Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Universitätsgelände Flugplatz, D-79110 Freiburg i. Br.
Germany
Email: basin@informatik.uni-freiburg.de

Grit Denker [1]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025
USA
Email: denker@csl.sri.com

Abstract

We compare two executable languages: the rewriting logic based specification language Maude and the higher-order, lazy, functional programming language Haskell. We compare these languages experimentally on a problem in modeling and reasoning about a security protocol for authentication. We explore differences in how models can be formalized and analyzed, as well as performance and tool use.

1 Introduction

Building and analyzing models of systems is at the heart of software engineering and many languages and formalisms have been proposed for these tasks. Rewriting logic has recently established itself as a powerful logical and semantic framework [23,25,26]. In rewriting logic, a model is formalized as a rewrite theory modulo equational theories. Rewriting logic, and its mechanization in systems like Maude [7,20], Elan [1,2] and Cafe-OBJ [4], has a number of strengths. These include support for structured theory specifications, algebraic data types and function specification in rich equational logics, and system support for built-in equations. The Maude system supports the

[1] Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

high-level formalization of models and their prototyping and analysis. Moreover, it allows reflective reasoning where rewrite theories are data manipulated by other theories, e.g., rules in one theory can be applied by strategies specified in another.

Of course, there are other executable languages that share many similarities with rewriting logic, such as data types and the specification of functions by equations. In this paper, we compare rewriting logic with one such competitor: Haskell, a modern functional programming language [15]. Haskell lacks some of rewriting logic's features, but offers others, such as higher-order polymorphic functions and lazy data types. There are also efficient tools for Haskell such as interpreters and compilers [16]. We compare these languages and how they can be used for building and analyzing models on a small but non-trivial problem: formalizing a model of a security protocol for authentication in a distributed environment and using infinite-state space model checking to find attacks. Rewriting logic has been shown to be particularly well-suited for specifying communication protocols, including those used for security [9,10]. It is interesting to see if Haskell can also rise to this challenge.

Our goal is to provide a critical investigation of the relative strengths and weaknesses of these two languages and their tool support. To achieve this, we have tried to keep our two formalizations as close to each other as possible. Our comparison then, is an analysis of the main differences, the reasons for them, and their effects. For example, we show that lazy data types in Haskell allow us to formalize infinite objects that must be specified using metaprogrammed approximations in Maude. And there are significant differences in higher-order versus reflective reasoning in formalizing and coordinating heuristics and search. On the tool side, we compare the development time and size of our formalisms as well as runtime performance.

We believe this work should be of interest to the rewriting logic community. In general, although there are results on how to embed other formalisms into rewriting logic [19,33], to our knowledge there are no in-depth comparisons with competing formalisms. We would like to encourage such comparisons since they help clarify myths and preconceptions about different approaches and bring to light limitations and advantages, as well as commonalities and differences. Moreover, comparisons help determine where progress is being made.

Our work should also be of interest to researchers working on formalizing and analyzing security protocols. In this community, formal methods play an important role, in particular, models where protocols are modeled as some kind of state transition system [3,5,11,18,22,27,28,32]. Although there has been work on comparing different paradigms on examples, there are few studies where the same model is analyzed in two different formalisms, as close as the different modeling languages allow. A better understanding of the differences between the various languages and tools would help protocol designers and consumers of formal methods, who must choose a tool best suited for their tasks.

Organization

The remainder of this paper is organized as follows. In Section 2 we provide background on Maude and Haskell. In Section 3 we explain the authentication protocol that we use for our case study and its model. In Sections 4 and 5 we present parts of our formalization. In Section 6 we compare the two formalizations and finally, in Section 7, we draw conclusions.

2 Background

Here we give brief overviews of rewriting logic and Haskell, concentrating on aspects relevant for our case-study and comparison.

2.1 Maude

Rewriting logic is an executable logic that extends algebraic specification techniques to concurrent and reactive systems. It is a simple, but expressive, logic that naturally models both static and dynamic aspects of communicating systems. Rewriting logic is mechanized in the Maude system. The Maude system supports the analysis of systems using metaprogramming, for example, prototyping systems by defining execution strategies in the logic itself [8].

We briefly outline the key syntactic features of Maude [6], an executable specification language based on rewriting logic, that are essential for our case study. In a Maude module one can declare *sorts*, *subsorts* (specifying sort inclusion) and *operators*. Operators are declared with the sorts of their arguments and result, and syntax is user definable using underscores '_' to mark the argument positions. Moreover, operators can have equational *attributes*, such as `assoc` and `comm`, stating, for example, that the operator is associative and commutative. These attributes are used by Maude to match terms modulo the declared axioms.

We will use logical axioms of two kinds: *equations*, introduced with the keywords `eq` or `ceq` for conditional equations, and *rewrite rules*, introduced with the keywords `rl` and `crl`. For more details on Maude, see [20].

2.2 Haskell

Haskell [15] is a general purpose, functional programming language incorporating many recent innovations in programming language design. These include higher-order functions, lazy evaluation, static polymorphic typing, user-defined algebraic data types, pattern matching, modules, and syntactic constructs such as list comprehension. There are good, freely available Haskell interpreters and optimizing compilers. We used the compiler GHC [16].

The most relevant language features for our study are lazy evaluation, data types, and list comprehension. *Lazy evaluation* is an evaluation strategy that ensures that expressions are expanded in a demand driven way and are not evaluated more than is necessary to provide a value at the "top level". For example, the head function `head` over lists (built from the empty list '`[]`' and cons '`:`') is defined, using pattern-matching, by `head (x:xs) = x`. Evaluating `head l` evaluates `l` only as far as is necessary to determine if the outermost constructor is a cons; the head and tail are not further evaluated.

Haskell *data types* are standard algebraic data types declared by naming the type and its constructors. As constructors are functions, their components are evaluated lazily; such types are sometimes called *lazy data types* to emphasize this. As an example, consider how we can model the infinite set of natural numbers:

```
nat = from 0
from n = n : from (n+1)
```

This Haskell program defines `nat` to be the infinite list `[0,1,2, ...]`. We can now evaluate the head of this list, which forces the computation of `from` far enough to produce the first element. Evaluation of the remainder is delayed until forced by further computation.

Haskell supports a notation for specifying (possibly infinite) sets using *list comprehension*, which is analogous to *set comprehension*. Sets in Haskell can be represented by lists. For example we can represent the set $\{2 \times x \mid 1 \leq x \leq 10 \wedge x \ mod\ 2 = 0\}$ in Haskell as

```
[2 * x | x <- [1 .. 10], x `mod` 2 == 0]
```

which is equal to $\{4, 8, 12, 16, 20\}$. The notation `[c | t <- xs, p]` represents a set where for each `t` in the list `xs`, for which the predicate `p` holds, we add an element `c` to the result set. In general, there can be more than one *generator* (expressions like `t <- xs`) as well as zero or more predicates. Moreover, in a generator, the term `t` can be a composite term, called a *pattern*, which is matched against each element of `xs`. Due to technical reasons, patterns in Haskell must be *linear*, which means that each variable in `t` can occur just once. One can often work around this by renaming common variables apart and adding predicates that equate them, e.g., translating the generator `[x,x] <- xs` (which selects all doubleton lists from `xs` whose elements are identical) to `[x,x1] <- xs, x == x1`. We will see applications of this shortly.

3 Modeling Security Protocols

3.1 Motivation and Background

A security protocol is a recipe that describes how *agents* (or *principals*) should act to achieve some goal. Protocols are often described using informal notation, for example as a sequence of instructions explaining the actions taken by the agents. For example, the following is a typical presentation of a protocol, in this case, a version of a public-key authentication protocol (NSPK) proposed by Needham and Schroeder [29]:

$$\begin{aligned} A \to B &: \ \{A, N_A\}_{K(B)} \\ B \to A &: \ \{N_A, N_B\}_{K(A)} \\ A \to B &: \ \{N_B\}_{K(B)} \end{aligned}$$

This protocol consists of three steps in which two agents, A and B, exchange messages in order to mutually authenticate each other.

A step $A \to B : X$ states that A exchanges the message X with B. Messages consist of atoms, like agent names and nonces (randomly generated strings), and are composed by tupling. Moreover, messages may be encrypted using an agent's key. Here, in the first step, A sends his name and a nonce N_A to B. The message is encrypted with B's public key $K(B)$. In the second step, B sends N_A back to A, along with his own nonce N_B, encrypted with A's public key. Sending N_A back authenticates B: for only B could return N_A, at least if we assume: (1) perfect cryptography, (2) that only B knows his private key, and (3) that other agents cannot guess the nonce N_A. B also sends along his own challenge, the nonce N_B, which A returns in the third step, thus demonstrating that she is really A.

Although security protocols are small and appear intuitive, this appearance is deceptive. The above protocol was proposed in 1978 and eighteen years went by before Lowe [17] discovered that, contrary to what was believed, the final step does not authenticate A because B can finish a run of the protocol with an agent who is other than she claimed to be in the first step. Moreover, in the attack given in [17], the attacker learns a nonce that is believed to be a secret between the two honest parties A and B.

The attack is a man-in-the-middle attack with the following steps.

$$\begin{array}{lcrl} A & \to & Spy: & \{A, N_A\}_{K(Spy)} \\ Spy(A) & \to & B: & \{A, N_A\}_{K(B)} \\ B & \to & A: & \{N_A, N_B\}_{K(A)} \\ A & \to & Spy: & \{N_B\}_{K(Spy)} \\ Spy(A) & \to & B: & \{N_B\}_{K(B)} \end{array}$$

An agent in the role `A` initiates (unknowingly) a communication with a malicious party (`Spy`), sending a nonce to the spy. The spy disguises himself as `A` and uses this nonce to fake a message to the agent in role `B`. `B`'s reply to `A` includes a second nonce and fits into the protocol session of agent `A`. Since `A` assumes that the second nonce came from the spy, `A` acknowledges its receipt by sending it back to the spy, encrypted with the spy's key. Thus, the spy learns the identity of the second nonce, which is intended to be a shared secret between the other agents. This violates the secrecy property of nonce N_B. The spy can use the nonce to send out the final message in B's protocol run, making B believe that he communicates with A. This violates the authentication property of NSPK. These kinds of errors, which can be exceedingly subtle, motivate the need for formal analysis.

3.2 An Asynchronous Knowledge-Based Model

A standard way of modeling a protocol is to abstract away any possible implementation of the steps and only model relevant aspects of the distributed system's *global state* and how the state evolves over time [5]. In our case, the global state provides an abstract snapshot of the system at some point in time and is comprised of the *local states* of each agent, the current *messages* in the network, and the knowledge of the attacker. A protocol then is modeled as a kind of "computation tree" of global states: the root models the system's starting configuration, and each node's successors model different ways that the global state can evolve by a step of (some run of) the protocol or interference by an attacker. Note that, in general, the tree is infinite: Agents may engage in arbitrarily many runs of the protocol (which can be interleaved with runs of other agents) and the attacker can send arbitrarily many messages.

Our model is an adaptation of that used in CIL [11], a specification language developed as an interface language for various protocol analysis tools. In this model, the global state consists of (cf. [5,28]):

(i) the knowledge of each agent (his status in different protocol runs),
(ii) the messages that have been sent but not yet received (this constitutes an asynchronous communication model), and
(iii) the knowledge of a possible attacker.

Modeling *asynchronous communication* means that a protocol step $A \to B : X$ is decomposed into two parts: A sending X to B, and B receiving X. This reflects the fact that communication in real networks is not atomic and all sent messages need not necessarily be received. *Agent knowledge* models, for each protocol run (an agent may take part in multiple runs of the protocol, playing different roles, i.e., A or B in NSPK): which step an agent is involved in; what his role is; and what information he has learned from the run. Finally, we will formalize the commonly used *attacker model* due to Dolev and Yao [13]: the attacker (also called the *spy*) controls the network and can read and intercept all messages. Moreover, he can construct messages from any information he can extract from messages previously sent on the network (extraction assumes perfect cryptography, i.e., he can only extract encrypted parts of messages when he has appropriate keys).

Our model provides a basis for formal analysis: A protocol has some property, precisely when the property holds for every global state in the tree (analogously, a protocol lacks some property precisely when the property fails for some global state in the tree). In the following sections, we present parts of our rewriting logic and Haskell formalizations of the NSPK protocol and how we formalize and check attacks.

4 Rewriting Logic Formalization

Below we provide some representative parts from our two specifications. Due to space limitations, we have omitted many details. Our intention is to give the reader a feel for the style of the specifications and provide a basis for the comparison afterwards.

4.1 Formalizing the Protocol

The most basic data type for cryptographic protocols is a field. For our application, a `Field` is either an agent's name, a nonce, or an encrypted message, which is a list of fields. Since we are interested in attacks other than those based on imperfect cryptography or guessing keys or nonces, we make some abstractions in specifying these data types. We identify an agent's name with his (public) key and we assume that only that agent has the inverse (private) key, unless it has been compromised. We also name nonces by integers. Here is the Maude specification of the data types involved:

```
sorts Nonce Agent Field Fields FieldList.
subsorts Agent Nonce < Field .
subsort  Field < Fields .

op nil : -> Fields .
op _,_ : Fields Fields -> Fields [assoc id: (nil).Fields] .
op [_] : Fields  -> FieldList .

op Nonce_ : MachineInt -> Field .
op Crypt : Agent FieldList -> Field .
```

There are three agents (including the spy) who can engage in the two roles `RoleA` or `RoleB`.

```
op Alice : -> Agent .
op Bob : -> Agent .
op Spy : -> Agent .

sort Role .
ops RoleA RoleB : -> Role .
```

We model the protocol state as a set of facts. A `Fact` is either the information on the local state of an agent (note that the agent is identified by the first entry in its field list and agent states are sequentially numbered), or a message between agents, or information "on the network", which can be used by the attacker.

```
sorts  Facts Fact FactSet .
subsort Fact < FactSet .

op [_] : FactSet -> Facts .
op mt : -> FactSet .
op _,_ : FactSet FactSet -> FactSet [assoc comm id: mt] .

op State : Role MachineInt FieldList -> Fact .
op Msg : Agent Agent FieldList -> Fact .
op Net : FieldList -> Fact .
```

As we model asynchronous communication, we formalize each step of NSPK by two rules: one for sending the message and the other for receiving it. Hence, the first step of the NSPK protocol is modeled by the following two rules:

```
rl [sndMsg1] :
   [State(RoleA,0,[a,b]),fs]
   => [State(RoleA,1,[a,b,mkNonce(fs)]),
       Msg(a,b,[Crypt(b,[a,mkNonce(fs)])]), fs] .

rl [rcvMsg1] :
   [State(RoleB,0,[b]), Msg(unknown,b,[Crypt(b,[a,na])]), fs]
   => [State(RoleB,1,[b,na,a]), fs] .
```

Whenever the global system state records that an agent `a` in `RoleA` is ready to start the protocol with an agent `b`, the rule `sndMsg1` can fire. The result is a new successor state where agent `a` records the freshly generated nonce and a message is added to the fact set. Note that as `a` and `b` are variables that are bound during matching, a successor is generated for every pair of agents that can start the protocol. Similarly, `rcvMsg1` formalizes that when some unknown agent has sent a message to an agent `b`, and `b` can enter the protocol in the role `RoleB`, then `b` can remove the message from the fact set and update his local state with its content. The variable `fs` represents all other facts in the fact set.

We also formalize transitions due to the attacker. Information that can be extracted from messages sent to, or intercepted by, the attacker are stored in the global state as data of the form `Net l` for list of fields `l`. We model possible attacker actions using eight additional rules that `intercept` messages (and place them on the net), `fake` new messages (2 rules), `decompose` messages on the net and place their parts on the net, `compose` messages from parts (2 rules), and `encrypt messages` (2 rules). We give two examples here: faking and encrypting messages.

```
crl [fake] :
```

```
    [Net([a]), Net(l), fs]
    => [Net([a]), Net(l), Msg(Spy,a,l), fs]
    if not(spyOf(a) == true) and not(Msg(Spy,a,l) in fs) .
crl [fake2] :
    [Net([a]), fs] => [Net([a]), Msg(Spy,a,[a]), fs]
    if not(spyOf(a) == true) and not(Msg(Spy,a,[a]) in fs) .

crl [encrypt] :
    [Net([a]), Net(l), fs]
    => [Net([Crypt(a,l)]), Net([a]), Net(l), fs]
    if not(Net([Crypt(a,l)]) in fs) .
crl [encrypt2] :
    [Net([a]), fs] => [Net([Crypt(a,[a])]), Net(l), fs]
    if not(Net([Crypt(a,[a])]) in fs) .
```

`fake` states that for an agent `a` on the network and for a message `l`, the message `Msg(Spy,a,l)` is inserted into the global state. The condition assures that the spy does not send a faked messages to himself and that the message is new. Note that we have adopted the convention of [30] where the message source and destination arguments are the true source and intended destination of the message, and they are not accessible to the receiver of the message. Hence, `Spy` is the sender here (and in `rcvMsg1` the sender is `unknown`). The rule `fake2` is similar but allows matching with only one state fact. The rules for `encrypt` select an agent `a` and a message `l` (which, in the second rule `encrypt2`, can be `a`) and add `l` encrypted with `a`'s key to the global state.

These rules formalize the (global) state transitions. The initial state is modeled as a fact set where `Alice` can start a run of the protocol, in the role `RoleA`, with the `Spy` and `Bob` can enter a run of the protocol in `RoleB`. Moreover, the names of all agents are on the net.

```
eq initial = [State(RoleA,0,[Alice,Spy]), State(RoleB,0,[Bob]),
              Net([Alice]), Net([Bob]), Net([Spy])] .
```

4.2 Model Checking

The aim of model checking is to explore the state space for possible attacks. In problems like ours where the state space is infinite, model checking based on enumerative search constitutes a semi-decision procedure.

We begin by characterizing what states represent attacks. For NSPK, a state represents a secrecy attack when an agent has sent his nonce, encrypted, to another agent (different from the spy) but the nonce has been compromised by the spy, i.e., is on the net.

```
op attack : -> Facts .
ceq [State(RoleB,2,[b,na,a,nb]), Net([nb]), fs] = attack
    if not(spyOf(a) == true) and not(spyOf(b) == true) .
```

By applying rewrite rules, Maude can be used to build parts of the computation tree rooted at the initial state. In particular, the Maude interpreter delivers *one* particular branch of the computation tree determined by interpreter's default evaluation strategy for applying rules. However, to find possible attacks we need to explore *all* possible branches.

We proceed by employing Maude's metalevel reasoning capabilities and define, at the metalevel (in Maude), a *strategy* that specifies how the rules should be applied. To do this we define, within a Maude module, a function that implements an iterative deepening search on a tree specified, implicitly, by an initial state and rewrite rules of another (object) module. We then specialize this to our initial protocol state and the module defining the protocol rules. In doing so, the protocol specification becomes a term on the metalevel that is passed to, and manipulated by, the search strategy.

Below we show part of our iterative deepening strategy. For a given state, this strategy performs a rewrite step that generates a successor state. At each step, we test for an attack or backtrack in case the branch terminates or the maximum depth is reached. To specify iterative deepening search, we formalize a function whose arguments are the module in which the reduction takes place (in our case the NSPK protocol), the current search path (i.e., a sequence of steps, where each step is a triple consisting of a rule label, a Maude-internal substitution number, and the new term), the current depth of the search tree, the maximum depths, and a list of protocol rule names (quoted identifier list `QIDL`).

```
ceq ids(M,path(APATH,step(L,N,T)),DEPTH,MAX,QIDL)
    = if DEPTH < MAX
      then (if nextRewrite(M, T, L, QIDL) == none
            then backtrack(M,path(APATH,step(L,N,T)),
                                DEPTH,MAX,QIDL)
            else ids(M,path(path(APATH,step(L,N,T)),
                                 nextRewrite(M,T,L,QIDL)),
                          DEPTH + 1,MAX,QIDL)
            fi)
      else backtrack(M,path(APATH,step(L,N,T)),DEPTH,MAX,QIDL)
      fi
    if T =/= {'attack}'Facts .
```

In theory, iterative deepening can find any attack, but in practice heuristics are needed to do so using manageable resources. Our heuristics are based on two simple ideas. First, the model contains *all* possible interleavings of agent/spy actions. It turns out that there is considerable redundancy: it is possible to ignore certain interleavings without removing any attacks from the search space. [12,31] give examples of such heuristics and prove that they are attack-preserving. Here, we formalize that certain action sequences must occur as a block (without other interleaved actions). For example, an agent receiving a message immediately sends the follow-up message. We formalize this with a predicate `extensionOf`, part of whose definition we present here (there is an equation defining `extensionOf` for each rule label). This predicate is called in order to test a possible new global state.

```
eq nextRewrite(M, T, QI, (L QIDL))
   = if (L extensionOf QI == true)
     then if meta-apply(M, T, L, none, 0) == error*, none
          then nextRewrite(M, T, QI, QIDL)
          else step(L, 0, term(meta-apply(M, T, L, none, 0)))
          fi
     else nextRewrite(M, T, QI, QIDL)
     fi .

eq QI extensionOf 'rcvMsg1
   = if (QI == 'sndMsg2) then true else false fi .
```

The second heuristic is that the actions of faking and composing messages are only meaningful if those messages can be received (i.e., if a message has the right format and there exists an agent in the system that can receive the message). Thus, we ignore faked messages that cannot be received. For example,

```
op _isReceivableIn_ : Fact FactSet -> Bool .
eq Msg(unknown,b,[Crypt(b,[a,na])]) isReceivableIn
   (State(RoleB,0,[b]), fs) = true .
```

states that a message is receivable when any agent sends a message to `b`, encrypted with `b`'s key, that contains an agent's name and a nonce, i.e., is in the format of the first message. In a similar way, we can specify messages resulting from steps 2 and 3 of the protocol as being the only other receivable messages, and ignore all non-receivable messages. Formats of messages are also defined as predicates on net facts.

Note that we implemented the second heuristic at the object level, as opposed to the metalevel. We defined the receivability predicate at the object level and extended the `fake`, `encrypt`, and `compose` protocol rules by conditions that test this predicate, i.e., we incorporated the test in the protocol rules.

As mentioned in Section 3.1 there is an attack for NSPK that violates the secrecy of the nonce sent by the agent in role B. This attack is found by Maude on ply 11 of the search tree using the following sequence of rules: `sndMsg1; intercept; decompose; encrypt; fake; rcvMsg1; sndMsg2; rcvMsg2; sndMsg3; intercept; decompose`. These eleven rules correspond to the actions described in lines one to four of the attack scenario in Section 3.1. The first two rules describe the first line: The attacker intercepts the message that is sent to him by `A`. The next two rules (`decompose` and `encrypt`) are local actions of the attacker where he sends a faked message to `B`. Line 2 to line 4 of the attack description are each mirrored by pairs of rules (line 2: `fake;rcvMsg1`, line3: `sndMsg2;rcvMsg2`, and line 4: `sndMsg3;intercept`). Finally, the attacker decomposes the information from the last message, which reveals `B`'s nonce.

5 Haskell Formalization

5.1 Formalizing the Protocol

As in the rewriting logic formalization, we begin by formalizing a data type, `Facts`, that encodes the global state.

```
data Agent = Alice | Bob | Spy
data Role = RoleA | RoleB
data Field = Agent Agent | Nonce Int | Crypt (Agent,[Field])
data Fact = State(Role,Int,[Field]) | Msg(Agent,Agent,[Field])
            |  Net [Field]
type Facts = Set Fact
```

Using lazy data types, we can explicitly formalize the infinite protocol computation tree. This is a tree of arbitrary branching, where each node contains an element of type `a` (where `a` is a type variable later instantiated by `Facts`) and a list of zero or more subtrees.

```
data Tree a = Node a [Tree a]
```

Similar to the Maude approach, each protocol step corresponds to two functions. We model these as "extension functions" of type `Facts -> [Facts]`: given a global state, each function specifies possible successor states after the action has occurred. Hence the first step is modeled by the following functions.

```
sndMsg1 fs =
   [ diff (Msg(a,b,[Crypt(b,[Agent a, Nonce na])])
          :State(RoleA,1,[Agent a, Agent b, Nonce na])
          :fs)
          [State(RoleA,0,[Agent a, Agent b])]
    | State(RoleA,0, [Agent a, Agent b]) <- fs]
   where na = mkNonce fs

rcvMsg1 fs =
   [ diff (State(RoleB,1,[Agent b, Nonce na, Agent a]):fs)
          [Msg(unknown,b,[Crypt(b, [Agent a,Nonce na])]),
           State(RoleB,0, [Agent b])]
     | Msg(unknown,b, [Crypt(b1,[Agent a, Nonce na])]) <- fs,
       b == b1,
       State(RoleB,0,[Agent b2]) <- fs, b == b2]
```

The formalization is similar to Maude's but uses set comprehension, instead of associative-commutative matching, to select facts in `fs`. For example, in `sndMsg1` we select (an instance of) `State(RoleA,0,[Agent a, Agent b])`. Given this, we build a successor state by adding to `fs` the facts `Msg(a, b, [Crypt(b,[Agent a, Nonce na])])` and `State(RoleA,1,[Agent a, Agent b, Nonce na])`, and deleting (using `diff`) `State(RoleA,0,[Agent a, Agent b])`. As mentioned in Section 2.2, equality constraints (guards like `b == b1`) arise due to Haskell's restriction to linear patterns. Note that, using matching, comprehension iteratively selects all such facts and, in doing so, generates a list of all possible extensions.

Next we formalize the five extension functions for our attacker model. Below are the functions for faking and encrypting messages.

```
fake fs = [Msg(Spy,a,l) : fs | Net [Agent a] <- fs,
                               not(spyOf(a)),
                               Net l <- fs,
                               Msg(Spy,a,l) `notin` fs]

encrypt fs = [ x:fs | x <- crypt, x `notin` fs]
    where crypt = [Net([Crypt (a,l)]) | Net [Agent a] <- fs,
                   Net l <- fs]
```

With `fake`, for example, we select from `fs` the name of an agent `a`, different from the spy, and a message `l` on the net, and insert `Msg(Spy,a,l)` into the global state if it is new. Again, set comprehension generates a list of all possible extensions, where each extension describes one such possible `fake`.
To formalize the protocol model itself, we employ a generic function `build_tree`, which builds a (possibly infinite tree) with root labeled by `init` and successors at each ply generated by a function `extension`.

```
build_tree extension init = Node init (map (build_tree extension)
                                           (extension init))

initial = [State(RoleA,0,[Agent Alice,Agent Spy]),
           State(RoleB,0,[Agent Bob]),
           Net([Agent Alice]), Net([Agent Bob]),
           Net([Agent Spy])]

nspk =  build_tree exts ("initial",initial)
   where  exts (_,fs) =  foldr (λ (l,f) res ->
                                (label l (f fs)) `union` res) []
              [("rcvMsg1",rcvMsg1), ("sndMsg1",sndMsg1),
               ("rcvMsg2",rcvMsg2), ("sndMsg2",sndMsg2),
               ("rcvMsg3",rcvMsg3), ("sndMsg3",sndMsg3),
               ("fake",fake), ("intercept",intercept),
               ("encrypt",encrypt), ("compose",compose),
```

```
            ("decompose",decompose)]
       label l fs = map (λ x -> (l,x)) fs
```

We complete our model specification by using `build_tree` to formalize the model `nspk`. The extension function iteratively applies (with `foldr`) each rule and unions together the results. The formalization is somewhat complicated by pairing each successor state with the name of the rule applied. This is a mere convenience for the heuristics we later introduce.

5.2 Model Checking

We have formalized an infinite-state model as a lazy tree. To perform infinite-state model checking based on lazy state-enumeration, we search the state space for global states that constitute attacks, constructing parts of the model on demand.

In Haskell we formulate the secrecy attack as follows:

```
secB (_,fs) =
  or [elem (Net [Nonce nb]) fs
      | State (RoleB,2,[Agent b,Nonce _,Agent a,Nonce nb]) <- fs,
        not (spyOf a), not (spyOf b)]
```

We implemented the identical heuristics used in Maude, although our formalization is quite different. We restrict interleavings (in an attack preserving way) using a generic function `filterPly` which takes a predicate `p` and a tree, and filters out all nodes (and their subtrees) where the predicate is false.

```
filterPly p (Node (l,a) ts) = Node (l,a) (filter (p l)
                                          (map (filterPly p) ts))

nspkOPT = filterPly pred nspk
   where extensionOf ("decompose",l)
           = l `elem` ["decompose","compose","encrypt","fake"]
         extensionOf ("compose", l) = l `elem` ["encrypt","fake"]
         extensionOf ("encrypt", l) = l `elem` ["fake"]
         extensionOf ("fake", l)
           = l `elem` ["rcvMsg1","rcvMsg2","rcvMsg3"]
         extensionOf ("rcvMsg1", l) = l `elem` ["sndMsg2"]
         extensionOf ("rcvMsg2", l) = l `elem` ["sndMsg3"]
         extensionOf (_,_) = True
         pred s (Node (l,x) _)  = extensionOf(s,l) && chCompose
            where chCompose = if (l == "compose" ||
                                  l == "encrypt")
                              then format (head x) else True
```

The predicate is based on a function `extensionOf` that specifies that only certain successor states are allowed, e.g., that receives are followed by sends, or that decomposing, composing, and encrypting, must all lead up to a faked message. We also achieve some pruning by employing the heuristic `chCompose` to check that a message built by a `compose` or `encrypt` extension will later be receivable after a `fake`.

We model check the optimized search space using iterative deepening search, `ids`, which iterates bounded depth first search with the bounds 0, 1, 2, ...

```
ids pred t = flatten [ idsn n pred t | n <- [0 ..]]

idsn 0 p (Node a _) = if p a then [a] else []
idsn n p (Node a l) = if p a then (a:rest) else rest
      where rest = flatten (map (idsn (n-1) p) l)
```

The result returned by `ids` is a stream (lazy list) of states for which `pred` holds.

With all the pieces in hand, we can now model-check. The command

```
head(ids secB nspkOPT)
```

applies iterative deepening search to the optimized tree, and `head` forces the computation of the first element of the stream, which is returned as the result. At ply 11 of the optimized tree, we find the same attack as in Maude.

6 Comparison

6.1 Qualitative

Here we compare the Maude and Haskell specifications with respect to three qualitative criteria: separation of concerns, reusability, and readability. These criteria were chosen to reflect how the languages are used and the qualities of the resulting specifications.

Separation of concerns: Conceptually, our approach to model checking can be subdivided into three tasks: specifying the model, formalizing heuristics, and search. Both approaches separate the tasks differently. In Haskell, lazy data types are central to the problem decomposition. They provide a direct formalization of the protocol model as an infinite tree, built from functions that describe the local state transitions. This formalization is independent of any notion of heuristics or search. Heuristics can then be seen as tree transducers that transform one (infinite) model into another. Search functions are simply generic tree search functions, i.e., the 4 lines specifying `ids` and `idsn`. Under this decomposition, we completely separate the tasks. During execution, lazy evaluation combines all three: search drives the coroutining of tree construction and heuristics to incrementally build and explore parts tree that are most likely to contain attacks.

The Maude specification is organized a bit differently. Whereas in Haskell all three tasks are implemented at one level, in Maude we use the object level and the metalevel for different tasks. The local state transitions are modeled as rewrite rules on the object level together with the cryptographic data types. Maude does not support lazy data type definitions like Haskell. One can locally declare evaluation strategies for operators, but full lazy data types require more sophisticated algorithms (see also Section 7). Instead we used metalevel search to build the computation tree up to some depth by the stepwise application of protocol rewrite rules. Heuristics are partly implemented as separate metalevel functions that are called at each rewriting step in order to decide whether a valid extension of the search tree is found. They are also partly implemented using additional conditions added to the protocol rules. In Haskell, we separated all heuristics from the protocol model. Nevertheless, our experiments suggested that in both approaches incorporating heuristics directly into the construction of the protocol model improves model checking performance (see the quantitative comparison below).

Reusability: As observed above, the specification can conceptually be decomposed into parts. By carrying out this decomposition in the specification language we can reuse the parts for other protocols. In Haskell, all three parts correspond to functions that are composed by straightforward function composition. Lazy data types play a role here in allowing us to formalize the model independently of heuristics and search. The function `build_tree` that constructs the model, and `ids` that conducts search, are completely generic. The specification of the protocol steps and heuristics are, of course, protocol specific. Moreover, standard higher-order functions (e.g., `map`, `filter`, and `foldr`) were used to build these parts from simpler functions.

Maude does not provide lazy data types or higher-order functions. As an alternative, we used parameterized modules and reflective programming to decompose the specification. The search strategy for instance, a metalevel specification that takes a module as an argument, can also be reused for other protocols.

Both parties did a fair amount of debugging, and here the Haskell specification proved more flexible. By using generic combinators, the various parts were easier to adapt for adding debugging information, changing output parameters, and the like. In Maude the search strategy encompasses about 100 lines of code and a change in the parameters can cascade through the entire specification, thus increasing the development time for extended strategies.

Readability: The two languages offer different possibilities for modeling, which showed their comparative advantages on different parts of the specification. The use of built-ins in Maude and, in particular, support for user defined syntax and rewriting modulo lead to a simpler and more intuitive encoding of the protocol rules than in Haskell. In Maude, associative-commutative matching lead to a very simple and elegant way of expressing how the global state is modified by removing and adding information. In contrast, this was achieved in Haskell using iterated set comprehension, which amounts to a technical device to simulate pattern matching patterns modulo associativity and commutativity.[2] Moreover, the Haskell formalization was complicated by technical restrictions concerning linearity of variable occurrences.

In contrast, formalizing the heuristics and search was more straightforward in Haskell. The use of higher-order functions played a large role here. So did the use of standard lazy programming paradigms. For example, programming with "streams of successes" leads to a simple formalization of search: we formulate search without backtracking by enumerating all attacks and later forcing (using `head`) the computation of just one attack. Finally, specifying everything at just one level (avoiding metaprogramming) made things simpler.

6.2 Quantitative

We now compare quantitatively our two developments. This provides some indication of the relative ease of modeling and system performance.

However, we begin with a caveat: care must be taken in interpreting the results. Different metrics favor different systems and metrics reflect only part of the story. Moreover, some metrics are surprisingly difficult to measure accurately. For example, the two developments took place over an approximately two week period but we found it difficult to accurately attribute development time because each of the authors had experience formalizing protocols in their respective formalism and could reuse parts of previous developments. Moreover, a considerable part of the development was spent trying to match the other's model as closely as possible. Even the simple metrics we give are not problem free. For example, the number of lines is influenced by layout conventions and there is a tradeoff between development time (e.g., spent optimizing the model), clarity of the specification, code size, and execution time.

Our first metric is code size, measured in lines of specification.

	Total	Libraries	Stripped	Modeling	Heuristics	Search
Maude	727	159	398	216	77	105
Haskell	313	20	224	88	24	5

Total measures the lines in the entire specification, including comments, but excluding auxiliary libraries (measured in *Libraries*). The Haskell specification imports definitions of standard functions like `map`, `filter` and `elem`. In Maude we import `META-LEVEL`. *Stripped* is the size of the specification removing all comments. In the case of Haskell we also removed type annotations, which serve as a kind of comment (since types are not necessary as the compiler infers them automatically). The last three columns compare the size of the model, and the size of the heuristics and search procedures. Note that the distributions are not quite comparable. In Haskell, the code for modeling

[2] And also idempotence, since two instances of comprehension can match the same term. As a result, we were able to encode attacker actions using one rule in Haskell where, in some cases (like `fake` and `encrypt`), the Maude formalization required two.

consists of both the rules and the model specification with `build_tree`. For Maude, it is only the rules. Rule application takes place during search and the code for building the computation tree is counted there.

Overall we can conclude that the Maude protocol specification is about twice the size of the Haskell counterpart. This is partially due to the fact that Maude currently requires all operations, variables and their types to be defined. The other significant difference concerns object level (in Haskell) versus metalevel (in Maude) specification of search.

As previously explained, the development times of the two approaches are hard to measure. From our experience it generally takes about 1.5 times longer to develop Maude code. In particular, the initial design of the search strategy and adaptations for different variants of those strategies were much more time consuming in Maude than in Haskell. This is in part because we had to write Maude code from scratch for standard routines such as traversing data structures.

At loading time, the Maude NSPK specification uses 30MB. This includes about 1MB for Core Maude, 25MB for the structured version of Full Maude, and the remainder for the meta representation of our protocol. During run-time an additional 2MB are used for search. The Haskell program is quite thrifty with memory. The Haskell compiler produces a C program, which is further compiled by gcc. The resulting binary uses only 2MB of memory when searching for the attack.

By design, both approaches result in a search space of the same size. The following table shows how many nodes are in the computation tree up to the ply listed, both with and without our pruning heuristics.

	1	2	3	4	5	6	7	8	9	10	11
Size unoptimized	26	807	31162	1478096	?						
Size optimized	2	3	4	12	37	44	51	66	159	379	652

Below we list execution times, measured in seconds, where the nth column records the time spent searching a tree of depth n and the final column is the time for an iterative deepening search at all levels until an attack is found (at depth 11). Test are performed on a 500 MHz Pentium III workstation with 512 MB memory and 512 KB cache. Note that for depths 1–10, the entire tree is (unsuccessfully) searched. The low run-time at depth 11 is because, due to the heuristics, the attack is found early on during search.

	1	2	3	4	5	6	7	8	9	10	11	all
Haskell unoptimized	.01	.11	5.8	349								
Maude unoptimized	.2	1.2	55	2391								
Haskell optimized(1)	.01	.01	.02	.04	.25	1.2	1.5	1.7	2.3	7.8	.13	7.8
Haskell optimized(2)	.01	.01	.01	.01	.02	.05	.06	.07	.08	.21	.11	.22
Maude optimized	.01	.03	.05	.07	.24	.41	.44	.48	1.9	3.0	.17	3.5

We have included two sets of timings for Haskell optimized with heuristics. The first is for the heuristics incorporated as explained in Section 5 where the unoptimized protocol tree is first (lazily) generated, and afterwards heuristics are applied to prune subtrees. The results surprised us. For the unoptimized case, Haskell is significantly (factor 6-10) faster, and this also holds at first in the optimized case. But what happens after depth 5? The problem turned out to be somewhat subtle. It is true that lazy evaluation brings together generation and testing during search. However, in the optimized case, on deeper plys the vast majority of the generated nodes are pruned. Following generation by testing means we must first allocate memory for the successors, only to look at them once, and then prune them. To solve this problem, we modified our specification so that heuristics are incorporated as they are in Maude: we test before generation, i.e., in the extension function itself we first test whether a composed, encrypted, or faked messages could ever be received. And if it isn't receivable, we do not generate it as an extension. This kind of optimization is analogous to deforestation transformations, which reduce execution time in functional languages by avoiding building intermediate data structures. Applying this optimization here had a drastic effect on run-time, as displayed in the second set of Haskell timings. Again, the conclusion is that compiled Haskell is significantly faster than interpreted Maude.

7 Conclusion

It shouldn't be too surprising that each approach has its advantages and disadvantages. The two languages were each designed with particular domains in mind where they will excel. Protocol analysis serves as a good case study as it brings out a number of interesting differences. For example, for specifying transition systems, rewriting logic, with its notion of rewriting-modulo, leads to an exceedingly perspicuous formulation. This support is lacking in Haskell where we ended up approximating such rewriting by using set comprehension and auxiliary functions. Conversely, higher-order programming with lazy data types offer the ability to decompose the specification in ways sometimes not feasible in Maude.

In both formalisms, one can learn from the counterpart, both in terms of extensions to the language and libraries. Simple changes to Haskell could make rules more readable. It is a relatively simple matter to lift the linearity restriction in patterns (by insisting that variables range over types belonging to the equality class). Further support for Maude-like specifications of protocol rules could be implemented by programming a rule compiler, which would take protocol rules in Maude-like syntax and generate specifications of the form we have given. But it is difficult to achieve the generality of Maude without making major changes to the language, such as adding in matching modulo equational theories.

On the Maude side, the comparison shows that such developments would benefit from library support for common data types and standard metafunctions for search. Efforts are currently underway to design general purpose search strategies for Maude that can be applied to model checking analysis. Type reconstruction, planned for the next Maude release, would also ease the design efforts. There are also interesting questions concerning how easily the elegance of higher-order programming can be simulated using parameterized modules in Maude. [14] suggests possibilities in this direction. Finally, Maude supports locally declared lazy evaluation strategies for operators. We plan to explore how this feature can be used in combination with strategies to provide the equivalent of lazy data types.

References

[1] P. Borovanský, C. Kirchner, and H. Kirchner. Strategies and Rewriting in ELAN. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, 1997.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A Logical Framework Based on Computational Systems. In Meseguer [24].

[3] S. Brackin. Automatically Detecting Most Vulnerabilities in Cryptographic Protocols. In Maughan et al. [21], pages 251–266. `http://schafercorp-ballston.com/discex/`.

[4] CafeOBJ Project. *Proc. of the CafeOBJ Symposium '98, Numazu, Japan.* April 1998.

[5] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A Meta-Notation for Protocol Analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, 1999.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* SRI International, Computer Science Laboratory, Menlo Park, CA, January 1999. `http://maude.csl.sri.com/manual/`.

[7] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [24], pages 65–89.

[8] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In Meseguer [24], pages 125–147.

[9] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[10] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience. In Maughan et al. [21], pages 251–266. `http://schafercorp-ballston.com/discex/`.

[11] G. Denker and J. Millen. CAPSL Integrated Protocol Environment. In Maughan et al. [21], pages 207–222. `http://schafercorp-ballston.com/discex/`.

[12] G. Denker, J. Millen, J. Kuester-Filipe, and A. Grau. Optimizing Protocol Rewrite Rules of CIL Specifications. In *13th IEEE Computer Security Foundations Workshop (CSFW'2000), 3-5 July 2000, Cambridge, United Kingdom*, pages 52–62. IEEE Computer Society, 2000.

[13] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983.

[14] J. Goguen. Higher-order functions considered unnecessary for higher-order programming. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, Massachusetts, 1990.

[15] S. P. Jones and J. H. (editors). Report on the Programming Language Haskell 98. www.haskell.org/definition/haskell98-report.ps.gz.

[16] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, March 1993.

[17] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[18] G. Lowe. Casper: A Compiler For the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[19] N. Martí-Oliet and J. Meseguer. Rewriting Logic As a Logical and Semantic Framework. In Meseguer [24], pages 189–225.

[20] Maude Web Site. `http://maude.csl.sri.com/`, 2000.

[21] D. Maughan, G. Koob, and S. Saydjari, editors. *Proc. DARPA Information Survivability Conference and Exposition, DISCEX2000, January 25-27, Hilton Head Island, SC, USA*. IEEE Computer Society Press, 2000. `http://schafercorp-ballston.com/discex/`.

[22] C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[23] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Comput. Sci.*, 96(1):73–155, 1992.

[24] J. Meseguer, editor. *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996.* Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

[25] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report. In U. Montanari and V. Sassone, editors, *Proc. 7th Intern. Conf. on Concurrency Theory: CONCUR'96, Pisa, August 1996*, pages 331–372, 1996. LNCS 1119.

[26] J. Meseguer. Research Directions in Rewriting Logic. In U. Berger and H. Schwichtenberg, editor, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, pages 347–398. Springer Verlag, 1998.

[27] J. Millen and H. Rueß. Protocol-Independent Secrecy. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[28] J. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–154. IEEE Computer Society, 1997.

[29] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[30] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[31] V. Shmatikov and U. Stern. Efficient Finite State Analysis for Large Security Protocols. In *11th IEEE Computer Security Foundations Workshop, Rockport, Massachusetts, June 1998*, pages 106–115. IEEE Computer Society, 1998.

[32] D. Song. Athena, an Automatic Checker for Security Protocol Analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 192–202. IEEE Computer Society, 1999.

[33] M.-O. Stehr and J. Meseguer. Pure Type Systems in Rewriting Logic. In *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-languages, Paris, France, September 28, 1999*, 1999.