The Lean 4 Theorem Prover and Programming Language

Leonardo de Moura[1](✉) and Sebastian Ullrich[2]

[1] Microsoft Research, Redmond WA, USA
leonardo@microsoft.com
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
sebastian.ullrich@kit.edu

Abstract. Lean 4 is a reimplementation of the Lean interactive theorem prover (ITP) in Lean itself. It addresses many shortcomings of the previous versions and contains many new features. Lean 4 is fully extensible: users can modify and extend the parser, elaborator, tactics, decision procedures, pretty printer, and code generator. The new system has a hygienic macro system custom-built for ITPs. It contains a new typeclass resolution procedure based on tabled resolution, addressing significant performance problems reported by the growing user base. Lean 4 is also an efficient functional programming language based on a novel programming paradigm called *functional but in-place*. Efficient code generation is crucial for Lean users because many write custom proof automation procedures in Lean itself.

1 Introduction

The Lean project[3] started in 2013 [9] as an interactive theorem prover based on the Calculus of Inductive Constructions [4] (CIC). In 2017, using Lean 3, a community of users with very different backgrounds started the Lean mathematical library project mathlib [13]. At the time of this writing, mathlib has roughly half a million lines of code, and contains many nontrivial mathematical objects such as Schemes [2]. Mathlib is also the foundation for the *Perfectoid Spaces in Lean* project [1], and the *Liquid Tensor* challenge [11] posed by the renowned mathematician Peter Scholze. Mathlib contains not only mathematical objects but also Lean metaprograms that extend the system [5]. Some of these metaprograms implement nontrivial proof automation, such as a ring theory solver and a decision procedure for Presburger arithmetic. Lean metaprograms in mathlib also extend the system by adding new top-level command and features not related to proof automation. For example, it contains a package of semantic linters that alert users to many commonly made mistakes [5]. Lean 3 metaprograms have

[3] http://leanprover.github.io

© The Author(s) 2021
A. Platzer and G. Sutcliffe (Eds.): CADE 2021, LNAI 12699, pp. 625–635, 2021.
https://doi.org/10.1007/978-3-030-79876-5_37

been also instrumental in building standalone applications, such as a SQL query equivalence checker [3].

We believe the Lean 3 theorem prover's success is primarily due to its extensibility capabilities and metaprogramming framework [6]. However, users cannot modify many parts of the system without changing Lean 3 source code written in C++. Another issue is that many proof automation metaprograms are not competitive with similar proof automation implemented in programming languages with an efficient compiler such as C++ and OCaml. The primary source of inefficiency in Lean 3 metaprograms is the virtual machine interpretation overhead.

Lean 4 is a reimplementation of the Lean theorem prover in Lean itself[4]. It is an extensible theorem prover and an efficient programming language. The new compiler produces C code, and users can now implement efficient proof automation in Lean, compile it into efficient C code, and load it as a plugin. In Lean 4, users can access all internal data structures used to implement Lean by merely importing the `Lean` package. Lean 4 is also a platform for developing efficient domain-specific automation. It has a more robust and extensible elaborator, and addresses many other shortcomings of Lean 3. We expect the Lean community to extend and add new features without having to change the Lean source code. We released Lean 4 at the beginning of 2021, it is open source, the community is already porting mathlib, and the number of applications is quickly growing. It includes a translation verifier for Reopt[5], a package for supporting inductive-inductive types[6], and a car controller[7].

2 Lean by Example

In this section, we introduce the Lean language using a series of examples. The source code for the examples is available at `https://github.com/leanprover/lean4/blob/cade2021/doc/BoolExpr.lean`. For additional details and installation instructions, we recommend the reader consult the online manual[8].

We define functions by using the `def` keyword followed by its name, a parameter list, return type, and body. The parameter list consists of successive parameters that are separated by spaces. We can specify an explicit type for each parameter. If we do not specify a specific argument type, the elaborator tries to infer the function body's type. The Boolean `or` function is defined by pattern-matching as follows

```
def or (a b : Bool) :=
  match a with
  | true  => true
  | false => b
```

[4] http://github.com/leanprover/lean4
[5] https://github.com/GaloisInc/reopt-vcg
[6] https://github.com/javra/iit
[7] https://github.com/GaloisInc/lean4-balance-car
[8] http://leanprover.github.io/lean4/doc

We can use the command `#check <term>` to inspect the type of term, and `#eval <term>` to evaluate it.

```
#check or true false -- Bool (this is a comment in Lean)
#eval  or true false -- true
```

Lean has a hygienic macro system and comes equipped with many macros for commonly used idioms. For example, we can also define the function `or` using

```
def or : Bool → Bool → Bool
 | true, _  => true
 | false, b => b
```

The notation above is a macro that expands into a `match`-expression. In Lean, a theorem is a definition whose result type is a proposition. For an example, consider the following simple theorem about the definition above

```
theorem or_true (b : Bool) : or true b = true :=
  rfl
```

The constant `rfl` has type `∀ {α : Sort u} {a : α}, a = a`, the curly braces indicate that the parameters `α` and `a` are *implicit* and should be inferred by solving typing constraints. In the example above, the inferred values for `α` and `a` are `Bool` and `or true b`, respectively, and the resulting type is `or true b = or true b`. This is a valid proof because `or true b` is *definitionally equal* to `b`. In dependent type theory, every term has a computational behavior, and supports a notion of reduction. In principle, two terms that reduce to the same value are called definitionally equal. In the following example, we use pattern matching to prove that `or b b = b`

```
theorem or_self : ∀ (b : Bool), or b b = b
  | true  => rfl
  | false => rfl
```

Note that `or b b` does not reduce to `b`, but after pattern matching we have that `or true true` (`or false false`) reduces to `true` (`false`).

In the following example, we define the recursive datatype `BoolExpr` for representing Boolean expressions using the command `inductive`.

```
inductive BoolExpr where
  | var (name : String)
  | val (b : Bool)
  | or  (p q : BoolExpr)
  | not (p : BoolExpr)
```

This command generates constructors `BoolExpr.var`, `BoolExpr.val`, `BoolExpr.or`, and `BoolExpr.not`. The Lean kernel also generates an inductive principle for the new type `BoolExpr`. We can write a basic "simplifier" for Boolean expressions as follows

```
def simplify : BoolExpr → BoolExpr
  | BoolExpr.or p q => mkOr (simplify p) (simplify q)
  | BoolExpr.not p  => mkNot (simplify p)
```

```
    | e                => e
  where
    mkOr : BoolExpr → BoolExpr → BoolExpr
      | p, BoolExpr.val true   => BoolExpr.val true
      | p, BoolExpr.val false  => p
      | BoolExpr.val true, p   => BoolExpr.val true
      | BoolExpr.val false, p  => p
      | p, q                   => BoolExpr.or p q

    mkNot : BoolExpr → BoolExpr
      | BoolExpr.val b => BoolExpr.val (not b)
      | p              => BoolExpr.not p
```

The function `simplify` is a simple bottom-up simplifier. We use the `where` clause to define two local auxiliary functions `mkOr` and `mkNot` for constructing "simplified" `or` and `not` expressions respectively. Their global names are `simplify.mkOr` and `simplify.mkNot`.

Given a context that maps variable names to Boolean values, we define a "denotation" function (or evaluator) for Boolean expressions. We use an association list to represent the context.

```
  abbrev Context := AssocList String Bool

  def denote (ctx : Context) : BoolExpr → Bool
    | BoolExpr.or p q => denote ctx p || denote ctx q
    | BoolExpr.not p  => !denote ctx p
    | BoolExpr.val b => b
    | BoolExpr.var x => if let some b := ctx.find? x then b else false
```

In the example above, `p || q` is notation for `or p q`, `!p` for `not p`, and `if let p := t then a else b` is a macro that expands into `match t with | p => a | _ => b`. The term `ctx.find? x` is syntax sugar for `AssocList.find? ctx x`.

As in previous versions, we can use tactics for constructing proofs and terms. We use the keyword `by` to switch into *tactic mode*. Tactics are user-defined or built-in procedures that construct various terms. They are all implemented in Lean itself. The `simp` tactic implements an extensible simplifier, and is one of the most popular tactics in mathlib. Its implementation [9] can be extended and modified by Lean users.

```
  ...
  @[simp] theorem denote_mkOr (ctx : Context) (p q : BoolExpr)
          : denote ctx (simplify.mkOr p q) = denote ctx (or p q) :=
    ...
  def denote_simplify (ctx : Context) (p : BoolExpr)
      : denote ctx (simplify p) = denote ctx p :=
    by induction p with
    | or p q ih₁ ih₂ => simp [ih₁, ih₂]
```

[9] https://github.com/leanprover/lean4/blob/cade21/src/Lean/Meta/Tactic/Simp/Main.lean.

```
  | not p ih        => simp [ih]
  | _               => rfl
```

In the example above, we use the `induction` tactic, its syntax is similar to a `match`-expression. The variables $\texttt{ih}_1$ and $\texttt{ih}_2$ are the induction hypothesis for `p` and `q` in the first alternative for the case `p` is a `BoolExpr.or`. The `simp` tactic uses any theorem marked with the `@[simp]` attribute as a rewriting rule (e.g., `denote_mkOr`). We explicitly provide the induction hypotheses as additional rewriting rules inside square brackets.

Typeclass Resolution. Typeclasses [16] provide an elegant and effective way of managing ad-hoc polymorphism in both programming languages and interactive proof assistants. Then we can declare particular elements of a typeclass to be *instances*. These provide hints to the elaborator: any time the elaborator is looking for an element of a typeclass, it can consult a table of declared instances to find a suitable element. What makes typeclass inference powerful is that one can *chain* instances, that is, an instance declaration can in turn depend on other instances. This causes class inference to recurse through instances, backtracking when necessary. The Lean typeclass resolution procedure can be viewed as a simple λ-Prolog interpreter [8], where the Horn clauses are the user declared instances.

For example, the standard library defines a typeclass `Inhabited` to enable typeclass inference to infer a "default" or "arbitrary" element of types that contain at least one element.

```
class Inhabited (α : Sort u) where
  default : α

def arbitrary [Inhabited α] : α :=
  Inhabited.default
```

The annotation `[Inhabited α]` at `arbitrary` indicates that this implicit parameter should be synthesized from instance declarations using typeclass resolution. We can define an instance for our `BoolExpr` type defined earlier as follows

```
instance : Inhabited BoolExpr where
  default := BoolExpr.val false
```

This instance specifies that the "default" element for `BoolExpr` is `BoolExpr.val false`. The following declaration shows that if two types α and β are inhabited, then so is their product:

```
instance [Inhabited α] [Inhabited β] : Inhabited (α × β) where
  default := (arbitrary, arbitrary)
```

The standard library has many builtin classes such as `Repr α` and `DecidableEq α`. The class `Repr α` is similar to Haskell's `Show α` typeclass, and `DecidableEq α` is a typeclass for types that have decidable equality. Lean 4 also provides code synthesizers for many builtin classes. The command `deriving` instructs Lean to auto-generate an instance.

```
deriving instance DecidableEq for BoolExpr

#eval decide (BoolExpr.val true = BoolExpr.val false) -- false
```

In the example above, the `deriving` command generates the instance

```
(a b : BoolExpr) → Decidable (a = b)
```

The function `decide` evaluates decidable propositions. Thus, the last command returns `false` since `BoolExpr.val true` is not equal to `BoolExpr.val false`.

The increasingly sophisticated uses of typeclasses in mathlib have exposed a few limitations in Lean 3: unnecessary overhead due to the lack of term indexing techniques, and exponential running times in the presence of diamonds. Lean 4 implements a new procedure [12], tabled typeclass resolution, that solves these problems by using discrimination trees[10]. for better indexing and tabling, which is a generalization of memoizing introduced initially to address similar limitations of early logic programming systems[11].

The hygienic macro system. In interactive theorem provers (ITPs), Lean included, extensible syntax is not only crucial to lower the cognitive burden of manipulating complex mathematical objects, but plays a critical role in developing reusable abstractions in libraries. Lean 3 support such extensions in the form of restrictive "syntax sugar" substitutions and other ad hoc mechanisms, which are too rudimentary to support many desirable abstractions. As a result, libraries are littered with unnecessary redundancy. The Lean 3 tactic languages is plagued by a seemingly unrelated issue: accidental name capture, which often produces unexpected and counterintuitive behavior. Lean 4 takes ideas from the Scheme family of programming languages and solves these two problems simultaneously by use of a *hygienic*, i.e. capture-avoiding, macro system custom-built for ITPs [15].

Lean 3's "mixfix" notation system is still supported in Lean 4, but based on the much more general macro system; in fact, the Lean 3 `notation` keyword itself has been reimplemented as a macro, more specifically as a *macro-generating macro*. By providing such a tower of abstractions for writing syntax sugars, of which we will see more levels below, we want to enable users to work in the simplest model appropriate for their respective use case while always keeping open the option to switch to a lower, more expressive level.

As an example, we define the infix notation $\Gamma \vdash$ `p`, with precedence 50, for the function `denote` defined earlier.

```
infix:50 "⊢" => denote
```

The `infix` command expands to

```
notation:50 Γ "⊢" p:50 => denote Γ p
```

[10] `https://github.com/leanprover/lean4/blob/cade21/src/Lean/Meta/DiscrTree.lean`.

[11] `https://github.com/leanprover/lean4/blob/cade21/src/Lean/Meta/SynthInstance.lean`.

which itself expands to the macro declaration

```
macro:50 Γ:term "⊢" p:term:50  : term => `(denote $Γ $p)
```

where the *syntactic category* (`term`) of placeholders and of the entire macro is now specified explicitly, implying that macros can also be written for/using other categories such as the top-level `command`. The right-hand side uses an explicit *syntax quasiquotation* to construct the syntax tree, with syntax placeholders (*antiquotations*) prefixed with `$`. As suggested by the explicit use of quotations, the right-hand side may now be an arbitrary Lean term computing a syntax object, allowing for *procedural* macros as well.

`macro` itself is another command-level macro that, for our `notation` example, expands to two commands

```
syntax:50 term "⊢" term:50 : term
macro_rules
  | `($Γ ⊢ $e) => `(denote $Γ $e)
```

that is, a pair of parser extension and syntax transformer. By separating these two steps at this abstraction level, it becomes possible to define (mutually) recursive macros and to reuse syntax between macros. Using `macro_rules`, users can even extend existing macros with new rules. In general, separating parsing and expansion means that that we can obtain a well-structured syntax tree pre-expansion, i.e. a *concrete* syntax tree, and use it to implement source code tooling such as auto-completion, go-to-definition, and refactorings.

We can use the `syntax` command for defining embedded domain-specific languages. In simple cases, we can reuse existing syntactic categories for this but assign them new semantics, such as in the following notation for constructing `BoolExpr` objects.

```
syntax "`[BExpr|" term "]" : term
macro_rules
  | `(`[BExpr| true])     => `(BoolExpr.val true)
  | `(`[BExpr| false])    => `(BoolExpr.val false)
  | `(`[BExpr| $x:ident]) => `(BoolExpr.var $(quote x.getId.toString))
  | `(`[BExpr| $p ∨ $q])  => `(BoolExpr.or `[BExpr| $p] `[BExpr| $q])
  | `(`[BExpr| ¬ $p])     => `(BoolExpr.not `[BExpr| $p])

#check `[BExpr| p ∨ true]
-- BoolExpr.or (BoolExpr.var "p") (BoolExpr.val true) : BoolExpr
```

The `macro_rules` command above specifies how to convert a subset of the builtin syntax for terms into constructor applications for `BoolExpr`. The term `$(quote x.getId.toString)` converts the identifier `x` into a string literal.

As a final example, we modify the notation `Γ ⊢ p`. In the following version, `Γ` is not an arbitrary term anymore, but a comma-separated sequence of entries of the form `var ↦ value`, and the right-hand side is now interpreted as a `BoolExpr` term by reusing our macro from above.

```
syntax entry := ident " ↦ " term:max
syntax entry,* "⊢" term : term
```

```
macro_rules
  | `( $[$xs:ident ↦ $vs:term],* ⊢ $p:term ) =>
    let xs := xs.map fun x => quote x.getId.toString
    `(denote (List.toAssocList [$[( $xs , $vs )],*]) `[BExpr| $p])
#eval a ↦ false, b ↦ true ⊢ b ∨ a  -- true
```

We use the *antiquotation splice* `$[$xs:ident ↦ $vs:term],*` to deconstruct the sequence of entries into two arrays `xs` and `vs` containing the variable names and values, respectively, adjust the former array, and combine them again in a second splice.

3 The Code Generator

The Lean 4 code generator produces efficient C code. It is useful for building both efficient Lean extensions and standalone applications. The code generator performs many transformations, and many of them are based on techniques used in the Haskell compiler GHC [7]. However, in contrast to Haskell, Lean is a strict language. We control code inlining and specialization using the attributes `@[inline]` and `@[specialize]`. They are crucial for eliminating the overhead introduced by the towers of abstractions used in our source code. Before emitting C code, we erase proof terms and convert Lean expressions into an intermediate representation (IR). The IR is a collection of Lean data structures,[12] and users can implement support for backends other than C by writing Lean programs that import `Lean.Compiler.IR`. Lean 4 also comes with an interpreter for the IR, which allows for rapid incremental development and testing right from inside the editor. Whenever the interpreter calls a function for which native, ahead-of-time compiled code is available, it will switch to that instead, which includes all functions from the standard library. Thus the interpretation overhead is negligible as long as e.g. all expensive tactics are precompiled.

Functional but in-place. Most functional languages rely on garbage collection for automatic memory management. They usually eschew reference counting in favor of a tracing garbage collector, which has less bookkeeping overhead at runtime. On the other hand, having an exact reference count of each value enables optimizations such as destructive updates [14]. When performing functional updates, objects often die just before creating an object of the same kind. We observe a similar phenomenon when we insert a new element into a purely functional data structure, such as binary trees, a theorem prover rewrites formulas, a compiler applies optimizations by transforming abstract syntax trees, or the function `simplify` defined earlier. We call it the *resurrection hypothesis*: many objects die just before creating an object of the same kind. The Lean memory manager uses reference counting and takes advantage of this hypothesis, and enables pure code to perform destructive updates in all scenarios described

[12] `https://github.com/leanprover/lean4/blob/cade21/src/Lean/Compiler/IR/Basic.lean`

above when objects are not shared. It also allows a novel programming paradigm that we call *functional but in-place* (FBIP) [10]. Our preliminary experimental results demonstrate our new compiler produces competitive code that often outperforms the code generated by high-performance compilers such as ocamlopt and GHC [14]. As an example, consider the function `map f as` that applies a function `f` to each element of a list `as`. In this example, `[]` denotes the empty list, and `a::as` the list with head `a` followed by the tail `as`.

```
def map : (α → β) → List α → List β
  | f, []    => []
  | f, a::as => f a :: map f as
```

If the list referenced by `as` is not shared, the code generated by our compiler does not allocate any memory. Moreover, if `as` is a nonshared list of list of integers, then `map (map inc) as` will not allocate any memory either. In contrast to static linearity systems, allocations are also avoided even if only a prefix of the list is not shared. FBIP also allows Lean users to use data structures, such as arrays and hashtables, in pure code without any performance penalty when they are not shared. We believe this is an attractive feature because hashtables are frequently used to implement decision procedures and nontrivial proof automation.

4 The User Interface

Our system implements the Language Server Protocol (LSP) using the task abstraction provided by its standard library. The Lean 4 LSP server is incremental and is continuously analyzing the source text and providing semantic information to editors implementing LSP. Our LSP server implements most LSP features found in advanced IDEs, such as hyperlinks, syntax highlighting, type information, error handling, auto-completion, etc. Many editors implement LSP, but VS Code is the preferred editor by the Lean user community. We provide extensions for visualizing the intermediate proof states in interactive tactic blocks, and we want to port the Lean 3 widget library for constructing interactive visualizations for their proofs and programs.

5 Conclusion

Lean 4 aims to be a fully extensible interactive theorem prover and functional programming language. It has an expressive logical foundation for writing mathematical specifications and proofs and formally verified programs. Lean 4 provides many new unique features, including a hygienic macro-system, an efficient typeclass resolution procedure based on tabled resolution, efficient code generator, and abstractions for sealing low-level optimizations. The new elaboration procedure is more general and efficient than those implemented in previous versions. Users may also extend and modify the elaborator using Lean itself. Lean has a relatively small trusted kernel, and the rich API allows users to export their developments to other systems and implement their own reference checkers. Lean

is an ongoing and long-term effort, and future plans include integration with external SMT solvers and first-order theorem provers, new compiler backends, and porting the Lean 3 Mathematical Library.

Acknowledgments. We are grateful to Marc Huisinga and Wojciech Nawrocki for developing the LSP server, Daniel Selsam for working with us on the new typeclass resolution procedure and interesting design discussions, Daan Leijen, Nikhil Swamy, Sebastian Graf, Simon Peyton Jones, and Max Wagner for advice and design discussions, Joe Hendrix, Andrew Kent, Rob Dockins, and Simon Winwood from Galois Inc for being early Lean 4 adopters, and providing useful feedback and suggestions, and the whole Lean community for all their excitement and pushing Lean forward.

References

1. Buzzard, K., Commelin, J., Massot, P.: Formalising Perfectoid Spaces. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 299–312. CPP 2020, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373830, `https://doi.org/10.1145/3372885.3373830`
2. Buzzard, K., Hughes, C., Lau, K., Livingston, A., Mir, R.F., Morrison, S.: Schemes in Lean. `https://arxiv.org/abs/2101.02602` (2021), arXiv:2101.02602
3. Chu, S., Murphy, B., Roesch, J., Cheung, A., Suciu, D.: Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. Proc. VLDB Endow. **11**(11), 1482–1495 (Jul 2018). https://doi.org/10.14778/3236187.3236200, `https://doi.org/10.14778/3236187.3236200`
4. Coquand, T., Huet, G.: The calculus of constructions. Inform. and Comput. **76**(2-3), 95–120 (1988)
5. van Doorn, F., Ebner, G., Lewis, R.Y.: Maintaining a library of formal mathematics. In: Benzmüller, C., Miller, B. (eds.) Intelligent Computer Mathematics. pp. 251–267. Springer International Publishing, Cham (2020)
6. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP) (Sep 2017). https://doi.org/http://dx.doi.org/10.1145/3110278
7. Jones, S.L.P.: Compiling Haskell by program transformation: a report from the trenches. In: In Proc. European Symp. on Programming. pp. 18–44. Springer-Verlag (1996)
8. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge (2012)
9. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The Lean theorem prover. In: International Conference on Automated Deduction. pp. 378–388. Springer (2015)
10. Reinking, A., Xie, N., de Moura, L., Leijen, D.: Perceus: Garbage free reference counting with reuse. Tech. Rep. MSR-TR-2020-42, Microsoft Research (2020)
11. Scholze, P.: Liquid tensor experiment. `https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment` (2020), project repository `https://github.com/leanprover-community/lean-liquid`
12. Selsam, D., Ullrich, S., de Moura, L.: Tabled typeclass resolution. `https://arxiv.org/abs/2001.04301` (2020), arXiv:2001.04301

13. The mathlib Community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 367–381. CPP 2020, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373824, https://doi.org/10.1145/3372885.3373824
14. Ullrich, S., de Moura, L.: Counting immutable beans: Reference counting optimized for purely functional programming. In: 31st Symposium on Implementation and Application of Functional Languages (2019)
15. Ullrich, S., de Moura, L.: Beyond notations: Hygienic macro expansion for theorem proving languages. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. pp. 167–182. Cham (2020)
16. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76. ACM (1989)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.