System Description: GAPT for schematic proofs

David M. Cerna[1] and Anela Lolic[2]*

[1] Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria
David.Cerna@risc.jku.at
[2] Institute of Discrete Mathematics and Geometry, TU Wien, Vienna, Austria
anela@logic.at

Abstract. In contrast to automated and interactive theorem provers, which focus on the construction of proofs, GAPT (General Architecture for Proof Theory) is a proof theory framework concentrating on the transformation and further processing of proofs. In this paper, we describe an extension of GAPT for handling schematic proofs, that is recursively defined LK-proofs.

1 Introduction

In this paper we describe an extension of the system GAPT (General Architecture for Proof Theory) for handling schematic proofs. GAPT is a proof theory framework consisting of data structures, algorithms, parsers, etc. which are useful for proof transformation, processing, and related theorem proving issues (in contrast to automated and interactive theorem provers which focus on the construction of proofs). A good reference summarizing the features of GAPT (*sans* the schematic components) is [6]. GAPT has served as a foundation for the implementation of a multitude of proof theoretic algorithms as well as an experimental base, i.e cut-elimination by resolution [1, 2], post-processing of resolution proofs, cut-introduction, expansion trees for proof import, and inductive theorem proving based on tree grammars. The graphical user interface *Prooftool* has been described in [6]. GAPT is implemented in Scala and licensed under the GNU General Public License and is available under `https://logic.at/gapt`.

Proof schemata serve as an alternative formulation of induction through primitive recursive proof specification. The seminal work concerning "proof as schema" was focused on proof analysis of Fürstenberg's proof of the infinitude of primes by Baaz et al. [1] using a rudimentary schematic formalism and CERES [2]. Schematic proof representation excels at proof analysis and transformation without "unrolling" the formal proof, thus making it particularly suited for analysis of inductive reasoning. For example, Herbrand's theorem can be extended to an expressive fragment (k-induction) of proof schemata [5, 7].

In this system description, we describe how to work with schematic proofs in GAPT, in particular how such proofs can be entered into the GAPT system

*This research is supported by the FWF projects P28789-N32 and I-2671-N35.

and which proof theoretic transformations can be applied. Our implementation is based on (parts of) the schematic CERES method (introduced in [5, 7]). Furthermore, the theoretic limitations of [7] concerning single parameter schema, mutual recursion, and induction over arbitrary inductive definitions have been dropped. While these changes entail that the soundness and completeness results discussed in [7] no longer hold, they provide the necessary flexibility needed for experimentation and future development of the formalism. Being that schematic proof transformation is heavily dependent on experimental and interactive analysis the development of a system as presented here is essential for the area.

2 Schematic Proofs

GAPT contains a simple tactics language called *gaptic* for the construction of proofs providing a comfortable bottom-up development of proofs similar to proof assistants such as Coq and Isabelle. Proofs can also be constructed using a top-down construction, however this would be particularly difficult for schematic proofs. In this section we provide an example construction of a schematic proof using the gaptic language. In particular we construct the FunctionIterationSchema, which can be found in the gapt/examples/schema/Schema directory of the current release. For a general overview of gaptic we refer to the GAPT User Manual[1]. Mathematically the FunctionIterationSchema is

$$P(a), \forall x(P(x) \to f(x)) \vdash \hat{\text{if}}(n, a) \tag{1}$$

where $\hat{\text{if}}(n, a)$ is a *defined function symbol* (see [7]) defined as follows:

$$\hat{\text{if}}(n+1, a) \equiv f(\hat{\text{if}}(n, a)) \qquad \hat{\text{if}}(0, a) \equiv a \tag{2}$$

and contains a schematic number of Π_1 cuts $\forall x(P(x) \to f(x))$. More complex examples can be found in the same directory, i.e. schema discussed in [4].

First of all note that schematic proofs are dependent on several core packages of GAPT.

```
import at.logic.gapt.expr._
import at.logic.gapt.proofs.Context._
import at.logic.gapt.proofs.gaptic._
import at.logic.gapt.proofs.Context
import at.logic.gapt.proofs.Sequent
```

These are the minimum imports necessary for schematic proof construction. For those not familiar with the GAPT system, the *context* stores all information needed for a proof's construction. Theoretically a schematic proof (or proof schema) is a ordered collection of pairs of proofs ending with the same end sequent referred to as *components* [4]. The schematic extension of GAPT generalizes this concept to proof tuples with the same end sequent allowing the use

[1] https://www.logic.at/gapt/downloads/gapt-user-manual.pdf

of inductive definitions other than the natural numbers. To enforce the shared end sequent condition the system allows for *proof name declarations* which each member of the component tuple must match. For the FunctionIterationSchema the following symbols and definitions must be added to the context within the proof object:

```
object FunctionIterationSchema extends TacticsProof {
// individual sort
  ctx += Context.Sort( "i" )
// numeric sort
  ctx += Context.InductiveType( "nat", hoc"0 : nat", hoc"s :
    nat>nat" )
//symbol definitions
  ctx += hoc"f:i>i"
  ctx += hoc"a:i"
  ctx += hoc"P: i>o"
//inductive symbol definition
  ctx += PrimRecFun( hoc"if:nat>i>i", "if 0 x = x", "if (s y)
    x = (f (if y x))" )
//Proof names
  ctx += hoc"phi: nat>nat"
//Proof definitions
  val esPhi = Sequent(Seq( hof"!x (-P(x) | P(f(x)))", hof"P(a)
    " ),Seq( hof"P(if(n,a))" ) )
  ctx += Context.ProofNameDeclaration( le"phi n", esPhi )
                                  . . .
  }
```

Note that the distinction between the inductive sort `Context.InductiveType( "nat", hoc"0:nat", hoc"s:nat>nat")` and the first-order term sort `Context.Sort("i")` must be made explicit, as well as the constants associated with the proof. Moreover, the definition of $\hat{\text{if}}(n+1, a)$ from (2) is added as a primitive recursive function `PrimRecFun( hoc"if:nat>i>i", "if 0 x = x", "if (s y) x = (f (if y x))" )`. Now we can construct the members of the proof tuple associated with the proof definition of *phi*:

```
  val esPhiSc = Sequent(
    Seq("Ant_1" -> hof"!x (-P(x) | P(f(x)))",
      "Ant_0" -> hof"P(a)" ),
    Seq( "Suc_0" -> hof"P(if(s(n),a))" ) )
  val phiSc = Lemma( esPhiSc ) {
    cut( "cut", hof"!x (-P(x) | P(f(x)))" )
    allR( "cut", fov"A" )
                  . . .
    ref( "phi" )
    unfold( "if" ) atMost 1 in "Suc_0"
    trivial
  }
```

```
  ctx += Context.ProofDefinitionDeclaration( le"phi (s n)",
    phiSc )
  val esPhiBc = Sequent(
    Seq("Ant_1" -> hof"!x (-P(x) | P(f(x)))",
      "Ant_0" -> hof"P(a)" ),
    Seq( "Suc_0" -> hof"P(if(0,a))" ) )
  val phiBc = Lemma( esPhiBc ) {
    unfold( "if" ) atMost 1 in "Suc_0"
    trivial
  }
  ctx += Context.ProofDefinitionDeclaration(le"phi 0", phiBc)
}
```

Note that the proof name has a function type of "nat > nat". While it acts as a place holding constant, for proper integration into the system it has been implemented as a lambda term and thus, takes arguments and has a return type.

Notice that the two members of the component tuple associated with phi are referred to as phiSc and phiBc. We add these proofs to the context using ProofDefinitionDeclaration. Taking a close look at the above code, one will notice that phiBc is added to the context with a lambda expression le"phi 0" and phiSc with le"phi (s n)"; these are the cases of the inductive definition handled by the proof. It is not necessary for the cases to match the inductive definition nor for all cases to be covered, but doing so can cause unexpected exceptions. An example of such a schema would be the NdiffSchema found in /gapt/examples/schema/NdiffSchema.scala. The last point we would like to address concerning proof construction as it relates to schema is the additional tactic added to the gaptic language, i.e. the reference tactic, and the extensive use of the unfold tactic. The reference tactic used in the phiSc, ref("phi"), allows one to reference a proof, including the proof itself (self reference). The system will check to see if the reference is valid, i.e. there exists a proof name matching the goal the reference was called on. The unfold tactic, while not particular to schema, is used for unfolding the primitive recursive definitions such as $\hat{\text{if}}(n+1, a)$.

The proofs phiBc and phiSc can be displayed in Prooftool (which is a viewer for proofs and other elements implemented in GAPT) using the command prooftool(FunctionInterationSchema.phiBc)(see Figure 1). Also, We can instantiate the proof schema of phi with an arbitrary value; instantiateProof.Instantiate(le"phi (s (s (s 0)))") instantiates the proof schema with the value 3. For more details, we refer the reader to a tutorial on schematic proof construction which can be found in the directory /gapt/examples/schema/Schema Tutorial.

3 Schematic Proof Analysis

Proof schemata, when instantiated, are essentially LK-proofs and thus, any of the proof analytic tools and methods of GAPT can be applied. However, there

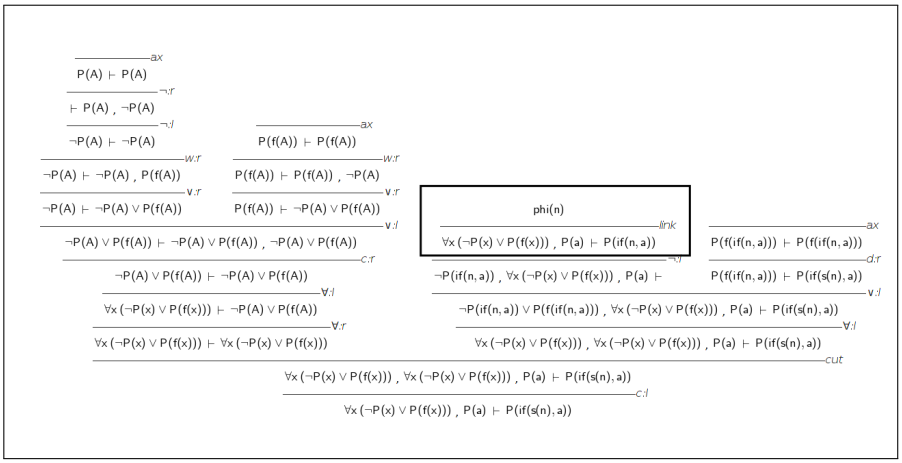

Fig. 1. Prooftool output of stepcase proof. Notice the link to the proof phi(n).

are also tools specifically designed for uninstantiated proof schemata, especially those which contain cuts, see [5, 7] for the theoretical details. For example, it is possible to extract schematic characteristic formula/clause sets, that is a recursive representation of the cut structure of the given proof schema. These objects are produced from an intermediate representation referred to as the *struct* of the proof. They are essential for the extraction of schematic Herbrand sequents (c.f. Herbrand systems [7]), and expansion proofs.

In the previous section we provided a short introduction to proof schema construction in GAPT by building the FunctionIterationSchema. Using this same example we provide a introduction to the schematic proof analysis capabilities of GAPT. This exposition can be found in the directory `/gapt/examples/schema/FunctionIterationRefutation`.

```
//imports the Context from the FunctionIterationSchema
object FunctionIterationRefutation extends TacticsProof(
    FunctionIterationSchema.ctx ) {
//Produces a schematic struct from the
//FunctionIterationSchema's Context
  val SCS: Map[CLS, ( Struct, Set[Var] )] = SchematicStruct( "
    phi" ).getOrElse( Map() )
//Produces a schematic characteristic formula
  val CFPRN = CharFormPRN( SCS )
//Constructs primitive recursive definitions of the
//characteristic formula
  CharFormPRN.PR( CFPRN )
                    . . .
}
```

Other than these few extra commands at the beginning of the object file, the construction of the refutation is quite similar to the construction of a proof schema. Essentially, one is proving $F \vdash$ where F is the schematic characteristic formula. All these commands do is add the names and normalizers for the characteristic formula to the context:

```
gapt> FunctionIterationSchema.ctx.names
res0: Iterable[String] = List(s, if, =, f, a, P, 0, phi, o,
    nat, i)
gapt> FunctionIterationRefutation.ctx.names
res1: Iterable[String] = List(phiSFAFF,phiSTATF,phiSFATF,s, if
    , =, f, a, P, 0, phi, o, nat, i)
```

The names associated with the characteristic formula phiSFAFF, phiSTATF and phiSFATF are automatically generated and are a combination of a proof name, phi, and a sequent of booleans, i.e. S(uccedent) F A(ntecedent) F F for phiSFAFF. The booleans mark cut ancestors. Once the schema of the characteristic formula is constructed an expansion proof for any instance can be produced (see Figure 2).

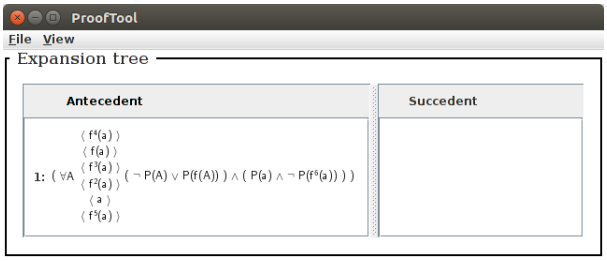

Fig. 2. Expansion Tree of instance five of the FunctionIterationRefutation schema.

4 Complex Example: Eventually Constant Schema

At IJCAR 2016, D. Cerna & A. Leitsch [3] presented a proof analysis of the so called *Eventually Constant Schema* using the method of [5] by manually producing instances of the proof and its cut structure using various theorem provers. This interactive process has been streamlined using the GAPT system and schemata of both the proof (EventuallyConstantSchema) and the characteristic formula (EventuallyConstantRefutation) can be found in gapt/examples/schema/Schema. This is a formal proof of the following statement:

Proposition 1. *A total monotonically decreasing function from* $\mathbb{N}$ *to a finite subset of* $\mathbb{N}$ *is eventually constant.*

The proof schema contains a sequence of Σ_2 cuts enforcing the function to be monotone decreasing. A nice property of this statement is that both its proof schema and refutation schema have the same recursive complexity [3] in terms of finite representation, making it a perfect example to illustrate the method. Though, the schema grows linearly and the refutation exponentially (Figure 3).

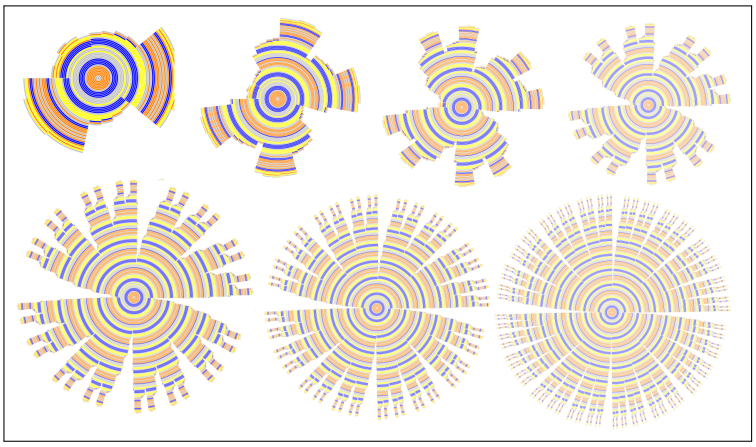

Fig. 3. Instances one to seven of the EventuallyConstantRefutation using the sunburst view of prooftool. Instance seven has 128 branches.

The components of the EventuallyConstantRefutation illustrate the exponential growth (Figure 4). This point was also addressed in [3].

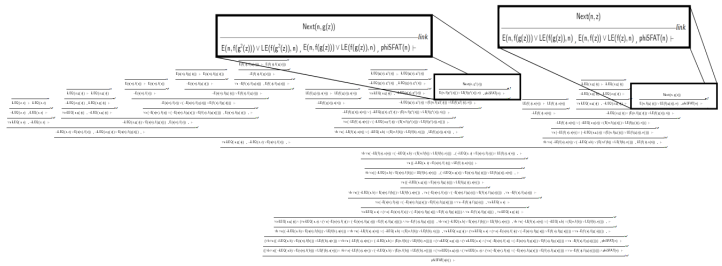

Fig. 4. The two inferences which are highlighted are two self references.

As for the FunctionIterationRefutation we can produce an expansion tree for the EventuallyConstantRefutation as well. Notice that it provides similar information as the substitutions discussed in [3] (see Figure 5).

$$
\begin{array}{c}
\langle z , z \rangle \\
\langle g^4(z) , g^3(z) \rangle \\
\langle g^2(z) , g(z) \rangle \\
\langle g^2(z) , g^2(z) \rangle \\
\langle g^4(z) , g^4(z) \rangle \\
\langle g(z) , g(z) \rangle \\
\langle g(z) , z \rangle \\
\langle g^3(z) , g^3(z) \rangle \\
\langle g^6(z) , g^6(z) \rangle \\
\langle g^5(z) , g^5(z) \rangle \\
\langle g^7(z) , g^6(z) \rangle \\
\langle g^3(z) , g^2(z) \rangle \\
\langle g^6(z) , g^5(z) \rangle \\
\langle g^5(z) , g^4(z) \rangle
\end{array}
$$

$$
(((\top \wedge \forall b \, \forall a \; (\neg \, \mathsf{LE}(f(a), s(0)) \vee (\neg \, \mathsf{iLEQ}(a, b) \vee (\mathsf{E}(0, f(b)) \vee \mathsf{LE}(f(b), 0))))) \wedge (\forall a
$$

$$
\begin{array}{c}
\langle g^5(z) \rangle \\
\langle g^6(z) \rangle \\
\langle z \rangle \\
\langle g^2(z) \rangle \, \mathsf{iLEQ}(a, g(a)) \\
\langle g^4(z) \rangle \\
\langle g(z) \rangle \\
\langle g^3(z) \rangle
\end{array}
\wedge (\forall a
\begin{array}{c}
\langle g^5(z) \rangle \\
\langle g^6(z) \rangle \\
\langle z \rangle \\
\langle g^2(z) \rangle \, \mathsf{iLEQ}(a, a) \, , \\
\langle g^4(z) \rangle \\
\langle g(z) \rangle \\
\langle g^3(z) \rangle
\end{array}
$$

Fig. 5. Part of the expansion tree of the ECS Schema refutation instance seven. Unlike the FunctionIterationRefutation, the ECS Schema has a very large expansion tree and thus we only present the most significant part.

5 Concluding Remarks

The features discussed in this system description can be used to work with recursively defined proofs in an interactive setting. The goal is to make the analysis carried out in [3] more streamlined in order to allow a similar analysis to be carried out on more complex proofs. Future work towards this goal will include the implementation of proof theoretic constructs such as proof projections as well as the integration of existing cyclic provers into the GAPT system.

References

1. Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. CERES: an analysis of fürstenberg's proof of the infinity of primes. *Theor. Comput. Sci.*, 403(2-3):160–175, 2008.
2. Matthias Baaz and Alexander Leitsch. Cut-elimination and redundancy-elimination by resolution. *J. Symb. Comput.*, 29(2):149–177, 2000.
3. David M. Cerna and Alexander Leitsch. Schematic cut elimination and the ordered pigeonhole principle. In *Automated Reasoning - 8th International Joint Conference, IJCAR, 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 241–256, 2016.
4. David M. Cerna and Michael Lettmann. Integrating a global induction mechanism into a sequent calculus. In *TABLEAUX'17*, Lecture Notes in Comp. Sci., Sept. 2017.
5. Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia, and Daniel Weller. Cut-elimination and proof schemata. In *Logic, Language, and Computation*, pages 117–136, 2013.
6. Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. System description: GAPT 2.0. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 293–301, 2016.
7. Alexander Leitsch, Nicolas Peltier, and Daniel Weller. Ceres for First-Order Schemata. *Journal of Logic and Computation*, 2017. https://doi.org/10.1093/logcom/exx003.