GAPT + Scala: Design Considerations and Examples

Tomer Líbal

Amadeus Project on Proof Compression

May 29th, 2012
Several different projects related to proof theory
All need implementations
Decision → design a general framework
Programming language - Scala
Many design decisions
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
Languages Support
Automated Prover
Generality

- Different languages: $\lambda$-calculus, First-order logic, etc.
- Different calculi: LK, Resolution, etc.
- General algorithms: Unification, Cut-elimination, etc.
- General applications: Theorem provers, user interfaces, etc.
Flexibility

- Time constraints → basic implementations
- Optimization of certain aspects later
- Several implementations for the same tasks
Community

- Developers with varying programming experience
- Skills in varying programming languages
- Varied theoretical background
Program

- Relatively bug-free
- Elegant and intuitive code
- Easy to read and well-documented code
- Algorithms at the frontier of research
- Rigorous and tested
Not among the aims

- Efficiency, although it should be supported if required
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
Quick overview

- Runs on Java Virtual Machine and is compatible with Java
- Support both Functional and OOP programming paradigms
- Although features set contains that of Java, it is more concise and elegant
- Strongly and statically typed
- Unlike Java, has type inference
The functional programming paradigm - some characteristics

- Computation is done via evaluation of mathematical functions
- No side-effects are allowed
- Immutable data-structures
- Higher-order functions
- Algebraic datatypes
The functional programming paradigm - benefits

- Reduced number of bugs
- Closer to (logic) specifications
- Fast development of algorithms
- Code is intuitive and many times self-explanatory
The OOP programming paradigm - some characteristics

- **Abstraction** - a concept or idea not associated with any specific instance
- **Encapsulation** - hiding the objects inner state, which can be accessed via methods
- **Polymorphism** - an instance may have several types (classes, traits, etc.)
- **Inheritence** - creation of subtypes and code reuse
The OOP programming paradigm - benefits

- Decoupling of implementation from specification (interfaces/traits)
- Complex (static) typing of objects using classes (traits) is possible
- **Modularity** - each element implementation is decoupled from other elements
- **Modifiability** and **Extensibility** - easy to change or extend the code
- **Re-usability** - elements can be easily re-used
- **Design patterns** - common coding habits are identified and optimized
Multi-paradigms and Scala benefits

- OOP → Functional - **Extractors** in order to expose the data as algebraic and allow pattern matching on it
- Functional → OOP - **Functions are objects**
- Strong and Static type system → more bugs are detected in compile time
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
Putting in together

GAPT

Scala

Generality
Generality

- Different languages: $\lambda$-calculus, First-order logic, etc.
- Different calculi: LK, Resolution, etc.
- General algorithms: Unification, Cut-elimination, etc.
- General applications: Theorem provers, user interfaces, etc.
Putting in together

<table>
<thead>
<tr>
<th>GAPT</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>OOP</td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
</tr>
</tbody>
</table>
Flexibility

- Time constraints → basic implementations
- Optimization of certain aspects later
- Several implementations for the same tasks
## Putting in together

<table>
<thead>
<tr>
<th>GAPT</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>OOP</td>
</tr>
<tr>
<td>Flexibility</td>
<td>OOP</td>
</tr>
<tr>
<td>Community</td>
<td></td>
</tr>
</tbody>
</table>
Community

- Developers with varying programming experience
- Skills in varying programming languages
- Varied theoretical background
## Putting in together

<table>
<thead>
<tr>
<th>GAPT</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>OOP</td>
</tr>
<tr>
<td>Flexibility</td>
<td>OOP</td>
</tr>
<tr>
<td>Community</td>
<td>OOP + Functional</td>
</tr>
<tr>
<td>Program</td>
<td></td>
</tr>
</tbody>
</table>
Program

- Relatively bug-free
- Elegant and intuitive code
- Easy to read and well-documented code
- Algorithms at the frontier of research
- Rigorous and tested
## Putting in together

<table>
<thead>
<tr>
<th>GAPT</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>OOP</td>
</tr>
<tr>
<td>Flexibility</td>
<td>OOP</td>
</tr>
<tr>
<td>Community</td>
<td>OOP+Functional</td>
</tr>
<tr>
<td>Program</td>
<td>Functional</td>
</tr>
</tbody>
</table>

- **In general:**
  - Data structures $\rightarrow$ OOP
  - Algorithms $\rightarrow$ Functional
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
Languages hierarchy

\[ \lambda \text{-calculus} \]
\[ \text{Higher-order logic} \]
\[ \text{First-order logic} \quad \text{Schema} \]
trait LambdaExpression extends LambdaFactoryProvider with Ordered[LambdaExpression] {
  def exptype: TA
  ...
}

class Var ( val name: SymbolA, val exptype: TA) extends LambdaExpression { ...}
trait HOLExpression extends LambdaExpression with HOL

case object AndC extends
  Var(AndSymbol, "(o \rightarrow (o \rightarrow o))") with
  Const with HOLExpression

- We distinguish higher-order variables and constants by having distinct symbols sets
- Const \rightarrow OOP: Abstraction and Polymorphism \rightarrow extra static typing
Factories

- The **Abstract factory** design pattern

The **abstract factory pattern** is a software **design pattern** that provides a way to encapsulate a group of individual **factories** that have a common theme. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic **interfaces** to create the concrete **objects** that are part of the theme. The **client** does not know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage.
Each language DS has as a member and is created from a factory
object LambdaFactory extends LambdaFactoryA {
    def createVar( .. ) = new Var( .. )
    def createAbs( .. ) : Abs = new Abs( .. )
    def createApp( .. ) = new App( f .. )
}

- The factory gives to each created element a reference to itself, for future creation of objects.
object HOLFactory extends LambdaFactoryA {
    def createVar( name: SymbolA, .. ) : Var =
        name match {
            case a: ConstantSymbolA => ..
            case a: VariableSymbolA => ..
        }
    def createApp( .. ) : App =
        ..
}
object Var {
    def unapply(expression: LambdaExpression) =
        expression match {
            case a: Var => Some((a.name, a.exptype))
            case _ => None
        }
}

object App {
    def unapply(expression: LambdaExpression) =
        expression match {
            case a: App => Some((a.function, a.argument))
            case _ => None
        }
}

Write functional-style code

```python
def betaReduce(expression: LambdaExpression):
    LambdaExpression => expression match {
        case App(abs: Abs, arg) => ..
        case Var( .. ) => ..
        case Abs( .. ) => ..
        case App( .. ) => ..
    }
```

- Beta-reduce can also be applied to HOL objects, etc. and will return HOL objects
Outline

GAPT - Aims

Scala

GAPT + Scala

From Theory to Practice
  Languages Support
  Automated Prover
ATP - aims

- Can be applied to any calculus over any language
- Can be interactive, automated or semi-automated
- Can be easily extended and modified
- Is not optimized or particularly efficient
trait Configuration[S] {
  def result: Option[S]
}

abstract class NDStream[S /*result type*/](
  val initial: Configuration[S],
  val myFun: Configuration[S] => Iterable[Configuration[S]]
) extends SearchAlgorithm {
  val results: Queue[S]
  def next: Option[S]
  ...
}
trait Prover {
    def refute(commands: Stream[Command]): NDStream[ResolutionProof] = {
        new NDStream(new MyConfiguration(
            new State(), commands, ()
        ), myFun) with BFSAlgorithm
    }

    def myFun(c: Configuration[ResolutionProof]): Iterable[Configuration[ResolutionProof]] = {
        ...
        conf.commands.head match {
            case com: InitialCommand => ...
            case com: DataCommand =>
                com(conf.state, conf.data).map(...)
            case com: ResultCommand => ...
        }
    }
}
The commands

- Commands are functions from (State, Data) to Iterable[(State, Data)]
- Data is specific to each command
- For example:

```scala
case class ResolveCommand(
  alg: UnificationAlgorithm
) extends DataCommand[Clause] {
  def apply(state: State, data: Any) = {
    // data is a pair of clauses
    // result is all resolvants,
    // one for each unifier
  }
}
```
An example of a simple interactive prover

def stream1: Stream[Command] = Stream.cons(getTwoClausesFromUICommand(PromptTerminal.GetTwoClauses),
    Stream.cons(VariantsCommand,
        Stream.cons(DeterministicAndCommand((
            List(ApplyOnAllPolarizedLiteralPairsCommand, ResolveCommand(FOLUnificationAlgorithm), FactorCommand(FOLUnificationAlgorithm)),
            List(ParamodulationCommand(FOLUnificationAlgorithm)))),
        Stream.cons(SimpleForwardSubsumptionCommand(new StillmanSubsumptionAlgorithm {val matchAlg = FOLMatchingAlgorithm})),
        Stream.cons(SimpleBackwardSubsumptionCommand(new StillmanSubsumptionAlgorithm {val matchAlg = FOLMatchingAlgorithm})),
        Stream.cons(InsertResolventCommand, Stream.cons(RefutationReachedCommand, stream1))))

def stream: Stream[Command] = Stream.cons(SetTargetClause(EmptyClause, Stream.cons(SearchForEmptyClauseCommand, stream1)))
Replaying

- Since there is no focus on efficiency, third-party theorem provers can be used.
- Given an inference, the prover tries to infer the conclusion from the premises via forward resolution.
- The prover is initialized with a stream of finitely many groups (for each inference) of infinitely many commands.
- Since the third-party provers used are sound, the runs over each group of commands terminate.
- The obtained conclusion is added to the global clauses set.
- The final refutation contains only Robinson’s calculus rules and para-modulation and includes all applied substitutions.
- Forward resolution poses more difficulties than refutational resolution and it is still an on-going work to solve all of them.
Thank you.