RESEARCH ARTICLE

Proof-carrying authorization in distributed systems with Beluga: a case study

Leah Weiner[1]*

[1] School of Computer Science, McGill University, Montreal, QC

*Email Correspondence:
leah.d.weiner@gmail.com

Abstract

The rising popularity of distributed systems creates a need for a secure method of message passing. One approach to access control is accomplished through proof-carrying and proof authorization. The requesting party must provide a proof of authorization of access while the serving party must verify the validity of the proof. PCML$_5$ is a programming language which enables a programmer to encode proofs and perform proof-checking procedures more easily.

The purpose of this project is to encode PCML$_5$ with Beluga, a new functional programming language which supports dependent types and which has built in some of the more common procedures. Such an implementation will provide formal guarantees regarding the language of PCML$_5$ itself. Moreover, this case study will aid in the understanding of the development and implementation of software specically for programming with proofs. It will provide insight into the tools needed to allow any programmer to easily specify and verify complex behavioral properties of programs.

Introduction

In order to understand the research laid out in this report, one must have a solid understanding of functional programming and of security in distributed systems. This report begins with a brief description of lambda calculus, a formal calculus of functions and the basis for functional programming. Following the discussion of lambda calculus and functional programming is a section devoted to introducing Beluga, the functional programming language in which the code for this project was written.

Next is an introduction to distributed systems and security within these systems. PCML$_5$, a prototype programming language which this project implements, is introduced together with the motivations behind implementing PCML$_5$ in Beluga.

Finally, there is a detailed description of the actual steps taken towards the implementation of PCML$_5$ in Beluga.

Lambda Calculus

The lambda calculus is a formal system intended to represent and to calculate with functions. These functions can take other functions as input and/or return functions as output. Such functions are called *higher-order functions*. Lambda terms are the foundation of this system and are defined as follows:

Variables: A variable is a lambda term

Lambda Abstraction: If M is a lambda term and x is a variable, then $\lambda x.M$ is a lambda term

Application: If M and N are lambda terms, then $M\ N$ is a lambda term

The lambda term $\lambda x.M$ represents the function $f(x) = M$, where M may or may not be dependent on the variable x. The identity function for example is represented as $\lambda x.x$. Observe that variable names are unimportant. $\lambda x.x$ and $\lambda y.y$ are equivalent functions. This equivalence is known in lambda calculus as *alpha-equivalence*.

Binding

Consider the lambda term $\lambda x.M$. If x occurs in the expression M, then x is said to be *bound to M*. A variable that is not bound by a lambda abstraction is said to be *free*. The free variables of a lambda term can be deduced by the following inductive rules:

Let x be a variable and let M and N be two lambda terms. Then

$FV(x) = \{x\}$

$FV(\lambda x.M) = FV(M) \setminus \{x\}$

$FV(M\ N) = FV(M) \cup FV(N)$

where $FV(M)$ denotes the free variables of the lambda term M.

Substitution

In lambda calculus, substitution is used to mimic function application. Consider the two lambda terms $\lambda x.M$ and N, and application is represented on paper as $M\ N$. During this application, every free occurrence of the variable called "x" (the variable that is bound to M) which occurs in the body of M, is being substituted by the lambda term N. We write $[N/x]M$ to represent this substitution.

Capture-avoiding substitution

When performing substitutions as described above, certain precautions must be taken. In order to properly perform the application $M\ N$, only free variables should be substituted (bound variables should not be touched). However, in cases when variable names are not distinct, performing substitution may result in the binding of a variable which was previously free. An example of this phenomenon, called name capture, is described below. One way to avoid name capture during substitution is to rename the free variables (the variables affected by substitution) in the relevant terms uniquely. This safe substitution is known as *capture-avoiding substitution.*

To see the importance of renaming, consider the following example: In the expression $\lambda x.xz$, z is the only free variable. However, to perform the substitution $[x/z]\ \lambda x.xz$, the result, before renaming would be $\lambda x.xx$. In this way, z has been "captured", its distinction from the variable x has been lost.

The correct way to perform this substitution is to first rename the free variables. Through alpha-equivalence, $\lambda x.xz == \lambda y.yz$. Now, $[x/z]\ \lambda y.yz = \lambda y.yx$, which is the intended result (7).

Functional Programming

Functional programming is a programming paradigm based on the ideas of lambda calculus. Functional programs use function evaluation as the basis for computation. The better known imperative programming languages, such as Java or C, rely on mutating data and changing the state of the machine rather than function evaluation.

One consequence of functional programming's reliance on function evaluation is its extensive use of recursion. The more familiar imperative programs rely more heavily on iteration. Writing code in a recursive fashion generally takes less space than iterative code, allowing functional programs to have code which is cleaner and easier to read.

Functional programs also avoid side effects. That is, observable changes such as writing data to disk or changing variable names are bypassed during computation. Conversely, imperative programs do use side effects (also referred to as referential opaqueness), so the output of imperative code can change depending on the current state of the system. Functional programs, which avoid this dependence on the current state of the machine to determine the output, do not output results dependent on the state of the machine and are therefore more predictable and thus easier to reason about.

Lastly, functional programming has the property that each expression has a single value (referential transparency) (10). The use of referential transparency ensures that each expression has a unique value, eliminating confusion and giving the programmer a greater sense of control.

Recall that substitution of lambda terms must be capture-avoiding. Issues such as name capture also arise when performing function application in functional programs. Functional programs then must be able to generate fresh variable names, rename variables and perform capture-avoiding substitution appropriately. Although these procedures are extremely common in functional programming, most functional programming languages require these procedures to be manually implemented.

Beluga

Beluga is a newly developed pure functional programming language which provides a novel programming and proof environment. The motivation behind the development of Beluga was the desire to provide an environment in which to set up formal systems and proofs and in which the aforementioned lack of support for the common operations in functional programming languages (capture-avoiding substitution, renaming, fresh name generation) is remedied.(4).

While Beluga successfully achieves automatic support for these operations, it does not include support for input and output operations, references or exceptions.

Types are the foundational tools used to build programs. Built-in types common to many programming languages include integer, character or double. Beluga does not have any predefined data types but it does support dependent types, or types which depend on other variables.

Because Beluga does not have a predefined type system, a Beluga programmer must declare his own types. For example, the integer type in Beluga can be initiated with the following piece of code:

```
nat : type.
zero : nat.
succ : nat -> nat.
```

The first line declares that natural numbers (called nat) are a type and the second line declares that zero is a natural number. The third line demonstrates how one can achieve natural numbers other than zero. This declared successor type, succ, is a dependent type. It's value is dependent on the value of the inputted nat. In the case of successor, succ N = N + 1 for any natural number N.

To get a better understanding of what code looks like in Beluga, we provide an example of the simple addition function below:

```
rec add: [. nat] -> [. nat] -> [. nat] =
fn x => fn y =>
   case x of
   | [. zero] => y
   | [. succ N] =>
        let [. R] = add [. N] y in [. succ R]
  ;
```

In line one, the keyword rec announces the start of a new (recursive) function. The name of this function is "add" and it takes in two variables of type nat and returns a variable of type nat.

Line two gives the input nat values arbitrarily chosen variable names. In this example, the first inputted natural number is called "x" and the second is called "y".

In the third line of this function, the programmer declares that there will be a case by case analysis on the value of "x", the first input to the add function. As x is a natural number, it is either equal to zero, or it is of the form succ N, where N is a natural number. It is important to notice that a natural number of the form succ N cannot equal zero.

The next line deals with the case when x equals zero. In this case, the function is told to return y. This makes sense, as 0 + y = y. This case acts as the base case in this recursive function.

The fifth line deals with the case when x is of the form succ N. It declares R to be a new variable equal to N + y. The successor of N + y is then returned as the final output to the function (after more recursive calls). To check the validity of this, notice that succ (N + y) = N + y + 1 and that x = succ N = N + 1. Thus, x + y = succ N + y = N + 1 + y = N + 1 + y, as desired.

Distributed systems

A distributed system is a set of computers which communicate through a network. Although each machine is independent, they collectively appear to an outsider (user) as one coherent system. The use of a distributed system is motivated by a system's reliability, performance and transparency.

Having a large number of machines working together clearly increases the reliability of the system as a whole. If one part fails, another is available to take over, resulting in a higher tolerance for faults. Similarly, the cooperation of a series of machines leads to the better performance of distributed systems when compared to a single centralized machine. Moreover, it is easier to share data and communicate when using a distributed system (8).

There are some disadvantages however. Writing software for a distributed system is more challenging. This difficulty may stem from the fact that each component of the distributed system may be written in a different language, that consistency across the system is hard to achieve, or that testing software in a distributed system is difficult. Also, despite a distributed system's higher fault tolerance, the greater amount of components means that more parts are prone to failure (but, as previously stated, the failure of a single piece of the system is less urgent). Additionally, data security is sometimes an issue (11).

Authentication and authorization

Certain information being passed along a network may be authorized only for the eyes of certain machines. It is therefore necessary for a distributed system to have some sort of access control. In order to grant a machine access to some set of data, it is necessary to ensure two things. Firstly, *authentication*, or certification of the identity of the requester, must be determined. Secondly, *authorization*, or the decision of allowing or disallowing the requester access to the desired information, must be figured. The set of rules by which this decision is made is called the *policy* (6).

There are two methods for access control. One possibility is having a central authority. Each request must first go through this central authority, which either approves or denies the client's request. Upon approval, the central authority tells the server to send the desired information directly to the requesting machine.

Alternatively, we can use the proof-carrying authorization method of access control wherein the middle man is cut out and the client can send a proof of authorization of access directly to the serving party. It is then the job of the server to verify this proof and, if applicable, provide the requester with the desired information. Here, the burden of proof is on the client, rather than on the server.

Systems which utilize proof-carrying authorization are both easier to implement and more efficient. In systems that don't use proof-carrying authorization, the verifier must decide whether or not to grant access to a requester. Programming such communications is difficult and time-consuming. However, in the method of proof-carry authorization, the requester can simply supply in its proof the reason why it should be granted access. Thus the method of proof-carrying authorization is faster and more straight-forward. While authorization of a proof in systems which use proof-carrying authorization takes time which is linear in the size of the proof, the verification or denial of access of a machine is much more time-consuming (6).

How to represent proofs

In order to make use of the method of proof-carrying authorization, it is imperative to be able to have a language which can represent proofs and policies. Proofs need to be represented as some sort of digital certificate. But in order for clients and servers to write and interpret proofs correctly, the language system must be unambiguous and must be easily dissected and understood by all the machines. This is difficult due to the complexity of most policies. Each distributed system has its own policy, and within a system, each set of data could have its own policy.

The problem of creating a language with which to encode the policy is discussed in the next section. Once this language is established however, implementing the language in a way that is understandable to a computer follows from converting each rule of the policy into an inference rule and representing proofs rigorously as a logical inference (6).

PCML$_5$

The growing popularity of distributed systems in the form of web-based services motivates the creation of a language in which policies can be encoded and distributed programs can be expressed (9).

Avijit, Datta and Harper introduced a programming language, PCML$_5$, in their paper entitled *Distributed Programming With Distributed Authorization* (9). PCML$_5$ is intended to simplify the programming of proofs for security in a distributed system with distributed authorization. It achieves this through its support for distributed computations, the implementation of an authorization logic, and the ability to represent both policies and proofs (9).

An authorization logic is a formal system of logic which supports the encoding of policies as theories. PCML$_5$ encodes an authorization logic by integrating the framework of the logic directly into PCML$_5$'s type system. For example, we can use an authorization logic to prove progress and preservation. The proof of progress states that if m is an expression, then either m is a value or m steps to some other expression m'. The proof of preservation states that if m is of type T and m steps to m' then m' is of type T. Together, progress and preservation ensure type safety.

Why implement PCML$_5$ in Beluga?

The successful implementation of PCML$_5$ in Beluga provides formal guarantees about PCML$_5$ itself. Specifically, we can gain insight into the theorems such as progress and preservation which together ensure type safety of the language. This case study will also aide in the understanding of both the development and implementation of software systems and programming with proofs. Implementing PCML$_5$ in Beluga raises awareness about possible issues that may arise during these sorts of implementations. Moreover, we will gain insight into the tools which are necessary to allow the average programmer to easily specify and verify complex behavioral properties.

Why Beluga?

Beluga is not the only language which can accomplish the desired task successfully. However, the use of Beluga is motivated by some of the properties inherent in Beluga programs as well as by a desire to fill some of the gaps in our knowledge of Beluga.

One advantage to using Beluga is its support for dependent types and its lack of pre-defined types. These properties allow for a simpler implementation of security procedures as defined by PCML$_5$, as we can simply define the types as prescribed by PCML$_5$. PCML$_5$ is designed to encode authorization policies and certificates easily. Beluga's support for automatic capture-avoiding substitution and renaming opens the Beluga framework to the ability to program with authorization policies and certificates, making Beluga sufficient for this project.

Moreover, this case study will aide in the understanding of some of the more technical issues regarding this task, as it is a future goal to implement security procedures in Beluga. A successful implementation of PCML$_5$ in Beluga will show that security is viable and will show simplicity of proof-carrying applications due to Beluga's built-in features. Upon successful implementation, we will have another working example of Beluga code, proof that Beluga can be learned quickly by the average programmer with no previous knowledge of Beluga, and a simpler way in which to write security procedures in Beluga. Although PCML$_5$ requires common substitutions and renaming, a Beluga programmer can essentially ignore these issues, thus encode PCML$_5$'s authorization logic more simply.

How to implement PCML$_5$ in Beluga

In order to implement PCML$_5$ with Beluga, one must first encode the primitive types. Beluga has no predefined data types, so this was all done by hand. An example of a piece of the authorization logic which PCML$_5$ encodes is expressed as:

Kinds K ::== Wld | A_1 Affirms A_2

Constructors A ::== α | A_1 x A_2 | $A_1 \rightarrow A_2$

These primitive types can be represented in Beluga as follows:

```
kind : type.                    %type kind
cnstc : type.                   %type constructor
wld : kind.
affirms : cnstc -> cnstc -> kind.
alpha : cnstc.
cross : cnstc -> cnstc -> cnstc.
arr : cnstc -> cnstc -> cnstc
```

Next, I implemented the inference rules. In order to do this, we make use of the fact that substitution and renaming is automatic in Beluga. Dependent types in PCML$_5$ (e.g. lam, exists, pi etc.) are thus easily converted into Beluga. When using these functions, a Beluga programmer need not think twice.

An example of an inference rule expressed in PCML$_5$'s logic is:

$$\frac{m{:}A_1{\times}A_2@w}{\pi_2 m{:}A_2@w}$$

The statements above the horizontal line are the assumptions, while the statement below the horizontal line is the conclusion following the assumption of the statements above the line. This inference rule states that if m is of type $A_1 \times A_2$ (and thus $m = (m1, m2)$) in the world w then the second element (second projection) of m is of type A_2 in world w.

To translate this inference rule into Beluga's language, we first must have previously declared the following types:

```
cnstc : type.
term : type.
world : term.
cross : cnstc -> cnstc -> cnstc.
scnd : term -> term.
```

We then implement the "sentence structure" with which Beluga will represent $m : A @ w$:

```
is_inwld : term ->  cnstc -> cnstc -> type.
```

Finally, we can represent the inference rule itself in Beluga as:

```
isinwld_scd : is_inwld M (cross A_1 A_2) W ->
              is_inwld (scd M) A_2 W.
```

Lastly, we can make use of our previously implemented primitive types and inference rules to implement the main theorems of the paper. Proving the theorems involves a case by case analysis on the derivation of some statement in the hypothesis of the theorem.

One theorem that is proven is the progress theorem. In the PCML$_5$ paper, the statement of the theorem is written as:

If $m : A@w$ then either $mval_A$ OR $\exists m', A'$, such that $m \,;\, A \rightarrow w\;\; m' \,;\, A'$.

In plain English, this translates to: if m is of type A in world w then either m is a value under the assumptions A or there is some m' and A' such that m evaluated under A in the world w steps to some expression m' under A'.

In order to encode this statement into Beluga, we must first somehow encode this "either" in the statement. We can do this by creating a type called result, which encodes that the result is either a value or an evaluation (world shift):

```
result : term -> type.
r_val : {W:cnstc}{AA:active_prin} is_val W M AA
   -> result M.
r_ws : is_ws M AA W M’ AA’ -> result M.
```

With this extra tool, we can now state the theorem in Beluga by following the rules for function declaration:

```
schema apctx = active_prin ;
rec thmc4 :
   {AA:[.active_prin]} [. is_inwld M A W] ->
   [. result M] =
   mlam AA => fn d => case d of
.....
```

In the proof of this theorem, we proceed by a case by case analysis on the derivation of `is_inwld M A W`.
In the appendix of Avijit, Datta and Harper's paper, each possible derivation is followed by a systematic reasoning as to why the theorem would hold true under this derivation of the hypothesis.

For example, consider the derivation of `is_inwld M A W` , written in the appendix as:

Case: $\frac{\Delta;\cdot\vdash^{A}_{\Sigma_c;\Sigma_t} m{:}A_1\times A_2@w}{\Delta;\cdot\vdash^{A}_{\Sigma_c;\Sigma_t} \pi_2 m{:}A_2@w}$

$\Delta;\cdot\vdash^{A}_{\Sigma_c;\Sigma_t} m : A_1\times A_2@w$ (Premise)
Either $m\ val_A$ (I.H.)
or $\exists m', A, .m; A \mapsto_w m'; A'$

Subcase: $mval_A$ (Lemma C.5)
$m =< m_1, m_2 >$
$pi_2, m_1, m_2.; A \mapsto_w m_2; A$

Subcase: $m; A \mapsto_w m'; A'$
$\pi_2 m; A \mapsto_2 \pi_2 m' : a'$

We can represent this in Beluga as:

```
| [. isinwld_scd D] => let [. R] = thmc4 [.AA] [. D] in
   (case [. R] of
    | [. r_ws T] => [. r_ws (isws_scd T)]
    | [. r_val W AA' V] =>
       (case [. V] of
        | [. isval_xV V1 V2] => [. r_ws (isws_pairscd V)]
      ))
.....
```

The Lemma C.5 that is referred to in the derivation analysis is a weakening lemma. In Beluga, it is not necessary to implement the lemma itself, and it can be implemented directly into the function (as can be seen above in the case of r_{val}). Moreover, it is not necessary to explicitly mention the impossible cases of [. V], the value. There is only one possible way the statement can be true under the assumptions of that case and so only that case needs to be expressed.

Moreover, we do not need to represent the contexts. They are either constant or can be easily included directly in the statement. This removes a lot of notation in Beluga's implementation.

Similar translations are done for all the possible derivations.

We then continue and implement a proof of preservation in Beluga. As described in PCML$_5$:

Theorem C.11 (Preservation of consistency). *Let $\Sigma_c;\Sigma_t$ be well-formed signatures, and Φ be an access control theory. Further, let all API constants declared in Σ_t be consistent in the sense of Def. C.7. If $m; A \mapsto^{\Sigma_c;\Sigma_t;\Phi}_{w} m'; A'$ and $\Delta;\cdot \mapsto^{A}_{\Sigma_c;\Sigma_t} m : A@w$, then $\Delta, \Phi, \cdot \mapsto^{A'}_{\Sigma_c;\Sigma_t} m' : A@w$*

Proof: Assume $m; A \mapsto^{\Sigma_c;\Sigma_t;\Phi}_{w} m'; A'$ and $\Delta;\cdot \mapsto^{A}_{\Sigma_c,\Phi;\Sigma_t} m : A@w$ We proceed by induction on the derivation of $m; A \mapsto^{\Sigma_c;\Sigma_t;\Phi}_{w} m'; A'$

This means if *m* is evaluated under *A* in world *w* to *m′* under *A′* and *m* is of type *m* in world *w* then *m′* is of type *A* in world *w*. This can be implemented in Beluga using a similar method as the previous proof. The statement of the theorem is written as follows:

```
rec thmc11 : [. is_ws M AA W M' AA'] ->
   [. is_inwld M A W] ->
   [. is_inwld M' A W] =
fn d => fn f => case d of
....
```

We will not review how to implement each specific case here, as it is the same methodology as the preservation cases.

Benefits of using Beluga

Following the successful implementation of PCML$_5$ in Beluga, it becomes clear that Beluga offers an advantageous environment in which to implement PCML$_5$ and encode proof-carrying code. The Beluga framework leads to a comparatively concise encoding of both the authorization logic and the theorems laid out in the PCML$_5$ paper. One factor leading to this concise representation is the elimination of the need to repeatedly represent the contexts. It was unnecessary to encode impossible or irrelevant cases, cutting down greatly on the amount of notation.

Moreover, PCML$_5$ required an implementation of separate lemmas for inversion, weakening, canonical forms, substitution or renaming. In Beluga however, these are either automatic or could be encoded directly in the function.

To get a more concrete idea of how much more concise PCML$_5$ is when implemented in Beluga, consider the exact amount of code needed. There is no way to save space when implementing the primitive types of PCML$_5$'s authorization logic in Beluga; each primitive type takes one line to encode. However, since in most cases we do not need to encode the contexts, we save a lot of symbolism when using Beluga to write out the inference rules and the theorems. The inference rules each take one line in Beluga. Due to the fact that we do not need to implement supplementary lemmas in order to successfully prove the progress and preservation theorems in Beluga, these full proofs of these theorems take up less space on the page. Moreover, had we used another functional programming language to implement $PCML_5$'s authorization logic, we would have had to implement these lemmas since the operations are not automatic in most other programming languages.

To get a more concrete understanding of the amount of space saved, consider that the eight pages it took the appendix of the paper to fully prove the progress and preservation theorems that took Beluga a mere 250 lines of code (approximately five pages).

Conclusions

The successful implementation of $PCML_5$ in Beluga further motivates the continuing development of Beluga. The successful implementation of $PCML_5$ in Beluga demonstrates that using Beluga to code proof-carrying authorization in a distributed system can be quite simple. Beluga allows for code to be concise, and its framework is compatible with proof-carrying code. Moreover, Beluga is an accessible and useful language that can be eventually used universally. Lastly, this project offers insight into the development of proof-carrying authorization in distributed systems.

References

[1] A. Felty and B. Pientka Reasoning with Higher-Order Abstract Syntax and Context: A Comparison. In First International Conference on Interactive Theorem Proving (ITP '10), Lecture Notes in Computer Science (LNCS), vol 6172, pages 227-242, Springer, 2010.
[2] A. W. Appel and E. W. Felten. Proof-Carrying Authentication. In CSS '99: Proceedings of the 6th ACM conference on Computer and communications security, pages 52-52, Aug 1999.
[3] B. Pientka. "A Beginner's Guide to Programming in Beluga." McGill University. Sept 2010 <http://www.cs.mcgill.ca/~cs523/handouts/intro-beluga.pdf>
[4] Brigitte Pientka. Beluga: programming with dependent types, contextual data and contexts. In 10th International Symposium on Functional and Logic Programming (FLOPS '10). Lecture Notes in Computer Science (LNCS), vol 6009, pages 1-12, Springer, 2010.
[5] Deepak Garg and Frank Pfennig. Non-Interference in Constructive Authorization Logic. In Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19), pages 283-296, 2006.
[6] Deepak Garg. An Introduction to Proof-Carrying Authorization. Nov 2007.
[7] Henrik Nilsson. "Untyped λ-calculus: Operational Semantics and Reduction Orders". Lecture 11. University of Nottingham. July 2012. <http://www.cs.nott.ac.uk/~nhn/G54FOP/LectureNotes/lecture11-9up.pdf>.
[8] Isup Lee. "Introduction to Distributed Systems". Univeristy of Pennsylvania. 2007. <http://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch1-DistSys-v4.pdf>
[9] Kumar Avijit, Anupam Datta, and Robert Harper. Distributed Systems With Distributed Authorization. In TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation, pages 27-38, ACM, 2010
[10] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A Proof-Carrying Authorization System. Technical Report TR-638-01, Princeton University, April 2001.
[11] Petru Eles. "Distributed Systems". Linkoping University. <http://www.ida.liu.se/~TDDB37/lecture-notes/lect1.frm.pdf>
[12] Stephen A. Edwards. "Functional Programming and the Lambda Calculus". Columbia University. 2008. <http://www.cs.columbia.edu/~sedwards/classes/2010/w4115-spring/functional.pdf>