {
   def main(args: Array[String]) {
      val ponger = new Pongs()
      val pinger = new Pings()
      pinger.start
      ponger.start
      ponger ! ping(pinger)
   }
}
```

Figure 3b: **Scala part of any HAPL program**

The *ltlquery* construct functions like a standard *if* block but it has a linear temporal logic formula instead of a boolean expression. If the expression, is staisfied by the runtime history mechanism at the time when it is executed then the operations in the ltlquery block execute. In order to evaluate the query expression, the system needs to interact with a history manager. At present, this part is handled by calling a stub function in the generated *Query* library. It is future work, to develop create a history manager generator that will maintain the appropriate state information to answer queries. Each query expression in HAPL must be a past-only, first-order linear temporal logic formula with the atomic predicates *send*, *receive*, *contains*, and *actsOnBehalfOf*. *Contains* predicates checks if a message contains particular attributes of a principal. Whereas the *actsOnBehalfOf* predicate checks whether an actor acts on behalf of a principal.

Send. In order to interact with other actors, an actor need to send asynchronous message. HAPL provides the *send* construct to make this happen. In the above code snippet, actor p1 is sending a message *replyDoc* with some information as argument to actor p2. Actors have mailboxes where they store the received messages. Each actor repeatedly pops one message from the mailbox and handles that message. If the actor has any action method with a signature same as the message popped then the action method is invoked. Otherwise, the message is discarded. In this way, the actor responds to the messages it gets. Although there is no particular order in which the actor should respond.

ProtectedInformation. This is a special construct in HAPL. Usually, privacy regulations are concerned only with any protected information of any principal. This construct gives a programmer a way to tag the data of any principal as protected so that the static analysis take take advantage of it. Below is a simple example of it's usage.

```
action getInformation[](patInfo:ProtectedInformation[alice, obese, bool]){
  ...
}
```

In this code example, the action method *getInformation* has a parameter *patInfo* of type ProtectedInformation[alice, obese, bool]. This actually implies *patInfo* has the boolean health information of a principal alice. Here obese is the attribute whose value is kept in *patInfo*. Indeed, the data is marked as ProtectedInformation.

Runtime Principal, Type Level Principal Variable (TLPV). HAPL supports Type Level Principals and Type Level Principal Variables (TLPVs) in order to keep tract of the principal in static time. The idea is described in the work on Runtime Principals by Tse and Zdancewic [18]. Runtime principals are judged to have singleton types that are constructed using principals or type level principal variables that statically represent principals. In our context, a type level principal variable provides a static representation for a Principal that is unchanged for the duration of that type level principal variable's scope. These type level principal variables do not have a runtime representation, but rather are used to parameterize the types for program-level principal values, actor values, and structures containing protected information. Additionally, actor classes and actor methods may be made polymorphic over principals by having them take type-level principal variable parameters, which can be bound to have specific roles.

As an example of how the principals and their data can be tracked statically in HAPL, consider the code below:

```
action getInformation[](patient:P[alice],
     patInfo:ProtectedInformation[alice, obese, bool])
{
  ...
}

action exchangeInformation[α](patient:P[α],
     patInfo:ProtectedInformation[α, obese, bool])
{
  ...
}
```

The action method *getInfo* takes two parameters—*patient*, which is a principal and *patInfo*, which is a ProtectedInformation. But with the help of TLPV we can track the principal and their data. First of all, *patient* can take a value which is only the principal *alice* as the type is parameterized by type level principal value *alice*. Same reason applies for *patInfo*. So, we can see that, we now have the information that *patient* is actually principal *alice* and *patInfo* is about the same principal.

Now, if we look at the *exchangeInformation* method then we can see that we have parameterized the action method with TLPV α. α is also used as a TLPV for both of the parameters *patient* and *patInfo*. Thus, if anyone calls this method then the method will take any principal and some data of the same principal unlike in the *getInformation* where the method only takes one single principal *alice* and her data.

Exist type, pack, unpack. Above example and the explanation shows that TLPVs are useful tracking individual principal and their data. But it does not explain how to track multiple (say, a million) principals and their data. In order to provide that HAPL has existential type using the *exists* keyword and the related *pack* & *unpack* constructs. In HAPL *pack* is an expression whereas *unpack* is a statement. The code given below shows the usage of these constructs.

```
action doPack[](record:(P[alice], ProtectedInformation)){
  val packed_record = pack[γ](record):exists[γ].(P[γ], ProtectedInformation);
  store = packed_record :: store;
}
action doUnpack[](store:exists[γ].(P[γ], ProtectedInformation))
{
  foreach packed_record in (store){
    unpack packed_record into [δ] and record{
      //send a message to the actor acting on behalf
      //of the principal(the first element of record)
    }
  }
}
```

The doPack action method takes a record with field types P[alice] and bool. *packed_record* is a variable that creates a packed data out of the *record*. That is, it hides the original TLPV in record with a meta variable TLPV. We added it in a list named *store*. Now, *store* has all the packed record whose original principal is hidden in static time.

The doUnpack action method takes the store as a parameter and, for every element of that list, tries to unpack the element into a the TLPV δ and a *record*. At compile time, we do not have the idea which principal is δ, but we have the knowledge that whoever δ is, *record* has their data. This is how we can relate a principal with their data.

Example: Somedays ago, there was a multistate outbreak of fungal meningitis and other infections linked to the use of injectable steroids from three recalled lots of preservative-free methylprednisolone acetate (MPA) distributed by the New England Compounding Center (NECC) [4]. Let us consider a situation where a program needs to send some kind of notification to all the patients who have been treated with the medicine described above so that the patients can have proper treatment to mitigate the effect of the drug. Pack and unpack constructs are useful in this situation as described by the code snippet written above.

Inrole. *inrole* is a very special expression in HAPL. It can check at runtime whether a principal is acting in a particular role. It also supports the check whether a principal is acting in a particular role and while acting on that role whether she is serving any particular principal. Consider the following example:

```
if(inrole(bob, alice, doctor)){
  ...
}
else{
  if(inrole(john, doctor)){
    ...
  }
}
```

The first *inrole* in this code snippet checks whether bob is alice's doctor whereas the later one is more general checking if bob is a doctor. *inrole* can be used as a normal expression or it can be used to build a query expression for the *ltlquery*.

The other constructs are standard ones. HAPL supports *if*, *list*, *foreach* to iterate over list, *record* of different types of data, *var* and *val* to declare variables etc.

4 Scope

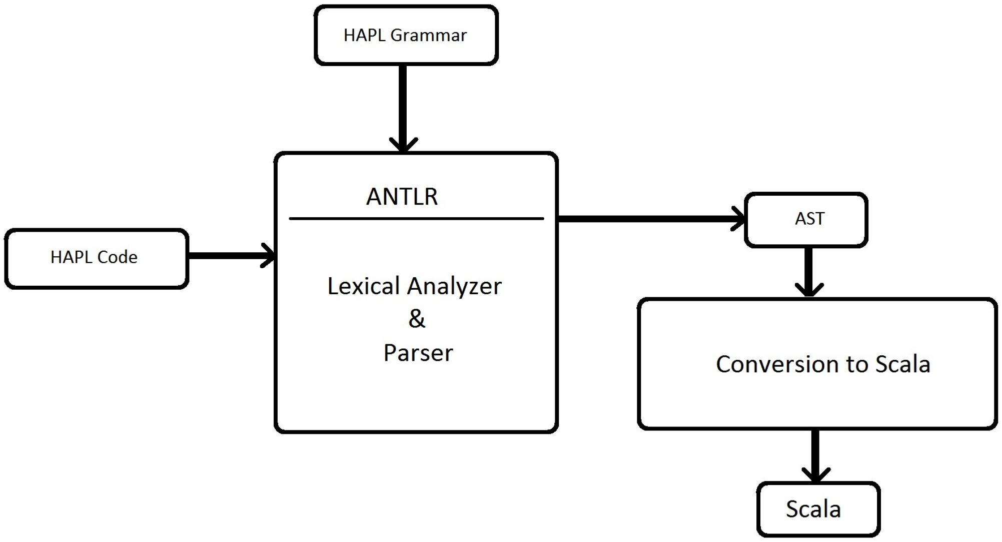

Figure 4: **Overview**

Although our framework [19] aims to have numerous implemented components, the scope of the work described in this report is fairly constrained.

The goal of this work is to implement the HAPL syntax in ANTLR [1] and output executable Scala code. We also need to make a list of all the query strings found in the input HAPL program and output them accordingly. Figure 4 gives us a high level overview of this work. In brief we have to follow the steps listed below in order to achieve the goals.

- **Generate a parser and the AST:** feed the HAPL grammar into a parser generators (i.e., ANTLR) so that the generator can generate parser to recognize language written in HAPL and produce an abstract syntax tree (AST).

- **Check the AST:** feed code snippet written in HAPL and inspect the resulting ASTs to test if the generated parser is functioning correctly.
- **Grab the AST:** fetch the AST which is needed to convert the HAPL code into Scala.
- **AST to Scala:** devise an algorithm to walk through the AST and convert it to Scala. Scala provides language support for Actor which is also a key component in HAPL.
- **Run the Scala code**.

5 Technical Details & Results

As we decided to implement HAPL syntax grammar via ANTLR [1], we had to follow the steps below in order to translate any HAPL program into Scala code.

Installation of ANTLR and others. We followed the video tutorial of [16] and instructions of [3] to install ANTLR on Eclipse. Here are the software and plugins with versions we used for this work -

- Eclipse 3.7.2 (Indigo) for 64-bit Windows OS
- Java JDK 7
- ANTLR 3.4 library
- ANTLR IDE 2.1.2
- Zest 1.3.0+
- GEF 3.7.0+
- Dynamic Language Toolkit (DLTK) Core 3.0.0
- GraphViz 2.28.0
- Scala distribution 2.9.2

HAPL syntax to ANTLR grammar. We implemented the formal HAPL syntax into ANTLR grammar. Figure 6 shows a sample HAPL grammar rule while figure 7 shows the ANTLR version of the same rule. Note that, the ANTLR version of the rule has two parts. The first part is the straightforward conversion from HAPL grammar rule to ANTLR grammar style. The second part of the rule starts after the -> sign. This is called a rewrite of a rule. In this case, we have rewritten the normal rule into a tree grammar rule which can be visualized as in figure 5. That being done, ANTLR auto generates the lexer and parser for the grammar. As the main goal of this work is to translate HAPL program in Scala, we turned the HAPL grammar into tree grammar in ANTLR. This gave us a nice abstract syntax tree (AST) of the input HAPL program. Being a tree, AST

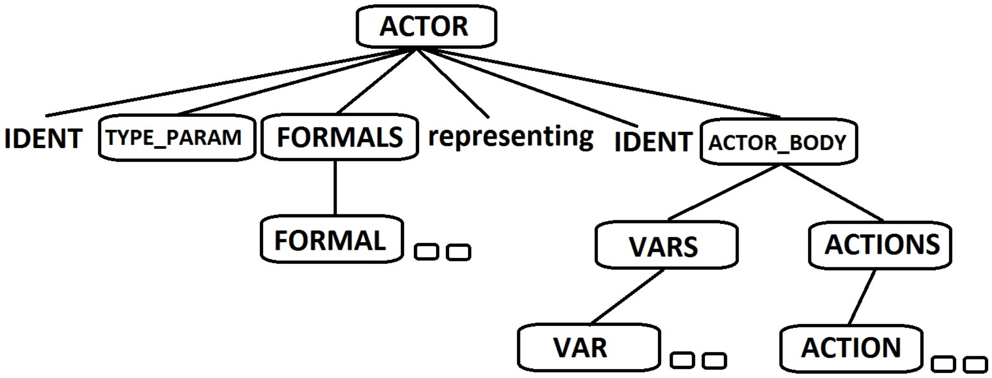

Figure 5: **Visualization of a tree grammar rule**

```
actorDecls = actorSignature '(' { var ident ':' type [,] } ')'
          'representing' ident '{' varDecls actionDecls '}'
```

Figure 6: **A HAPL grammar rule**

```
actorDecls
   : 'actor' IDENT typeParamDec '(' actorArgs* ')' 'representing'
     IDENT '{' varDecls actionDecls* '}'
   -> ^(ACTOR IDENT typeParamDec ^(FORMALS actorArgs*) 'representing'
      IDENT ^(ACTOR_BODY varDecls actionDecls*) )
```

Figure 7: **An ANTLR grammar rule of the HAPL grammar rule of figure 6**

can be converted into a dot image which is can be viewed by GraphViz [2]. It is very easy to understand the tree grammar viewing these images. They actually helped a lot finding incorrect implementation of the tree grammar so that we could fix the errors.

Note, that at present, only the syntactic rules of HAPL are enforced. HAPL's static semantics (type system and static analysis for checking code against privacy specifications) are the subject of future work.

HAPL AST to Scala. After parsing HAPL, into an AST, it is necessary to translate HAPL program into executable Scala code. This step allows us to convert the HAPL program into Scala code, although, the runtime mechanism for handling queries is the subject of future work. At this time, we provide libraries with stub functions which will work as hooks for the future runtime history mechanisms. On the other hand, this will give us compilable code in Scala. This is why we generated a small library code that is always prepended with the actual Scala output.

ANTLR provides us a nicely structured AST for any input HAPL program. This HAPL program already passed through lexer and parser which makes it syntactically correct. Having AST at hand, we need to walk through this tree and emit related compilable Scala code for all the expressions, statements and other possible constructs of HAPL. We did that by handling case by case

```
import scala.actors.Actor
class Packed[T](contents: T){ def unpack():T={return contents}}
class P(){ }
class Date(){ }
class ProtectedInformation(){ }
class PatientRecord(){ }
object Role{
  def setRole(p:P, role:String):Unit={}
  def inrole(p:P, role:String):Boolean={return true}
  def inrole(p:P, p2:P, role:String):Boolean={return true}
}
object Query{
  def ltlQuery(query:String):Boolean={return true}
}
case class ping(a0:Actor)
case class pong(a1:Actor)
class Pings() extends Actor {
  def act() {
    while(true) {
      receive {
        case pong(ponger:Actor) =>
          ponger ! ping()

} } } }
class Pongs() extends Actor {
  def act() {
    while(true) {
      receive {
        case ping(pinger:Actor) =>
          pinger ! pong()

} } } }
object test {
  def main ( args : Array[String]){
    val ponger = new Pongs()
    val pinger = new Pings()
    pinger.start
    ponger.start
    ponger ! ping(pinger)
} }
```

Figure 8: **Scala Output of a HAPL program**

of the language constructs and in a recursive manner. Also, we called the stub functions wherever necessary. Figure 8 shows the Scala output of the HAPL program of figure 3a where the top part is the generated code for the library and the part starting with 'case class' is the generated Scala code out of the HAPL AST.

Scala Main. We have added some pure Scala syntax to HAPL syntax in order to write the main function in Scala that instantiates the actors and runs them properly. This portion of the syntax is not the part of the standard HAPL syntax. If we see the HAPL program in appendix A then we will see this Scala codes in *object test* block which is exactly the same in the ouput Scala code. Also, figure 3b shows an example code of *Scala Main* program.

Run the Scala code. The output Scala code is ready to run. For example, if we run the code of figure 8 then within object test two instances of Pings and Pongs will be created and started. Pongs's intance will be initiated so that it can send a ping message to itself. Upon receiving a ping message it sends a pong message to the Pings's instance. Pings's instance receives a pong message and then returns a ping message. Thus they started the communication via message passing and in this simple example they remain reactive by sending each other indefinite number of message responses.

6 Conclusion

In this paper we have outlined how we implemented a HAPL to Scala translator. Our hope is future researchers and developers will find it useful while designing and implementing type checkers, runtime history mechanism and static analysis for HAPL.

7 Acknowledgement

This work is dedicated to the memory of William Hale Winsborough, who collaborated in the development of HAPL. HAPL is designed by Jeffery von Ronne, William H. Winsborugh, and Md. Shamim Ashik.

Jeffery von Ronne supervised the work described in this report. I also thank him for reviewing this report and suggesting necessary changes.

As discussed earlier, HAPL is a part of the "Privacy Policy Enforcement" project [19]. Jianwei Niu and Omar Chowdhury are also actively working on this project, specifically, in the part of policy analysis.

References

[1] Antlr. http://www.antlr.org/.

[2] Graphviz - graph visualization software. http://www.graphviz.org/.

[3] Install antlr as eclipse plugin. http://stackoverflow.com/questions/8343488/antlr-ide-in-eclipse-doesnt-work.

[4] Multistate fungal meningitis outbreak investigation. http://www.cdc.gov/HAI/outbreaks/currentsituation/.

[5] Senate banking committee, Gramm-Leach-Bliley Act, 1999. Public Law 106-102.

[6] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[7] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language (EPAL 1.2), 2003.

[8] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *IEEE Symposium on Security and Privacy*, pages 184–198, 2006.

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[10] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6:158–173, 2004.

[11] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977.

[12] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for history-based access control and reputation systems. *J. Comput. Secur.*, 16(1):63–101, 2008.

[13] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[14] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, volume 526, pages 46–67, 1977.

[15] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:2000, 2000.

[16] S. Stanchfield. Antlr 3.x tutorial. http://vimeo.com/groups/29150/page:2/sort:date.

[17] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[18] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, pages 179–193, 2004.

[19] W. H. Winsborough, J. von Ronne, O. Chowdhury, J. Niu, and M. S. Ashik. Towards practical privacy policy enforcement, 2011.

A Example HAPL code implementing a simple policy

In this section we will list a simple policy, an example HAPL code implementing the policy and the output Scala program.

The policy. If any parent asks for protected health information of their son/daughter then the doctor should give them the information if the child is not adult. In case the son/daughter is adult, the doctor is not permitted to give the information to the parent unless the patient has already approved it.

HAPL code. The HAPL program implementing this policy is shown here. In this program we have three actors - Medical, Doctor and Parent. The idea of implementing the policy is the parent actor sends a *requestHealthInformation* message to the Doctor. The Doctor actor then tries to find whether the patient is a child. It sends an *isChild* message to the Medical to determine it. The Medical actor then makes a query whether any actor acting on behalf of the patient has sent *childInfo* message to the Medical. It then replies the Doctor accordingly. If the Doctor actor gets positive feedback from the Medical actor, it then sends the information to the Parent actor via *getInformation* message. Otherwise, it asks the Medical whether the patient has given any approval beforehand to release her protected information to her parent. The Medical actor does another query to answer the Doctor actor's request. It sends the answer via a replyApproval message. If the Doctor gets a positive reply from the Medical then it sends the information to the parent. Otherwise, the Doctor actor replies the Parent with a *decline* message.

```
message isChild[β] (P[β]);
message hasApproval[β] (P[β], Actor);
message replyDoc[α] (P[α], bool);
message replyApproval[α] (P[α], Actor, bool);
message init[α](Actor, Actor, ProtectedInformation[α, heartDisease, bool]);
message requestHealthInfo[α] (P[α]);

message initParent[](Actor);
message getInformation[α](ProtectedInformation[α, heartDisease, bool]);
message decline[]();

actor Medical[MedicalSystem] (var doctor:Actor, var recList:List[(P[β],bool)])
        representing MedicalSystem {

  action isChild[α] (pat:P[α]) {
    ltlquery( E act. O send(act, this, childInfo( ))
           AND actsOnBehalfOf(act, pat)
          )
    {
      send(this, doctor, replyDoc(pat, true));
    }
    else{
      send(this, doctor, replyDoc(pat, false));
    }
  }
  action hasApproval[α] (pat : P[α], parent_pat:Actor) {
    ltlquery( E act. O send(act, this, approval(parent_pat))
           AND actsOnBehalfOf(act, pat)
          )
  {
   send(this, doctor, replyApproval(pat, parent_pat, true));
  }
  else{
   send(this, doctor, replyApproval(pat, parent_pat, false));
  }
 }
}

actor Doctor[theDoctor]
   ( var parent: Actor, var medical : Actor , var pat : P[myPat],
    var info : ProtectedInformation[myPat, heartDisease, bool]
   )
   representing theDoctor {

 var patientStatus = false;

 action init[myPat] (med:Actor, par : Actor,
     inf : ProtectedInformation[myPat, heartDisease, bool]) {
  info = inf;
  medical = med;
  parent = par;
 }

 action requestHealthInfo[myPat] (pat:P[myPat]) {
  send(this, medical, isChild(pat));
 }

 action replyDoc[myPat] ( pat:P[myPat], status:bool)
 {
  if(status == true){
   send(this, parent, getInformation(info));
  }
  else{
   send(this, medical, hasApproval(pat, parent));
  }
 }

 action replyApproval[myPat](pat:P[myPat], parent_pat : Actor, status : bool)
 {
  if(status == true){
   send(this, parent_pat, getInformation(info));
  }
  else{
   send(this, parent_pat, decline());
  }
 }
```

```
}

actor Parent[theParent]
       ( var doctor : Actor, var pat: P[alice],
         var information: ProtectedInformation[alice, heartDisease, bool]
       )
       representing theParent {

  action initParent[] (doc:Actor) {
    doctor = doctor;
    send(this, doctor, requestHealthInfo(pat));
  }
  action getInformation [alice]
         ( status : ProtectedInformation
           [alice, heartDisease, bool]
         )
  {
    information = status;
  }
  action decline[]()
  {

  }
}

object test {\n"+
  def main(args: Array[String]) {
    var p:Actor = null
    val pat = new P()
    val pi = new ProtectedInformation(false)
    val doctor1 = new Doctor(p, p, pat, pi)
    val medical1 = new Medical(doctor1)
    val parent = new Parent(doctor1, pat, pi)

    medical1.start
    doctor1.start
    parent.start
    doctor1 ! init(medical1, parent, pi)
    parent ! initParent(doctor1)
  }
}
```

Scala program. The Scala output program is shown below. We can see that there are one import declaration and several generated classes in the Scala program. The import supports the Scala actor.

HAPL has some types that Scala does not have such as, P, ProtectedInformation. Again HAPL has some construct that Scala does not support such as, pack, unpack, ltlquery, query expression.

The generated classes are there as either stub class or provides stub functions so that whenever anyone writes anything that HAPL supports but Scala does not then the conversion algorithm will point to the stub class or call the stub functions so that the Scala code remains compilable.

Each message in the HAPL program is converted to a case class in Scala. So, the action methods becomes a pattern match for the case classes.

```
import scala.actors.Actor

//##################### Generated classes #############################
class Packed[T](contents: T){ def unpack():T={return contents}}
class P(){ }
class ProtectedInformation(var a0:Any){ }
object Role{
  def setRole(p:P, role:String):Unit={}
  def inrole(p:P, role:String):Boolean={return true}
  def inrole(p:P, p2:P, role:String):Boolean={return true}
}
object Query{
  def ltlQuery(query:Q):Boolean={return false}
  def send(s:Actor, d:Actor, m:String):Unit={
    println(s+" is sending message ("+m+") to "+d)
  }
}
class Q(){}
class LV(d:String) extends Q{ }
class FV(d:Any) extends Q{ }
class Constant(d:String) extends Q{ }
class Forall(d:Any, q:Q) extends Q{ }
class Exists(d:Any, q:Q) extends Q{ }
class Once(q:Q) extends Q{ }
class Historically(q:Q) extends Q{ }
class Not(q:Q) extends Q{ }
class LtlSend(q1:Q, q2:Q, q3:Q) extends Q{ }
class LtlReceive(q1:Q, q2:Q, q3:Q) extends Q{ }
class Contains(d0:Any, d1:Any, d2:Any) extends Q{ }
class ActsOnBehalfOf(d0:Any, d1:Any) extends Q{ }
class Since(q1:Q, q2:Q) extends Q{ }
class And(q1:Q, q2:Q) extends Q{ }
class Or(q1:Q, q2:Q) extends Q{ }
class Message(v:Any*) extends Q{ }
//##################### end of generated classes #############################

case class isChild(a0:P)
case class hasApproval(a0:P, a1:Actor)
case class replyDoc(a0:P, a1:Boolean)
case class replyApproval(a0:P, a1:Actor, a2:Boolean)
case class init(a0:Actor, a1:Actor, a2:ProtectedInformation)
case class requestHealthInfo(a0:P)
case class initParent(a0:Actor)
case class getInformation(a0:ProtectedInformation)
case class decline( )
class Medical(var doctor : Actor) extends Actor
{
  def act() {
    while(true) {
      receive {
        case isChild(pat : P) =>
          if( Query.ltlQuery( new Exists(new LV("act"),
              new And(new Once(new LtlSend(new LV("act"),new FV(this),
              new Message("childInfo"))),
              new ActsOnBehalfOf("act", pat))))
             )
          {
            doctor ! replyDoc(pat, true)
            Query.send(this, doctor, "replyDoc")
          }
          else {
              doctor ! replyDoc(pat, false)
              Query.send(this, doctor, "replyDoc")
          }

        case hasApproval(pat : P, parent_pat : Actor) =>
          if( Query.ltlQuery( new Exists(new LV("act"),
              new And(new Once(new LtlSend(new LV("act"),new FV(this),
              new Message("approval",new FV(parent_pat)))),
              new ActsOnBehalfOf("act", pat))))
             )
          {
            doctor ! replyApproval(pat, parent_pat, true)
            Query.send(this, doctor, "replyApproval")
          }
          else {
              doctor ! replyApproval(pat, parent_pat, false)
              Query.send(this, doctor, "replyApproval")
          }

      }
    }
  }
}
class Doctor(var parent : Actor, var medical : Actor,
            var pat : P, var info : ProtectedInformation) extends Actor {
  var patientStatus = false
  def act() {
    while(true) {
```

```
   receive {
    case init(med : Actor, par : Actor, inf : ProtectedInformation) =>
     info = inf
     medical = med
     parent = par

    case requestHealthInfo(pat : P) =>
     medical ! isChild( pat )
     Query.send(this, medical, "isChild")

    case replyDoc(pat : P, status : Boolean) =>
     if( status==true) {
      parent ! getInformation( info )
      Query.send(this, parent, "getInformation")
     }
     else {
       medical ! hasApproval( pat, parent )
       Query.send(this, medical, "hasApproval")
     }

    case replyApproval(pat : P, parent_pat : Actor, status : Boolean) =>
     if( status==true) {
      parent_pat ! getInformation( info )
      Query.send(this, parent_pat, "getInformation")
     }
     else {
       medical ! decline( )
       Query.send(this, parent_pat, "decline")
     }

   }
  }
 }
}
class Parent(var doctor : Actor, var pat : P,
     var information : ProtectedInformation) extends Actor {
 def act() {
  while(true) {
   receive {
    case initParent(doc : Actor) =>
     doctor = doctor
     doctor ! requestHealthInfo( pat )
     Query.send(this, doctor, "requestHealthInfo")

    case getInformation(status : ProtectedInformation) =>
     information = status
```

```
        case decline() =>

       }
     }
   }
 }
object test {
  def main ( args : Array[String]){
    var p : Actor = null
    val pat = new P()
    val pi = new ProtectedInformation(false)
    val doctor1 = new Doctor(p, p, pat, pi)
    val medical1 = new Medical(doctor1)
    val parent = new Parent(doctor1, pat, pi)
    medical1.start
    doctor1.start
    parent.start
    doctor1 ! init(medical1, parent, pi)
    parent ! initParent(doctor1)
  }
}
```