POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Matemáticas e Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Embedding Logical Frameworks in Scala

Autor: Daniel Espino Timón

Supervisora: Nada Amin

Director: Martin Odersky

LAUSANNE, JUNIO DE 2014

Contents

1 Resumen

El Framework Lógico de Edimburgo ha demostrado ser una poderosa herramienta en el estudio formal de sistemas deductivos, como por ejemplo lenguajes de programación. Sin embargo su principal implementación, el sistema Twelf, carece de expresividad, obligando al programador a escribir código repetitivo. Este proyecto presenta una manera alternativa de utilizar Twelf: a través de un EDSL (Lenguaje Embebido de Dominio Específico) en Scala que permite representar firmas del Framework Lógico, y apoyándonos en Twelf como backend para la verificación, abrimos la puerta a diversas posibilidades en términos de metaprogramación.

El código fuente, así como instrucciones para instalar y configurar, está accesible en https://github.com/akathorn/elfcala.

2 Abstract

The Edinburgh Logical Framework has proven to be to be a powerful tool in the formal study of deductive systems, such as programming languages. However, its main implementation, the Twelf system, lacks expressiveness, requiring the programmer to write repetitive code. This project presents an alternative way of using Twelf: by providing a Scala EDSL (Embedded Domain Specific Language) that can encode Logical Framework signatures and relying on Twelf as a backend for the verification, we open the door to different possibilities in terms of metaprogramming.

The source code, along with instructions to install and configure, is accessible at https://github.com/akathorn/elfcala

3 Introduction and objectives

The Edinburgh Logical Framework has been proven to be a powerful tool in the formal study of deductive systems, such as programming languages. However its main implementation, the Twelf system, lacks expressiveness, requiring the programmer to write repetitive code, and a module system, making difficult to reuse and organise code.

This project presents an alternative way of using Twelf. By providing a Scala Embedded Domain Specific Language (EDSL) that can encode Logical Framework signatures and relying on Twelf as a backend for the verification, we open the door to different possibilities in terms of metaprogramming and code reuse.

In addition, it provides groundwork for further developing the system in order to achieve similar capabilities to those provided by Twelf, effectively reducing the dependency on it.

Section 4 provides a brief introduction to LF and its core type theory, followed by a short presentation of Twelf and its limitations. In section 5 the EDSL is described along with some of the design decisions taken for its implementation. Section 6 analyses in more detail how to overcome some of the limitations in Twelf by using metaprogramming. Related work is discussed in section 7. Finally, section 8 concludes this report and suggests what could be the next steps in further developments of this project.

4 The Edinburgh Logical Framework

In this section, a brief introduction to the Edinburgh Logical Framework (LF) is presented, along with a description of its most successful implementation, Twelf. Some limitations of Twelf are then described, leading to an explanation of the usefulness of an alternative implementation embedded in Scala.

Readers familiar with LF and Twelf might want to skip this section. For a full description of LF, please refer to the original paper [4]. For more information about Twelf, you can read the paper describing the system [7] or directly go to the official wiki [1].

4.1 Description of LF

The Edinburgh Logical Framework provides a way to encode logics and reason about them. It has been proved useful in the study of programming languages, as it allows them to be represented and reasoned about.

The logic being defined is usually called *object logic*. An object logic is defined by a list of declarations that we call *signature*. We will usually use both terms interchangeably.

One of the strengths of LF is its *judgements as types* principle, which states that judgements in the object logic can be represented as types in LF. Therefore, checking the correctness of a given proof is equivalent to type-checking the LF term describing it. This also means that proof-search can be used to evaluate expressions in the object logic.

4.1.1 Type theory

The underlying logic in LF consists of a dependently-typed λ-calculus. Terms are stratified in three different levels: *Kinds*, *Families* and *Objects*.

We use K, L to denote Kinds, A, B, C for Families, and M, N, P for objects. a, b, c are used to represent constants and x, y for variables.

The abstract syntax is given by the following grammar:

$$
\begin{array}{ll}
\textit{Kinds} & K ::= \mathit{Type} \mid \Pi x : A.K \\
\textit{Families} & A ::= a \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\
\textit{Objects} & M ::= c \mid x \mid \lambda x : A.M \mid MN
\end{array}
$$

Terms of the form $\Pi x : A.B$ can be written as $A \to B$ if x does not appear free in B, both for family and kinds. The $\to$ operator is right associative.

Signatures are used to keep track of the types and kinds assigned to constants, and *contexts* are used to keep track of the types assigned to variables. The syntax for signatures and context is given by the following grammar:

$$
\begin{array}{ll}
\textit{Contexts} & \Sigma ::= \langle\rangle \mid \Sigma, a : K \mid \Sigma, c : A \\
\textit{Signatures} & \Gamma ::= \langle\rangle \mid \Gamma, x : A
\end{array}
$$

The type theory of LF gives also a notion of *valid signature* and *valid contexts* by means of type checking rules [4]. This will not be discussed in depth in this report.

4.2 Twelf

Work on the Edinburgh Logical Framework led to the implementation of Twelf.

Twelf provides a concrete syntax for LF signatures. The right column gives the corresponding LF abstract syntax term (or terms).

$$
\begin{array}{lll r}
\textit{Expressions} & e ::= & \mid c \mid x & a, c, x \\
 & & \mid \{x : e\}\ e & \Pi x : A.B, \Pi x : A.K \\
 & & \mid [x : e]\ e & \lambda x : A.M, \lambda x : A.B \\
 & & \mid e\ e & AM, MN \\
 & & \mid \mathit{type} & \mathit{Type} \\
 & & \mid e- > e \mid e < -e & \\
 & & \mid \{x\}\ e \mid [x]\ e \mid \_ \mid e : e \mid (e) & \\
\textit{Signatures} & \multicolumn{2}{l}{sig ::= empty \mid c : e.\ sig} &
\end{array}
$$

Figure 1 shows an example of how we can define the natural numbers in Twelf.

```
nat : type.
z : nat.
s : nat -> nat.

plus : nat -> nat -> nat -> type.
plus/z : plus z N N.
plus/s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

Figure 1: Natural numbers in Twelf

The type *plus* defines a relation between three natural numbers, representing the addition.

By using this syntax, Twelf allows to easily encode an object logic. In addition, it provides means to do logic programming [5] (by allowing to execute queries) and to do proofs about meta properties of the encoded language, such as termination, mode, and totality. For this, special commands starting with $\%$ are used.

By automatically checking the totality of a given relation, it is possible to use Twelf to prove theorems of the form $\forall a_1 : A_1, a_2 : A_2 \ldots \exists b_1 : B_1, b_2 : B_2$.

4.2.1 Limitations of Twelf

One of the main limitations of Twelf is the absence of polymorphism. Many definitions have the same shape save for some small pieces, and this means that while programming in Twelf you end up copy/pasting very often. This not only makes Twelf code rather verbose and redundant, but it also makes it difficult to maintain.

As an example, consider the definition of a list of natural numbers:

```
nat-list: type.
nat-nil: nat-list.
nat-cons: nat -> nat-list -> nat-list.
```

Following the same pattern, we could easily define lists for other types, but we have to rewrite the whole definition:

```
exp-list: type.
exp-nil: exp-list.
exp-cons: exp -> exp-list -> exp-list.
```

This becomes a more problematic issue if we also want to make proofs about lists, as then we would have to write the same proof for any type we want to have a list for.

Instead, we would like to write something more general, such as:

```
list: type.
nil: list.
cons: t -> list -> list.
```

And then we want to be able to instantiate it for different types t.

Furthermore, it should be possible to write proofs for this generic list, and instantiate it for different types without having to rewrite everything.

More complex examples can be constructed in which several parameters are involved, and in which we not only want to define a generic type but the generic "shape" of a proof.

In section 6 we explain how we can use metaprogramming in our EDSL to write polymorphic LF code that can be passed to Twelf to be checked.

5 Embedding in Scala

One of the powerful features of Scala is the ability to define Embedded Domain Specific Languages in an elegant way. Using techniques such as implicit conversions makes it easy to provide a readable syntax for an underlying representation.

5.1 Core type theory

Finding an appropriate way to encode an LF signature opens the door to many possibilities in terms of metaprogramming. This will prove to be useful in order to overcome the limitations of Twelf described in section 4.2.1.

Figure 2 shows how the LF abstract syntax can be encoded in Scala. Using case classes to represent the LF terms provides a straightforward and concise encoding.

It is not practical for a programmer to describe LF signatures by constructing the terms at this low level, and for this purpose a higher level syntax is presented in section 5.2. However, the use of a limited number of case classes makes it easier to encode rules by pattern matching the terms.

A Scala implementation of LF typechecking is provided. It implements the rules presented in [4], which can be easily encoded by using pattern matching over the possible terms. The implementation also includes a reducer for LF terms that is used to check if two terms are equal.

For both the typechecker and the reducer the representation of LF terms proved to be adequate.

At this moment, the system lacks the ability of inferring types and therefore it is more verbose than Twelf, as every variable has to be given a type explicitly. This will become clear in the examples.

5.2 Higher level syntax

While the representation of terms is very simple and easy to work with at low level, it is hard to define whole signatures with it. For this purpose, a higher level syntax is provided.

In order to make the core representation simple, consistent, and easy to extend, it is important that this layer of abstraction does not interfere with the lower level representation. Thus, we are effectively implementing *syntax sugar* for the core type theory.

```
// Variables
case class Variable(x: Name)
case class Constant(c: Name)

// Terms
abstract class Term

// Kinds
abstract class Kind extends Term

object Kind {
  case object Type extends Kind
  case class Pi(x: Variable, a: Family, k: Kind) extends Kind
}

// Families
abstract class Family extends Term

object Family {
  case class Const(d: Constant) extends Family
  case class Pi(x: Variable, a: Family, b: Family) extends Family
  case class Abs(x: Variable, a: Family, b: Family) extends Family
  case class App(a: Family, m: Object) extends Family
}

// Objects
abstract class Object extends Term

object Object {
  case class Const(c: Constant) extends Object
  case class Var(x: Variable) extends Object
  case class Abs(x: Variable, a: Family, m: Object) extends Object
  case class App(m: Object, n: Object) extends Object
}
```

Figure 2: LF Abstract Syntax encoded in Scala

In order to define a signature in Scala, a class, trait, or object has to be defined by extending the *Signature* trait provided by the implementation. Figure 3 provides an example for the encoding of natural numbers presented in figure 1 in which the types of the variables are not fully specified. This means that it will not type according to the checker implemented in Scala, but later we will show how this representation can be typechecked by Twelf.

Figure 4 shows how we can fully specify the type of the variables. The consequence of doing so is that this example can be successfully typechecked directly by the Scala implementation.

```
trait Naturals extends Signature {
  val nat = |{ Type }
  val z   = |{ nat }
  val s   = |{ nat ->: nat }

  val plus   = |{ nat ->: nat ->: nat ->: Type }
  val plus_z = |{ plus (z) (‘n) (‘n) }
  val plus_s = |{ plus (‘n1) (‘n2) (‘n3) ->: plus (s(‘n1)) (‘n2) (s(‘n3)) }
}
```

Figure 3: Natural numbers encoded in Scala

```
trait Naturals extends Signature {
  val nat = |{ Type }
  val z   = |{ nat }
  val s   = |{ nat ->: nat }

  val plus   = |{ nat ->: nat ->: nat ->: Type }
  val plus_z = |{ !!(‘n, nat)/ { plus (z) (‘n) (‘n) } }
  val plus_s = |{ !!(‘n1, nat) (‘n2, nat) (‘n3, nat)/
                   { plus (‘n1) (‘n2) (‘n3) ->: plus (s(‘n1)) (‘n2) (s(‘n3)) } }
}
```

Figure 4: Natural numbers encoded in Scala, without type inference

5.3 Integration with Twelf

The implemented software allows to interact with a subprocess running a Twelf server instance. Through this interface, signatures defined in the Scala representation of LF can be checked, with the powerful addition of the meta proof checking capabilities provided by Twelf.

An extended trait offers the possibility to use Twelf to prove totality for a given relation. There are three Twelf commands involved in this: *%mode*, *%worlds* and *%total*. For example, the following Twelf code:

```
%mode term +A -B ...
%worlds () (term _ _ ...)
%total (A) (term A _ ...)
```

Can be written in the Scala EDSL as:

```
% { mode (term) ++(A) --(B) ... }
% { worlds (term) (?) (?) ... }
% { total (A) (term) (A) (?) ... }
```

This declarations can be included in a signature definition, and they will be transformed conveniently and passed to Twelf to be checked.

In the rest of this document, we will often assume that the Twelf backend is used for typechecking, and thus we rely on type inference and omit most of the types for variables.

6 Metaprogramming

Embedding the Logical Framework in Scala opens the door to many interesting possibilities in terms of metaprogramming, as we can write code to programmatically construct signatures and declarations.

6.1 Generics

Perhaps one of the most powerful features presented here is the ability to define *generics*.

In the context of this project, a generic is a set of LF declarations that depends on parameters left to be specified later. In this section we will first present how they can be used through several examples, and then some details about the implementation are discussed.

6.1.1 Generic list

As we saw in section 4.2.1, given a type we can define a list for such type in Twelf. Figure 5 shows how we can encode a generic list in Scala.

```
val genericList = generic { t =>
  val list = |{ Type }
  val nil  = |{ list }
  val cons = |{ t ->: list ->: list }
}
```

Figure 5: Generic list in Scala

The function *generic* is provided by the *Signature* trait. It takes as parameter an anonymous function representing the generic definition, and it produces an object with methods corresponding to each LF declaration contained in the object.

We can instantiate the generic by applying the resulting object to another LF type:

```
genericList(nat)
genericList(exp)
...
```

The object provides methods to access the definitions inside the generic. For instance, if we want to reference the *nil* for lists of natural numbers, we can do it as

```
genericList.nil(nat)
```

Usually, it is more convenient to import all definitions. As an example, suppose we want to define the size of a list of natural numbers:

```
import genericList._

val nat_list_size      = |{ list(nat) ->: nat ->: Type }
val nat_list_size_nil  = |{ list_size (nil(nat)) (z)  }
val nat_list_size_cons = |{ list_size (L) (N) ->:
                            list_size (cons(nat) (?) (L)) s(N)) }
```

The variable “?” is equivalent to the Twelf symbol “_”, which means that we don’t care about the value or type of the variable in the position where we use it.

Clearly, this definition applies for every list, not only a list of natural numbers. Thus, we can also make it generic:

```
val genericListSize = generic { t =>
  import genericList._

  val list_size      = |{ list(t) ->: nat ->: Type }
  val list_size_nil  = |{ list_size (nil(t)) (z)  }
  val list_size_cons = |{ list_size (L) (N) ->:
                          list_size (cons(t) (?) (L)) s(N)) }
}
```

Note that when we are inside the definition of *genericList*, we use *list*, *nil* and *cons* without giving the generic parameter, while in *genericListSize* we have to use them as *list(t)*, *nil(t)* and *cons(t)*. This is because the system can’t assume that the parameter t is the same for the declarations imported from *genericList*. If we enclose all the declarations inside the same generic definition it will be assumed that the generic parameter is the same, as it is shown in figure 6.

```
val genericList = generic { t =>
  val list = |{ Type }
  val nil  = |{ list }
  val cons = |{ t ->: list ->: list }

  val list_size      = |{ list ->: nat ->: Type }
  val list_size_nil  = |{ list_size (nil) (z)  }
  val list_size_cons = |{ list_size (L) (N) ->:
                          list_size (cons (?) (L)) s(N)) }
}
```

Figure 6: Generic list and size in Scala

6.1.2 Generic proofs

In LF theorems are represented as types, and proving the theorem is equivalent to showing a term of the type of the theorem. Theorems of the form $\forall a_1 : A_1, a_2 : A_2 \ldots \exists b_1 : B_1, b_2 : B_2$ will additionally require to prove totality with respect the arguments $a_1, a_2 \ldots$.

Some theorems share the same “shape” and proof schema. In Twelf one would have to copy and paste the code, but we can do better by using generics. The idea is to define the shape of the theorem as a generic, and then instantiate it when necessary to produce each of the theorems we want.

One example of this is what we call *recursive extensions*. Given a context, represented as a list, we want to prove that, for certain relations, if the relation holds for a context it will also hold if we extend the context an arbitrary number of times. In other words, if it holds for a sublist of a list L, it will hold for L.

To illustrate this, let us consider two different relations that have this property. Assume we have two relations: *lkp* (for look-up) and *ev* (for evaluation in an environment). The look-up takes a natural number and retrieves the element in the corresponding position, starting from the end of the list. The evaluation takes an expression (a term of type *exp*) and evaluates the result in the given context.

The type of this relations, in Scala syntax, is the following:

```
val lkp = |{ list(t) ->: nat ->: t ->: Type }
val ev  = |{ list(t) ->: exp ->: t ->: Type }
```

Here we omit the rest of the signature, for a complete example please refer to the project implementation [3].

For both relations, we would like to prove that if they hold for a list L_1, it will also hold for any list L_2 such that L_1 is a sublist of L_2.

The type of this theorem in LF would be:

```
val exts = |{ rel (G1) (I) (X) ->: sub_list(envtype) (G1) (G2) ->:
              rel (G2) (I) (X) ->: Type }
```

Where *rel* can be either *lkp* or *ev*.

Assuming that we have a proof *ext* for a single extension (adding one element at the beginning of the list), a proof of the theorem would be the following:

```
|{ exts (A) (sub_list_rfl(envtype)) (A) }
|{ ext (B0) (?) (B) ->:
   exts (A) (S) (B0) ->:
   exts (A) (sub_list_ext(envtype) (S)) (B) }
```

The first declaration corresponds to the base case, and the second one to the induction step. To effectively prove this as a theorem, we also need to check the mode, termination, and totality of the relation given by *exts*.

Now, we can put together the proof inside a generic:

```
val recursiveExtension = generic { (rel, ext, envtype) =>
  val exts = |{ rel (G1) (I) (X) ->: sub_list(envtype) (G1) (G2) ->:
                rel (G2) (I) (X) ->: Type }
  % { mode (exts) ++(A) ++(S) --(B) }
  |{ exts (A) (sub_list_rfl(envtype)) (A) }
  |{ ext (B0) (?) (B) ->:
     exts (A) (S) (B0) ->:
     exts (A) (sub_list_ext(envtype) (S)) (B) }

  % { worlds (exts) (?) (?) (?) }
  % { total (S) (exts) (?) (S) (?) }
}
```

And instantiate it to prove the theorem for both *lkp* and *ev*:

```
recursiveExtension(lkp(nat), ext_lkp(nat), nat)
recursiveExtension(ev, ext_ev, nat)
```

Where *ext_lkp* and *ext_ev* are the one-step proof of the extension for *lkp* and *ext*, respectively.

Now we can typecheck this signature using Twelf, and prove the theorem for both relations. To accomplish this in Twelf, we would have to write the theorem for one of the relations, and then copy and modify it for the other one.

6.1.3 Implementation

The generics are implemented by using another powerful characteristic of Scala, namely macros.

Macros are an experimental feature that allows to analyse and transform a given Scala Abstract Syntax Tree (AST) at compile time.

By using this feature, we can implement the function *generic* as a macro to transform a set of LF declarations into an object that instantiates them for different types them when necessary.

For every LF declaration in the generic, a method takes as parameters the generic parameters is produced. Each of this methods checks whether the current generic has been already instantiated for the given parameters. If not, the declarations will be generated and added to the current signature.

Furthermore, the resulting object can be applied to a given parameter to generate all definitions enclosed by the generic.

As an example, the definition of a generic list show in figure 5 would yield code similar to the following after macro expansion:

```
val genericList = {
  trait Generic {
    private var defined: Set[List[Family]] = Set.empty

    private def define_family(t: Family) = {
      val list = ...
      val nil  = ...
      val cons = ...

      defined = defined + List(t)
    }

    def apply(t: Family) =
      if (!(defined contains List(t))) {
        define_family(t)
      }

    def list(t: Family) = {
      if (!(defined contains List(t))) {
        define_family(t)
      }
      Symbol(t + "_list")
    }

    def nil(t: Family) = {
      if (!(defined contains List(t))) {
        define_family(t)
      }
      Symbol(t + "_nil")
    }

    def cons(t: Family) = {
      if (!(defined contains List(t))) {
        define_family(t)
      }
      Symbol(t + "_cons")
    }
  }

  new Generic {}
}
```

7 Related work

Although there is no previous work on implementing a Logical Framework as an EDSL, the John Boyland's Twelf library [2] implements a method to overcome some of the limitations of Twelf.

The library manages to simulate a module system by using a C preprocessor to put together several signature files. Furthermore, it implements "functors" as a mechanism to generalise types in the same spirit of what we call generics in this project. This is achieved by defining variables that will be substituted by the preprocessor by the actual types, inlining them in the signature code.

8 Conclusions and future work

This project doesn't provide an alternative to Twelf, as fundamental features such as proof-search, type reconstruction, and metaproving capabilities are not implemented. However, integration with Twelf makes it possible to benefit from the metaprogramming environment provided by the Scala EDSL while retaining the mentioned features from Twelf.

This integration can be further extended by allowing the possibility of importing Twelf code to be seamlessly integrated in a Scala-defined signature. One approach to do this would be by implementing a Twelf parsing and using macros to generate Scala signatures at compile time, thus making it possible to import and use them. This would also simplify the task of testing Twelf code, as Scala testing libraries could be used to check the Scala signatures produced at compile time. If parsing capabilities are provided, the Twelf back-end could be extended to provide access to the query functions implemented in Twelf, as the result produced by said queries could be easily parsed back into Scala.

Future work in terms of approaching a Twelf-like system would start by implementing type reconstruction, as this would greatly simplify writing code to be checked by the Scala backend. Additionally, proof-search in the fashion of [5] can be implemented and, if extended with mode and termination checking [9] it would make possible to possible to prove metatheorems by using schema-checking as described in [6]. Having mode and termination checking would also make it possible to implement the $\mathcal{M}_2$ meta-logic system [10], which is used by Twelf to check totality assertions.

The programming workflow could be improved by typechecking at compile time (using macros), thus making it possible to pinpoint the exact location of a type error in the Scala code. Then, it would be interesting to integrate this enhanced error detection in an editor such as Emacs, effectively creating an interactive environment to construct signatures and proofs similar to the Twelf Emacs mode.

References

[1] The twelf project. http://twelf.org/wiki/Main_Page, 2014.

[2] John Boyland. John boyland's twelf library. https://github.com/boyland/twelf-library, 2012.

[3] Daniel Espino. Embedding logical frameworks in scala. https://github.com/akathorn/elfcala, 2014.

[4] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[5] Frank Pfenning. Logic programming in the lf logical framework. *Logical frameworks*, pages 149–181, 1991.

[6] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Automated Deduction—CADE-11*, pages 537–551. Springer, 1992.

[7] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. *Automated Deduction—CADE-16*, pages 202–206, 1999.

[8] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[9] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In *Programming Languages and Systems—ESOP'96*, pages 296–310. Springer, 1996.

[10] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for lf. In *Automated Deduction—CADE-15*, pages 286–300. Springer, 1998.