Modulaire zoekheuristieken in Scala

Benoit Desouter

Promotor: prof. Tom Schrijvers
Begeleider: Nicolas Van Cleemput

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Toegepaste wiskunde en Informatica
Voorzitter: prof. dr. Willy Govaerts
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2011-2012

UNIVERSITEIT
GENT

Voorwoord

Veel praktische problemen zijn van combinatorische aard. Denken we maar aan het opmaken van een productieplanning, een lesrooster of een dienstregeling. Dergelijke problemen kunnen theoretisch gezien altijd opgelost worden met een brute force strategie. In de praktijk moeten we haast altijd onze toevlucht nemen tot heuristieken. Constraint programming heeft daarenboven technieken ontwikkeld om de zoekruimte zoveel mogelijk te beperken [25]. Toch zijn zoekheuristieken nog steeds essentieel voor het succesvol oplossen van een constraint satisfaction problem.

Bestaande tools voor constraint programming bieden slechts beperkte mogelijkheden voor het formuleren van eigen zoekheuristieken. Modulaire zoekheuristieken zijn speciaal ontworpen om dit arbeidsintensieve werk te verlichten. Ze bieden ons een verzameling primitieve heuristieken die als het ware als Legoblokjes kunnen gecombineerd worden tot een grotere strategie.

De bestaande implementatie[1] in C++ kent een aantal problemen die de uitbreidbaarheid, onderhoudbaarheid en algemene toepasbaarheid beperken. Door een nieuwe implementatie te creëren in Scala hopen we hieraan tegemoet te kunnen komen. De keuze voor de relatief nieuwe programmeertaal Scala is hierbij geïnspireerd door de geavanceerde overervingsvormen die deze taal aanbiedt, zijn ondersteuning voor ingebedde domeinspecifieke talen en zijn hoge abstractieniveau.

Dit werk is er niet alleen gekomen door mijn interesse, er zijn nog tal van mensen die, op hun manier, een steentje bijgedragen hebben om mij dit werk te helpen kneden in zijn uiteindelijke vorm. Ik zou dan ook graag in de eerste plaats mijn thesisbegeleiders willen bedanken voor de grote hoeveelheid tijd en energie die ze in mij en dit werk stopten. Tevens wens ik mijn promotor te bedanken voor het vertrouwen en de geboden kans.

Daarnaast wil ik mijn ouders speciaal bedanken. Zij hebben mij in de loop der jaren alle kansen gegeven die ik maar wou en mij ten volle gesteund in mijn keuzes. Zij hebben mij de mogelijkheden gegeven om te staan waar ik nu sta en om mij te verdiepen in mijn interesses. Tot slot ook nog een dankjewel voor de zwemmers van Reddersclub Kortrijk met wie ik een aangename vrije tijd kon doorbrengen wanneer ik daar aan toe was.

Benoit Desouter, juni 2012

[1] Er bestaat ook een implementatie in Haskell die opgevat is als een codegenerator.

Toelating tot bruikleen

„De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik.
Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef."

"The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use.
In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation."

Benoit Desouter, juni 2012

Modulaire zoekheuristieken in Scala

door

Benoit Desouter

Masterproef ingediend tot het behalen van de academische graad van

Master in de ingenieurswetenschappen: computerwetenschappen

Academiejaar 2011–2012

Promotor: prof. Tom Schrijvers

Begeleider: Nicolas Van Cleemput

Faculteit Ingenieurswetenschappen en Architectuur

Universiteit Gent

Vakgroep Toegepaste wiskunde en Informatica

Voorzitter: prof. dr. Willy Govaerts

Samenvatting

In dit werk bespreken we de implementatie van modulaire zoekheuristieken in Scala. Modulaire zoekheuristieken komen tegemoet aan de moeilijkheid om nieuwe heuristieken te implementeren in hedendaagse constraint solvers. Een systeem van eenvoudig te combineren bouwblokken, de primitieve heuristieken, ligt hiervan aan de basis.

Scala is hiervoor een uitermate geschikte programmeertaal omdat zijn geavanceerd overervingsmechanisme toelaat codeduplicatie tussen de bouwstenen of combinators te vermijden. Daarnaast gebruiken we zelfgedefinieerde controleabstracties om patronen binnen één combinator te elimineren. Zo moet bij zes op de 21 geïmplementeerde basiscombinatoren nog maar één enkele methode geïmplementeerd worden om een volwaardige combinator te bekomen. Gemiddeld gaat het om 2,13 methodes, terwijl in een bestaande C++ implementatie zeven methodes vereist zijn. Een dergelijk resultaat is niet mogelijk met standaard enkelvoudige overerving, terwijl we toch de problemen die met meervoudige overerving gepaard gaan weten te vermijden. Bovendien laat het overervingsmechanisme toe de bouwstenen te combineren op een statisch getypeerde manier, waar eerder expliciet mixin inheritance nodig was.

Enerzijds is dus de drempel voor het toevoegen van een nieuwe primitieve heuristiek veel lager: in vergelijking met een eerdere implementatie in C++ is er minder code nodig; de resterende code is van een hoog niveau en dus makkelijker te onderhouden . Het design vereist geen casts in de code en is constraint-solveronafhankelijk. Hier hebben we de koppeling voorzien met de constraint solver JaCoP. Anderzijds is ook het samenstellen van nieuwe heuristieken gebruiksvriendelijk: we hebben we een domeinspecifieke taal voorzien voor het combineren van de primitieve heuristieken. Een nieuwe zoekheuristiek kan dus sneller en betrouwbaarder geïmplementeerd worden. Gezien het praktisch belang van constraint solving is dit een belangrijk resultaat. We kunnen dan ook veronderstellen dat modulaire zoekheuristieken in de toekomst de standaard voor het oplossen van dergelijke problemen zal worden.

Trefwoorden

Constraint programming, search combinators, traits, DSL, heuristieken

Modular search heuristics in Scala

Benoit Desouter

Supervisor(s): Tom Schrijvers, Nicolas Van Cleemput

Abstract— **Real world problems often need heuristics to be solved efficiently. However, existing constraint solvers do not proficiently support the development of new heuristics.** *Search combinators* **are a new approach to this, but the existing implementation in C++ has several problems which compromise extensibility. This article presents the modular implementation of search combinators in Scala. We provide a new constraint-solver independent implementation based on traits featuring a clean design minimizing code duplication and casts. It provides a domain specific language for composing combinators into a custom heuristic for solving a constraint satisfaction problem.**

Keywords— **Constraint programming, search combinators, traits, DSL, heuristics**

I. Introduction

MANY real-world problems need heuristics to be solved efficiently. Search combinators are an approach to efficiently develop custom heuristics for solving constraint satisfaction problems. This ability can be essential for solving combinatorial problems [1]. Existing constraint solvers like Gecode [2] or MiniZinc [3] do not support this well: either only a few predefined heuristics are available, or it becomes a large effort to implement a new heuristic in a low-level general programming language.

Search combinators were developed with extensibility in mind. An implementation in C++ has been discussed in [1], [4]. Based on this code, we have created a new implementation in Scala, trying to make a compromise between static composition of combinators and dynamic composition as suggested in these papers. More concretely, we made use of Scala's advanced inheritance mechanism, *traits*, to

- use static composition instead of an explicit implementation of mixin inheritance [5],
- lower the amount of duplicated code,
- refactor the design in order to avoid casts as much as possible, and
- provide a clean domain specific language for composing combinators.

Scala seems like a promising implementation language for realising these goals, because of its support for traits [6] and domain specific languages. It is platform-independent, running on the Java virtual machine. Operators can be implemented as methods, functions are first-class, and case classes provide objects much like algebraic data types.

The structure of this article is as follows: in the next section we introduce combinators, the major problem that arises in implementing them and how we solve it. In Section III we discuss how to solve the all-interval problem and explain the fundamentals of constraint solving. Next, we give a quick overview of available combinators and constraints. Section V discusses the implementation in depth. We evaluate our work in Section VI. We conclude in Section VII.

B. Desouter is a Computer Science student, Ghent University (UGent), Ghent, Belgium. E-mail: Benoit.Desouter@UGent.be .

II. Overview

Search combinators are building blocks for creating new search heuristics. These derived heuristics can in turn be used as a building block. Hence we use the term heuristic and combinator interchangeably.

Combinators work together by means of a message protocol [1]. Each combinator supports the messages presented in Figure 1. However, some combinators do not act on every message, while others combinators have common behaviour. Extracting common behaviour in a superclass is not a feasible option because of the number of combinations. Moreover a superclass should not be used only for factoring code. Each class has a semantic meaning [6]. Traits can help us overcome this problem.

A trait is a fundamental unit of code reuse. It encapsulates methods and field definitions, which can be mixed in in any class. Each class can have a single superclass but can mix in any number of traits [7]. The precedence between traits is determined by *linearisation*.

Using traits allows us to define a complete heuristic without explicitly coding all the methods. An example will make this clear (Figure 2): the Print combinator has each of the methods listed in Figure 1 by mixing them in from several traits. Likewise, we have implemented many basic heuristics. In terms of these we have defined more advanced heuristics, for example branch-and-bound. We use these to solve combinatorial problems.

III. Constraint solving fundamentals

In Figures 3 and 4 we show how to solve the "all-interval problem". This problem is as follows: find an all-interval series of size $n \in \mathbb{N}$, which is a vector $s = \langle s_1, \ldots, s_n \rangle$, that is a permutation of $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ and the vector v of differences $\langle |s_2 - s_1|, |s_3 - s_2|, \ldots, |s_n - s_{n-1}| \rangle$ is a permutation of $\mathbb{Z}_n \setminus \{0\} = \{1, 2, \ldots, n-1\}$.

I first make a model of the problem: I create variables and constrain them. The space keeps track of the variables and their domain. The idea is to reason about the relations between the variables and refine their domains. Whenever no more knowledge can be derived,

```
def next(currentNode: Node, nextNode: Node)
def enter(sm: StateManager, na: NodeAllocator, top:
    SearchHeuristic, currentNode: Node): Int
def child(sm: StateManager, na: NodeAllocator, top:
    SearchHeuristic, currentNode: Node, childNumber: Int):
    Option[Node]
def stat(node: Node)
def init(node: Node)
def copy(oldNode: Node, newNode: Node)
```

Fig. 1. All messages of the protocol for combinators.

```
package searchCombinators
import node._
case class Print(
  protected val subCombinator: SearchHeuristic,
  variableOrArray: DomainVariableOrArray,
  visualizer: Visualizer = Visualizer
  ) extends SearchHeuristic with SubCombinator
    with OnEnterSolution {
  // First OnEnterSolution!
  def onEnterSolution() {
      val stringBuilder = new StringBuilder()
      /* ... Fill stringBuilder ... */
      visualizer.shout(this, stringBuilder.toString)
  }
}
```

Fig. 2. An example heuristic defined without explicitly coding all methods.

```
package searchCombinators.examples
import searchCombinators._
/** Model for all-interval series problem. */
class AllInterval(size: Int, space: Space) {
  assert(size > 2)
  val numbers = IntVarArgs(size,0, size-1,space)
  val differences = IntVarArgs(size-1,1,size-1,space)
  // Impose constraints
  // Set up variables for distance
  0 until size-1 foreach { i =>
    space.imposeConstraint(Distance(numbers(i+1),numbers(i),
      differences(i)))
  }
  space.imposeConstraint(Alldiff(numbers))
  space.imposeConstraint(Alldiff(differences))
  // Break mirror symmetry
  space.imposeConstraint(numbers(0) < numbers(1))
  // Break symmetry of dual solution
  space.imposeConstraint(differences(0) > differences(size-2))

}
```

Fig. 3. Modelling the all-interval series problem.

```
object AllInterval extends App {
  val space = Space()
  val model = new AllInterval(5, space)
  val searchHeuristic = Print(IntSearch(model.numbers,
    List(MinDomSize()),BisectHigh()),model.numbers)
  Runner.run(searchHeuristic,space)
}
```

Fig. 4. One possible heuristic for solving the all-interval series problem.

add an extra constraint to the space until a solution or a contradiction is found. Of course, you should not throw away solutions. So whenever you add an extra constraint, you need to backtrack and investigate the addition of its negation.

In theory it does not matter which constraint you add, but in practice it is critical for the ability and speed of finding a solution. Clearly you end up with a tree of decisions. Leaf nodes are either a solution to the problem or indicate a contradiction. With combinators we can define and explore several heuristics for searching this tree. In Figure 4 we create a simple strategy and run it. If we think of a better strategy, we only need to change the code a little.

Before discussing the available heuristics, let us work out a simple example of the reasoning and the addition of extra constraints. Suppose we have three variables X, Y and Z in $[0, 5]$. Also, $X + Y = Z, X > 0$ and $Y > 0$. Then we can derive that X and $Y \in [1, 4]$ and that $Z \in [2, 5]$. At this point we are stuck. So we first suppose $X > 3$ and afterwards we will come back to $X \leq 3$ (depicted in Figure 5).

IV. Available combinators and constraints

We have implemented 21 combinators. Discussing all of them is outside the scope of this paper, but we present the most important ones:

- An *integer search* attempts to find a solution for the given (array of) variable(s) using the given strategy for selecting variables and values. Every practical heuristic using integer variables eventually needs one or more instances of this combinator.
- *Prune* cuts the search tree below the current node. It is used if our heuristic decides this is not an interesting part of the tree.
- The *binary and* combinator initially performs its first heuristic and then its second. Thus its subsearches are performed at increasingly deeper levels of the tree [4]. A variable input *and* allowing more then two subcombinators is available.
- The *if* combinator uses its first subsearch as long as a condition c is true. Afterwards, it uses the its second subsearch.

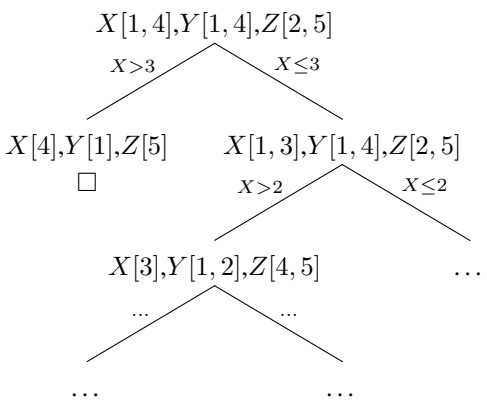


Fig. 5. The addition of extra constraints leads to a tree structure.

```
/* Make a new integer domain variable with domain [1,10]. */
val foo = IntVar (1,10)
/* Make another integer domain variable. */
val foo2 = IntVar (5,10)
/* Make an array of domain variables with domain [1,10]. */
val arrFoo = IntVarArgs (3,1,10)

/* Variable value must equal some given value. */
val constraint1 = foo := 5
/* Variable values must differ. */
val constraint2 = foo != foo2 // or =/=
/* Variable value must be less than given value. */
val constraint3 = foo < 5
/* Variable value must be >= than some other variable value. */
val constraint4 = foo >= foo2
/* All different */
val constraint5 = Alldiff (arrFoo)
```

Fig. 6. Some of the available constraints.

We provide several constraints. These are implemented using case classes and must be mapped onto those of the constraint solver. The syntax for constraints in the JaCoP constraint solver inspired our own, but we use well-known symbols whenever possible [8]. Figure 6 lists some of them.

In the next section, we delve into the implementation. As said before, we focus on the structure of the code and user-friendliness, not on the behaviour of the combinators.

V. Implementation

We focus on several aspects of our implementation. We start with traits and go on to control abstractions. Third, we discuss syntactic sugar for our domain specific language. Finally we explain solver independence.

A. Traits

Combinators are defined as subclasses of the class `SearchHeuristic`. We make their implementation easier by defining traits of common behaviour, and by implementing additional control abstractions. We introduced seven traits for helping with the combinator's implementation. Like the combinators themselves, their power is in their combination and their order of precedence:

1. Several methods are remarkably often empty. Therefore `EmptyCombinator` provides an empty implementation of these methods.
2. In other combinators, a method only calls the method of the same name on the subcombinator. This is implemented in `SubCombinator`.
3. Combinators that cannot have subcombinators should throw an exception should the `child`-method ever be called (`NoChildMethod`)
4. A frequently recurring pattern features the `enter` method being called on the subcombinator and if a solution is found, some additional action must be taken. We provide `OnEnterSolution`: if you mix it in, `enter` should not be implemented, but instead `onEnterSolution`. This avoids repeating the pattern of calling the subcombinator and checking for a solution.
5. Similarly, `OnChild` allows to execute a programmer-defined side-effect only if a child node was returned by the subcombinator.
6. Trait `DoAndSucceedEnter` allows a programmer-supplied side-effect before succeeding.
7. And finally, `CopyItem[T]` helps managing the local state of the combinators.

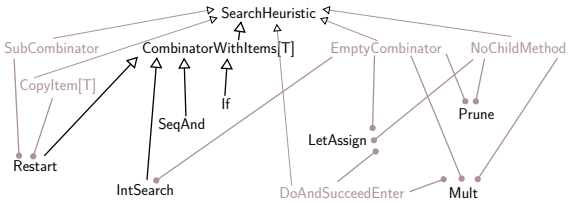


Fig. 7. Structure of some combinators and the traits.

```
trait OnChild extends SearchHeuristic {
  abstract override def child(/* ... */): Option[Node] = {
    val maybeChild = super.child(/* ... */)
    maybeChild match {
      case Some(ch) ⇒ onChild(currentNode,ch)
      case None     ⇒ ;
    }
    maybeChild
  }
  def onChild(currentNode: Node, childNode: Node)
}
```

Fig. 8. Example of defining a trait: `OnChild`.

The resulting structure can only be conveniently shown for a subset of the combinators (Figure 7). We use a gray line ending in a bullet for indicating trait inclusion, and use the conventional notation for normal inheritance. For reasons of clarity we did not indicate that the `LetAssign`, `Prune` and `Mult` combinators also inherit from `SearchCombinator`. Thus, a trait can be defined similar to a class (Figure 8). It has a set of supplied and needed methods.

B. Control abstraction

As an example of a control abstraction, we discuss an aspect of the *if* combinator listed in Figure 9. It is implemented as a regular method, but it has a function as one of its arguments. To be more widely usable, the method has been made generic. To use it, we have to give it the current node from which we extract the condition flag (`false` if condition still true), and a function that takes the node and the flag as arguments. The return value of the abstraction is that of the function, which can be `()` (read: "unit", in some way Scala's equivalent of the Java keyword `void`).

In its usages, we see an anonymous function is often used as an argument for the control abstraction. This type of functions lists it parameters before the arrow, using parentheses if there is more than one, and lists its

```
case class If
  (private val cond: VarCond,
   private val then: SearchHeuristic,
   private val `else`: SearchHeuristic)
  (implicit nodeAllocator: NodeAllocator)
  extends CombinatorWithItems[Boolean](nodeAllocator) {

  /* .. methods not using control abstraction left out ... */

  /** Control abstraction simplifying the implementation. */
  private def ifCondition[T](nodeContainingFlag: Node,
    action: Function2[SearchHeuristic, Boolean, T]): T = {
    val conditionFlag = lookupItem(nodeContainingFlag)
    if (conditionFlag) {
      action(`else`, conditionFlag)
    } else {
      action(then, conditionFlag)
    }
  }

  def child(/* ... */): Option[Node] = {
    ifCondition(currentNode, (sub, conditionFlag) => {
        val maybeChild = sub.child(/* ... */)
        /* Another control abstraction */
        ifChild(maybeChild) { insert(_, conditionFlag) }
        maybeChild
      })
  }

  def stat(node: Node) = {
    ifCondition(node,(sub,_) => sub.stat(node))
  }

  def copy(oldNode: Node, newNode: Node) = {
    val conditionFlag = ifCondition(oldNode,
        (sub, conditionFlag)
        => { sub.copy(oldNode,newNode); conditionFlag } )
    insert(newNode,conditionFlag)
  }
}
```

Fig. 9. Control abstraction for the "if" combinator.

result afterwards. The result can be a block statement, surrounded by curly braces. For example, an anonymous square function can be defined as follows:

```
val foo = (x: Int) => x * x
```

In the above code the type of x has to be explicitly mentioned because it cannot be derived from the context. The most typical occurrences of anonymous functions are those with a single statement. The underscore sign is a dummy indicating we are not interested in the value of this parameter.

C. Syntactic sugar

More complex heuristics are implemented in terms of the 21 basic combinators. To make this more readable, syntactic sugar has been added. A good way to define them is as a function in the package object. As an example, a *for* combinator can be implemented using *let* and *restart* combinators (Figure 10).

The Figure also shows some of the syntactic sugar is implemented in the package object. In Table I we list the syntactic sugar for combinators. Some combinators have a condition as a parameter; syntactic sugar is also

```
package object searchCombinators {
  /** Factory function for for combinator. **/
  def For(letVar: LetVar, lower: Int, upper: Int, sh:
   SearchHeuristic): SearchHeuristic = {
    Let(letVar, lower, Restart(letVar <= upper,
      sh | (+letVar & %)))
  }

  /** Syntactic sugar for prune combinator. **/
  def % = Prune()

  /** Syntactic sugar for true condition. **/
  def @@ = TrueCond()
}
```

Fig. 10. The "for" combinator is derived from "let" and "restart".

present for these conditions. It is for example possible to state that one variable is smaller that another using the symbol $<$.

Most of the sugar for combinators is implemented as methods of abstract class `SearchHeuristic`. However, we often need quite a lot of parentheses. This is because the priority of operators is fixed. If we want to choose our symbol such that it has a logical meaning, it often does not have the right priority. Luckely enough, it is not a crucial issue for providing a DSL.

Traits and abstract types go well together. The combination has extensively been used in the implementation of conditions and for providing constraint-solver independence.

D. Solver independence

We have already mentioned that the system is constraint solver independent. For demonstration purposes, we used the JaCoP constraint solver. This solver is more suited than Gecode because it is implemented in Java and not in C++. Java integrates seamlessly with Scala. Gecode no longer has Java bindings available. A new solver can be plugged in by implementing the following:

- the traits for variables and arrays (`IntVar`, ...) mapping our variables onto those of the constraint solver.
- the state manager, which provides backtracking for spaces. To allow this, a constraint solver must provide enough methods in its API.
- the space, which main task is to map our constraints onto those of the constraint solver.

VI. Evaluation

We have evaluated several aspects of our system. We first discuss trait effectivity. Next, we focus on code size. Performance comes last.

A. Trait effectivity

With a total of seven traits and 21 basic combinators, shared behaviour is quite common. A single combinator mixes in 1.76 traits on average, with a standard derivation of 1.06. Thus there are some combinators in which it is quite easy to extract common behaviour, allowing

TABLE I
AVAILABLE SYNTACTIC SUGAR FOR COMBINATORS.

Symbol	Combinator
`A & B`	`SeqAnd(A,B)`
`A | B`	`SeqOr(A,B)`
`!A`	`Once(A)`
`c ? A <> B`	`If(c,A,B)`
`x := i`	`IntAssign(x,i)`
`x := y`	`LetAssign(x,y)`
`x := d`	`DVarAssign(x,y)`
`+x`	`Incr(x)`
`%`	`Prune()`

us to mix in as much as three traits. Others use only one or do not use any trait. These are the most complex heuristics in terms of implementation, but are absolutely necessary: integer search mixes in one trait, *and* and *if* do not mix in any. This is absolutely due to their specialized and complex behaviour; they have more than one subcombinator. Traits can only be mixed in once, and cannot be renamed on at compilation time. So it is only logical it is much harder to extract common behaviour.

Mixing in three traits is quite a lot. But what does this mean in terms of work left? We calculated that out of seven messages needed for sequential operation[1] only 2.19 still need to be implemented on average, with a standard derivation of 1.79. As can be intuitively expected, this is due to the same complex combinators.

Thirteen combinators only need one message. This is the "enter" message, or method that can be implemented as a substitute, because that is what defines the behaviour of the combinators most. It defines how to process a node.

B. Size of the code base

In terms of number of code lines, we have reached a constraint-solver independent design in about 1900 eighty character lines, including coupling to the JaCoP constraint solver. That coupling takes 215 lines. The C++ code is about 3250 lines. However, this is physical line count and C++ code lines are wider. At the moment of writing, there are no tools available for measuring logical line count that support Scala. Moreover, we have used much longer identifiers. The numbers only tell us something about the order of magnitude of the development and maintenance effort.

We had expected a bigger overall reduction in code size, but have identified a few reasons for why this is not the case:

- Solver independence asks for a more complex design, with more classes and types than is necessary in a system that only works together with a single constraint solver.
- Adding a user-friendly DSL requires extra code compared to the case where user-friendliness is not present at all.
- Due to absence of many imperative-style looping constructs in Scala, it is more difficult to write complex combinators like *and*, *restart* or *if*. It is not always straightforward to rewrite their loops in a functional style because there are often several complicated stopping conditions. You end up writing simple 'while' loops and increase counters yourself if you do not think too hard. That practice takes loads of code lines.

From the list above, it is clear the increase in code size is situated in the library of the system. It is more interesting to look at that part of the code users will most come in contact with and not at the back-end. We do have achieved impressive code size reduction in that part.

It takes less code to add a new, reasonably simple, combinator, to combine them in a new heuristic and to solve a constraint satisfaction problem using these heuristics. For example, solving the all-interval problem takes about thirty lines in our Scala implementation, while it takes about 70 in the original C++ implementation.

C. Performance

We have implemented both a general and specific coupling to the JaCoP constraint solver. We evaluate both and discuss.

C.1 General coupling

In terms of performance, our system is slow when using a general coupling with the JaCoP constraint solver. We expected this coupling to be slow, since it requires starting from the root space every time we want to process a node. But it is general: it can be used in whatever order the solution tree is searched. We have used the 'all interval' and 'Golomb ruler' problems as test cases.

For the 'all interval' problem we varied problem size from five to twelve (included). Execution times are ignorably small for size smaller than seven (Figure 11). At problem size eleven, it takes about eleven seconds to find the solution. At that point execution times quickly become huge. At size twelve, the program finishes after somewhat more a minute.

For purposes of comparison, we have measured the execution times of a traditional implementation of the problem using the same constraint solver, JaCoP, and making use of its Scala bindings. Problem size eleven here only takes $1085\,\mathrm{ms}$ on average. But it remains true execution times increase very quickly: at problem size twelve, mean execution time already is $5124\,\mathrm{ms}$.

Concerning the Golomb ruler problem, execution times are even worse (Figure 12). We used the more complex branch-and-bound heuristic here. Argument size varied from three to five. We stopped execution for argument size six after two hours. Comparing to a traditional approach, argument size eleven there requires about two minutes. But it is obvious even with a traditional approach, we will not be able to go much further

[1]The C++ code supports parallelism but therefore needs another message, which we did not count in our comparison with our sequential implementation.

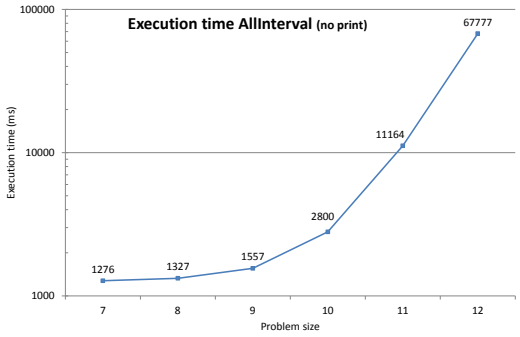


Fig. 11. Execution times for the 'all interval' problem.

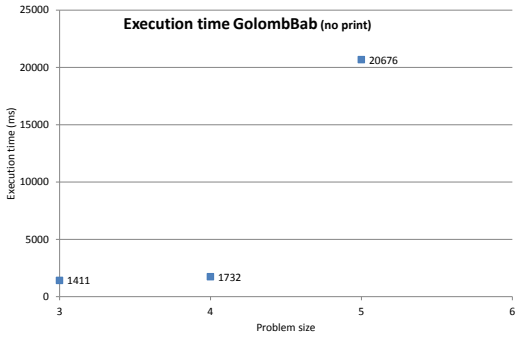


Fig. 12. Execution times for the 'Golomb ruler' problem.

on our standard PC: the problem of size ten only took 6900 ms on average.

C.2 Depth-first coupling

We have also written a specific coupling that can only be used if the search tree is traversed in depth-first style. We expected its performance to be better because it can create a new space for a certain node by applying modifications to the space associated with the parent node, rather than starting from the root space.

For the 'all interval' problem better performance is very obvious starting at argument size eleven (Figure 13). Execution time is a little less than three seconds on average, compared to the earlier eleven seconds. The problem of size thirteen can now be solved in 44 s on average. For smaller problems, the gain is less obvious but still measurable. For sizes eleven to thirteen, execution time is in the same order of magnitude as that of a traditional implementation.

The 'Golomb ruler' problem exhibits a considerable performance improvement at size five (Figure 14). For smaller sizes, improvement is less remarkable. Despite the improvements, the program is still slower than a traditional implementation by orders of magnitude.

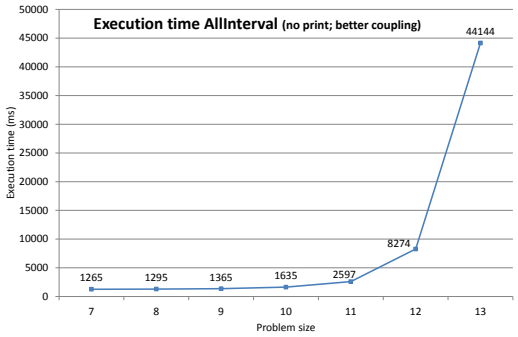


Fig. 13. Execution times for the 'all interval' problem optimised for DFS.

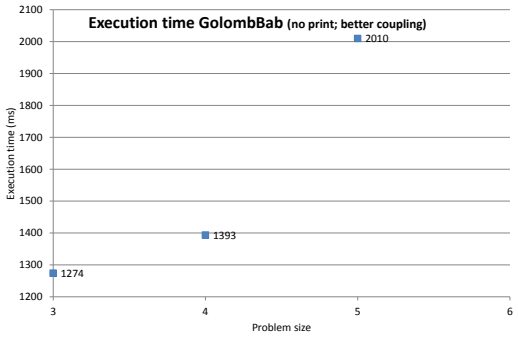


Fig. 14. Execution times for the 'Golomb ruler' problem optimised for DFS.

C.3 Discussion

We can conclude our implementation in combination with the first coupling is not usable for any practical purposes. Indeed performance was not the focus of this master dissertation. The difficulty lies in making a good coupling for a great deal. Comparing our results to those from the original implementation in C++, we see our implementation is slower by several orders of magnitude [1]. However, it is misleading to compare performance as we use a different solver.

The paper also states combinators work without runtime overhead compared to a traditional implementation. In our case, that is definitely not true. For the 'all interval' problem, results are in the same order of magnitude. For that particular problem, we used a simple *integer search* combinator. Thus, we hope to be able to achieve a fair result for heuristics that consist of a limited number of combinators by paying some attention to performance (less indirections, better designed solver independence).

VII. Conclusions and Future Work

The trait system has proven to be quite effective in extracting common behaviour between combinators. It

has also helped us refactoring the design. We succeeded in avoiding casts as much as possible. Our system is fully constraint-solver independent and due to Scala's support for DSL's and case classes, it is quite usable. However it is a pity we cannot redefine the priority of the operators. In contrast, the ability to define own control abstractions and make them look as if they were language syntax is well-developed, but of less use in this thesis.

Scala has allowed us to make a design where casts are only used in the constraint-solver specific code. In fact, with some additional, but system-wide, work, we think it would be possible to completely eliminate these. We did not investigate this further.

We think it would be nice to visualize the active combinators and the search tree during the search. This could be done using a tool like CP-VIZ [9], [10]. Future Scala versions or derived languages could address redefinition of the priority of the operators.

We believe the Scala language and the ideas behind it, the combination of the functional and object-oriented paradigm and its mixin-like inheritance, are indeed very powerful. Personally, the learning curve was not too steep; however it takes you longer to fully master the language than languages with a lower level of abstraction like Java.

REFERENCES

[1] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey, "Search combinators" in *Proceedings of the 17th international conference on Principles and practice of constraint programming*, Berlin, Heidelberg, 2011, CP'11, pp. 774–788, Springer-Verlag.

[2] The Gecode constraint solver website, `http://www.gecode.org/`, last visited May 18, 2012.

[3] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack, "Minizinc: towards a standard CP modelling language" in *Proceedings of the 13th international conference on Principles and practice of constraint programming*, Berlin, Heidelberg, 2007, CP'07, pp. 529–543, Springer-Verlag.

[4] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey, "Search combinators", Unpublished.

[5] Gilad Bracha and William Cook, "Mixin-based inheritance", *SIGPLAN Not.*, vol. 25, no. 10, pp. 303–311, September 1990.

[6] Martin Odersky and Matthias Zenger, "Scalable component abstractions", *SIGPLAN Not.*, vol. 40, no. 10, pp. 41–57, Oct. 2005.

[7] Martin Odersky, Lex Spoon, and Bill Venners, *Programming in Scala*, Artima Press, California, 2nd edition, 2010.

[8] The JaCoP constraint solver website, `http://www.jacop.eu/`, last visited May 18, 2012.

[9] Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson, "A generic visualization platform for CP" in *Proceedings of the 16th international conference on Principles and practice of constraint programming*, Berlin, Heidelberg, 2010, CP'10, pp. 460–474, Springer-Verlag.

[10] CP-VIZ visualization platform website, `http://cpviz.sourceforge.net/`, last visited March 21, 2012.

Inhoudsopgave

Hoofdstuk 1

Inleiding

In dit inleidend hoofdstuk willen we de context van deze thesis schetsen. We leggen uit welke hoogniveau beslissingen we genomen hebben, en waarom.

1.1 Het nut van modulaire zoekheuristieken

Veel praktische problemen zijn van combinatorische aard. Denken we maar aan het opmaken van een productieplanning, een lesrooster of een dienstregeling. Deze problemen hebben een zeer grote zoekruimte waarin de oplossingen zich ergens bevinden. Constraint programming is een programmeerparadigma dat probeert die zoekruimte te verkleinen. De tools die hiervoor gebruikt worden noemen we *constraint solvers*. Constraint programming heeft een aantal standaardheuristieken voor zoeken naar oplossingen. Daarnaast is het belangrijk om zoveel mogelijk domeinkennis te gebruiken om sneller tot een oplossing te komen, of om grotere problemen te kunnen oplossen. Die kennis is in feite een eigen, domeinspecifieke, heuristiek, maar kan helaas vaak niet gemakkelijk ingeplugd worden in een constraint solver.

Modulaire *zoekheuristieken* werden ontwikkeld om een inherent probleem van de huidige constraint solvers op te lossen: er is nood aan een effectieve manier om nieuwe zoekstrategieën te definiëren. Een adequate zoekheuristiek maakt immers het verschil tussen het effectief oplossen van een combinatorisch probleem en een faling [20]. Bestaande solvers kunnen hier vaak niet aan tegemoet komen: ondersteuning voor nieuwe heuristieken toevoegen is

- al vlug een niet te onderschatten hoeveelheid werk, hetzij voor de eindgebruikers, hetzij voor de ontwikkelaars van het systeem.
- vaak niet mogelijk omdat het systeem hier architecturaal niet is op voorzien.

Modulaire zoekheuristieken vertonen een modulair design en zijn ontworpen om te eenvoudig te worden gecombineerd tot meer complexe heuristieken. Daarom gebruiken we ook wel de term *combinators*. In het vervolg van het document gebruiken we die term die het best in de context past: met „combinators" leggen we meer de nadruk op het combineerbaar zijn; met „zoekheuristieken" benadrukken we dat elke bouwsteen ook voldoet aan de interface. Primitieve bouwstenen hebben echter een zeer beperkt nut op zichzelf.

1.2 Doelen voor de realisatie in Scala

Een implementatie van modulaire zoekheuristieken, in C++, wordt reeds besproken in de papers die het concept introduceren [20, 19]. Met deze — efficiënte — implementatie zijn echter enkele structurele problemen, die we in deze thesis pogen te verhelpen:

1. *codeduplicatie* die niet eenvoudig te verhelpen lijkt met standaard enkelvoudige overerving.

2. de code staat zelf in voor het vrij complexe *geheugenbeheer*.

3. het implementeren van een nieuwe zoekheuristiek vraagt nog steeds vrij veel code: enerzijds *boilerplate code* en anderzijds ook omwille van het *laagniveaukarakter* van C++.

4. de code bevat een *expliciete implementatie* van een dynamisch overervingsmechanisme dat toelaat een heuristiekspecificatie at runtime samen te stellen, *mixin inheritance* [20].

5. de gebruikte *constraint solver* library, Gecode (beschikbaar op `http://www.gecode.org`), is vrij diep verankerd in de code.

6. de code bevat veel *casts*, wat de typeveiligheid niet ten goede komt.

7. de code bevat heel wat terugkerende patronen die telkens opnieuw volledig uitgeschreven worden.

In Codefragment 1.1 illustreren we deze problemen: op regel 9 zien we een voorbeeld van een vaak terugkerend patroon. We kunnen deze controleflow beter abstraheren. Het merendeel van de methodes roept enkel `super` op. In regel 29 zien we dat we zelf de grootte van de items in de combinatorboom moeten berekenen om de juiste hoeveelheid bytes te kunnen alloceren. In dit geval heeft een printcombinator geen eigen items, maar zijn subcombinatoren misschien wel.

Codefragment 1.1: Structurele problemen van de C++ implementatie.

```
1  class Print : public SearchHeuristic {
   protected:
3    /* Left out field definitions. */
   public:
5    Print(const char* x0, SearchHeuristic* s0, bool bv=false) : x(x0), s(s0), is_boolVar(bv) {}
     virtual int enter(/* ... */)
7    {
       int c = s->enter(sm,vm,na,top,n);
9      if (c == SOLUTION) { /* Frequently repeated pattern. */
           /* ... */
11       } else {
           /* ... */
13       }
       }
15     return c;
     }
17   virtual void next(Node* n0, Node* n1)
     {
19     s->next(n0,n1); /* Boilerplate code */
     }
21   virtual bool child(/* ... */)
     {
23     return s->child(sm,vm,na,top,n,i,ret);
     }
25   virtual void stat(Node* n) {
       s->stat(n); /* Boilerplate code */
27   }
     /* Manual bookkeeping. */
29   virtual size_t itemSize(void)
     {
31     return s->itemSize(); /* Boilerplate code */
     }
33   virtual void init(Node* n, VarAccessor::VM& vm, LetBinding& env) {
       s->init(n,vm,env);
35     idx = VarAccessor::idx(&vm,x);
```

```
    size = VarAccessor::size(&vm,x);
37  }
    virtual void dispose(StateMgr& sm, NodeAllocator& na, SearchHeuristic* top, Node* n)
39  {
    s->dispose(sm,na,top,n);/* Boilerplate code */
41  }
    virtual void copy(Node* n0, Node* n1) { s->copy(n0,n1); } /* Boilerplate code */
43  virtual void steal(Node* n0, Node* n1) { s->steal(n0,n1); } /* Boilerplate code */
    /* Manual (complex) memory management */
45  virtual ~Print(void) {
    delete s;
47  }
};
```

We willen dan ook een nieuwe implementatie creëren met volgende eigenschappen:

- verminderde (of volledig vermijden van) codeduplicatie door gebruik van geavanceerde overervingsvormen.
- verminderen van de implementatietijd van een nieuwe zoekheuristiek door het vermijden van boilerplate code, en door het verhogen van het abstractieniveau. Terugkerende patronen kunnen worden gedefinieerd als een abstractie.
- verbeteren van typeveiligheid, enerzijds door het vermijden van casting, en anderzijds door zoekspecificaties statisch te gaan samenstellen.
- algemene reductie van codeomvang.
- onafhankelijkheid van de specifieke constraint solver library. Het is altijd aangenaam wanneer je systeem in vele contexten kan gebruikt worden.

Dit alles willen we bereiken met liefst zo weinig mogelijk performantieverlies, alhoewel het duidelijk is dat we er hoogstwaarschijnlijk wel zullen bij inschieten.

1.3 Waarom Scala?

Nu lijkt Scala een interessante taal om deze doelen proberen te verwezenlijken, en wel om de volgende redenen. Scala heeft een nieuwe, geavanceerde overervingsvorm, *traits*, die toelaat methodes te bundelen en te hergebruiken waar nodig. Hiermee hopen we codeduplicatie te verminderen, boilerplate code te vermijden, en zoekspecificaties statisch te gaan controleren op typecorrectheid. Ten tweede biedt Scala ondersteuning voor het inbedden van domeinspecifieke talen, wat ons moet toelaten het abstractieniveau te verhogen. Onder meer door invloeden uit functionele talen en door typeinferentie heeft deze taal een compacte syntax van een hoog niveau. De taal draait bovenop de Java virtuele machine, waardoor we bevrijd zijn van het manuele geheugenbeheer, en de resulterende code platformonafhankelijk is.

1.4 Structuur van dit werk

In het volgende hoofdstuk belichten we de features van Scala die niet bekend zijn bij het grote publiek. Daarnaast geven we de nodige achtergrond over constraint programming en tonen we hoe ons systeem kan gebruikt worden voor het oplossen van praktische problemen. In Hoofdstuk 3 focussen we op technische uitwerking: de codestructuur en de verschillende technieken die we hebben toegepast. Het daaropvolgende hoofdstuk evalueert onze code. We eindigen met conclusies en suggesties voor toekomstig werk.

Hoofdstuk 2

Achtergrondkennis

In dit hoofdstuk bespreken we de voorkennis die nodig is voor een goed begrip van de technische hoofdstukken die volgen. We leggen eerst uit wat het doel is van constraint programming en hoe het oplossen van een praktisch probleem in zijn werk gaat. Aangezien ons eigen systeem ontwikkeld is in Scala, introduceren we daarna enkele minder bekende concepten uit de taal. Hierbij gaan we uit van kennis van een objectgeoriënteerde programmeertaal met enkelvoudige overerving, zoals Java. Aan het einde van dit hoofdstuk tonen we hoe praktische problemen kunnen worden opgelost met onze implementatie van modulaire zoekheuristieken. Na dit hoofdstuk is de lezer in staat technische details van de implementatie te begrijpen.

2.1 Constraint programming

Constraint programming is een techniek om problemen op te lossen waarbij de oplossingen moeten voldoen aan een reeks van beperkingen. Die beperkingen worden opgelegd op een manier die doet denken aan wiskundige stelsels. Constraint programming is dus een *declaratieve* programmeertechniek. Ter illustratie tonen we een eenvoudig probleem in Voorbeeld 2.1.

Voorbeeld 2.1 *„Een boer beschikt over 20 meter prikkeldraad en wil hiermee een wei voor vier koeien afspannen. Elke koe heeft zes vierkante meter nodig, zodat de totale oppervlakte van de wei 24 vierkante meter is. Welke lengte en breedte kan deze weide hebben?". We noteren x voor de lengte en y voor de breedte. Dan hebben we de volgende beperkingen:*

$$x \times y = 24 \tag{2.1}$$

$$2(x + y) = 20 \tag{2.2}$$

$$x \leq y \tag{2.3}$$

Een oplossing is een toekenning aan één of meerdere variabelen, zodat alle beperkingen voldaan zijn, en indien gewenst, een bepaalde objectieffunctie gemaximaliseerd of geminimaliseerd wordt. Deze variabelen behoren tot een eindig domein, in de praktijk meestal een eindige deelverzameling van de gehele getallen $\mathbb{Z}$. De techniek is schaalbaar in die zin dat de solver niet alle oplossingen hoeft te genereren en testen (het generate-and-test paradigma bekend van Prolog). In de plaats daarvan maakt men gebruik van de verbanden tussen de variabelen om oplossingen te elimineren.

Een eenvoudig voorbeeld maakt dit duidelijk: veronderstel drie variabelen X, Y en Z met een domein $[0, 5]$. Veronderstel bovendien dat $X + Y = Z$, $X > 0$ en $Y > 0$. Dan kunnen we afleiden dat X en Y zeker in $[1, 4]$ moeten liggen, en dat Z moet liggen in $[2, 5]$. Op dit moment kunnen we niet verder zonder bijkomende veronderstellingen te maken. Willen we echter geen oplossingen verliezen, dan moeten we een gevalsanalyse maken waarin we — in het ene geval — een bijkomende beperking C opleggen, en in het andere geval $\neg C$. We kunnen bijvoorbeeld eerst

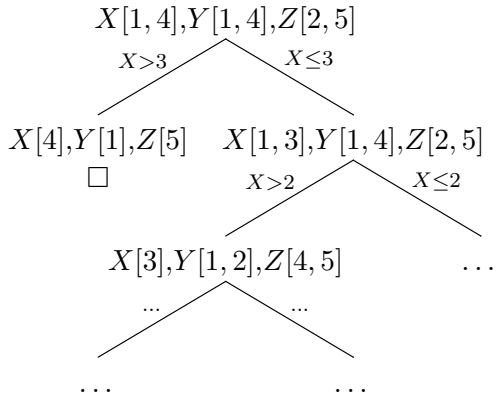
veronderstellen dat $X > 3$ en daarna $X \leq 3$. Nu kunnen we weer verder de domeinen inperken, tot op het punt waar we opnieuw een gevalsanalyse moeten maken. Een *constraint solver* kan deze redeneringen automatisch maken.

Door het afwisselen van domeininperking door redeneren, in de literatuur *propagatie* genoemd, en het maken van bijkomende veronderstellingen, bekomen we een boomstructuur waarbij in elke knoop een bijkomende veronderstelling wordt gemaakt, en waarbij de bladeren ofwel een oplossing voorstellen, ofwel een tegenstrijdigheid vertonen. Het is duidelijk dat niet alle oplossingen even diep in de boom zitten, zoals ook geïllustreerd in Figuur 2.1. Er is heel wat keuzevrijheid welke bijkomende veronderstelling men maakt, en hiermee kan men de structuur van de boom, en dus de uitvoeringstijd van het programma, zeer sterk beïnvloeden. Wanneer men bijvoorbeeld een bepaalde objectieffunctie wenst te maximaliseren, kan de solver bijvoorbeeld meer snoeien in de boom wanneer aan het begin een goede kandidaatoplossing gevonden wordt. Zoals voor alle branch-and-bound problemen, kan men een snellere oplossing bekomen door meer *domeinkennis* te gebruiken.

Voor het maken van een bijkomende veronderstelling, onderscheidt men heuristieken voor het kiezen van de variabele, en voor het kiezen van de waarde waarop men gaat splitsen. De keuze voor een bepaalde variabele kan bijvoorbeeld gebeuren omdat deze variabele:

- het kleinste domein heeft. Dit noemt men *first fail*.
- het grootste domein heeft.
- de grootste/kleinste ondergrens of bovengrens heeft.
- al tot heel veel tegenstrijdigheden heeft geleid. In dit geval heeft deze variabele de hoogste *accumulated failure count*.
- er heel veel beperkingen gebruik maken van deze variabele, en er dus veel propagatie zal kunnen gebeuren.
- ...

Als waarde waarop men gaat splitsen, kan men bijvoorbeeld de mediaan, het gemiddelde, een random getal uit het domein, of zelfs de onder- of bovengrens nemen. Nog tal van andere heuristieken zijn denkbaar.



Figuur 2.1: Door het maken van bijkomende veronderstellingen bekomen we een boomstructuur.

2.1.1 Constraint solvers in de praktijk

Constraint solvers zijn zowel geïmplementeerd als bibliotheek in uiteenlopende programmeertalen, als aangeboden als stand-alone tool. Beide aanpakken hebben hun voor- en nadelen:

- Een probleemmodellering in een algemene programmeertaal is vaak langdradiger dan in een taal met syntax speciaal ontwikkeld voor dit doel. Dit doet dus afbreuk aan de declarativiteit.
- Een stand-alone tool is vaak moeilijker te incorporeren in andere computerprogramma's.
- Bij een implementatie in een algemene taal is het voor de ontwikkelaars vaak een niet te onderschatten inspanning om built-in zoekheuristieken te voorzien, terwijl het design van stand-alone tools hier vaak ook niet op voorzien is [20].

Modulaire zoekheuristieken proberen hieraan tegemoet te komen door het voorzien van bouwblokken die op eenvoudige wijze met elkaar kunnen gecombineerd worden. In de volgende sectie leggen we enkele minder bekende concepten uit Scala uit. Deze zullen ons toelaten gemeenschappelijk gedrag van de blokken te hergebruiken, blokken te definiëren in termen van reeds bestaande blokken en ze te combineren op een compacte en elegante manier zonder daar zelf speciale mechanismen voor te moeten voorzien.

2.2 Scala: functionele aspecten en traits

Scala is een relatief nieuwe objectgeoriënteerde programmeertaal voor de Java virtuele machine met invloeden van het functioneel programmeerparadigma. Scala is statisch getypeerd maar heeft *type-inferentie*, wat meteen zorgt voor een belangrijke codereductie in vergelijking met een equivalent Javaprogramma. In de volgende subsecties gaan we in op de verschillende aspecten van Scala die minder bekend zijn bij een publiek dat hoofdzakelijk vertrouwd is met Java.

2.2.1 Basisbegrippen

In deze sectie geven we een overzicht met absolute basiskennis van Scala. We vertrekken vanuit het standpunt van de ervaren Java programmeur. Na dit overzicht is het voor de lezer eenvoudiger de codefragmenten in het vervolg van deze thesis te begrijpen.

Variabelen Variabelen worden aangemaakt met het keywoord **var**. In de meeste gevallen is geen typeaanduiding nodig, omdat Scala type-inferentie heeft. Als er wel een aanduiding is, moet deze na de variabelenaam komen, zoals in Pascal. We kunnen dus het volgende schrijven:

var myCar = **new** CitroenDeuxChevaux(1980,"Bob","red")
val myCar: CitroenDeuxChevaux = **new** CitroenDeuxChevaux(1980,"Bob","red")

Een typeaanduiding is wel verplicht als de variabele een algemener type moet hebben dan dat van het object.

Scala kent naast **var** ook het keywoord **val**. Dat keywoord dient om final variabelen aan te geven, variabelen zoals je die in meestal functionele programmeertalen aantreft. Je kunt een dergelijke variabele dus maar één keer een waarde toekennen:

```
scala> val foo = 5
foo: Int = 5

scala> foo = 6
<console>:8: error: reassignment to val
```

Omdat Scala een functionele stijl aanmoedigt, komen zo'n variabelen in goede code veel meer voor dan een Java-programmeur zou vermoeden. Als je zelf in Scala begint te programmeren en een goede stijl wil ontwikkelen, denk dan dubbel na bij elke **var** die je schrijft.

Functies, procedures en methodes In een interactieve Scala sessie kun je functies definiëren zonder dat ze in een klasse voorkomen. De syntax is voor functies is als volgt:

def naam(param1: Type1, param2: Type2, ...): ReturnType = { Body }

Het returntype mag weggelaten worden als het afgeleid kan worden. Dat is meestal het geval: de voornaamste uitzonderingen zijn recursieve functies. Daarentegen is het returntype nuttige documentatie en ik vermeld het daarom vaak toch.

In Scala is een procedure een functie met returntype Unit. Dit type heeft maar één waarde, () (lees als „unit"). In de code schrijf je onmiddellijk het gelijkheidsteken na het haakje dat de parameterlijst afsluit, dus:

```
def helloWorld() = {
  println("Hello world!")
}
```

Het bovenstaande „Hello world" voorbeeld kan nog beter: bij procedures mag je het gelijkheidsteken weglaten. Eventueel kun je ook de lege haakjes weglaten, maar dat doe je beter niet omwille van een conventie bij methodes.

Uit het voorbeeld blijkt ook dat puntkomma's op het einde van een statement meestal niet verplicht zijn. De goede stijl is om ze weg te laten; slechts in enkele bijzondere gevallen zijn ze toch nodig.

Zowel bij procedures als functies mag je de accolades weglaten als de body slechts uit één statement bestaat. De types van de parameters mag je daarentegen *nooit* weglaten.

Methodes zijn procedures of functies die deel uitmaken van een klasse. De syntax is analoog aan die van functies en procedures, maar de conventie zegt dat je de lege haakjes van een parameterloze methode weglaat enkel en alleen als je de state van het object waarop de methode wordt opgeroepen niet wijzigt. Dit ondersteunt het *uniform access principle* [14]: je ziet niet of iets nu geïmplementeerd is als een veld of als een methode. Laat ons bijvoorbeeld een rechthoek modelleren (Codefragment 2.1).

Codefragment 2.1: Modellering van een rechthoek.

```
1 class Rectangle(val width: Int, val height: Int) {
    def area = width * height
3 }
```

Maar er kon net zo goed **val** area = ... staan. Het enige verschil zit in de tradeoff uitvoeringstijd versus geheugengebruik.

Klassenparameters Scala kent geen constructors zoals Java die kent. In de plaats daarvan zijn er klassenparameters: men zet de parameters die men nodig heeft om een object te construeren meteen na de naam van de klasse, tussen ronde haakjes. Als er geen parameters zijn, dan schrijf je ook geen haakjes. De parameters kunnen overal in de klasse gebruikt worden. Dus klassenparameters zijn net als private velden in Java.

Wil men er meteen ook een accessormethode (in Javatermen een „getter") bij, dan vermeldt men **val**. Wil men zowel een accessor als een mutator (of „setter"), dan schrijft men **var**. In

Codefragment 2.1 kan men dus ook van buiten de klasse aan de breedte en hoogte, maar deze eigenschappen kunnen niet gewijzigd worden.

Als men bijkomende operaties moet doen in de mutatormethode (of accessormethode), bijvoorbeeld controles op het uur van een klasse die een digitale klok modelleert, kan men deze ook zelf schrijven. Dat valt echter buiten het bestek van deze introductie.

Alle code in een klassendefinitie die niet in een methode voorkomt behoort tot de initialisatie van de klasse. Ook daar kan men de twee soorten variabelen definiëren. Wil je vermijden dat deze extra velden zichtbaar zijn buiten de klasse, dan zet je er **private** voor. Codefragment 2.2 vat dit samen.

Codefragment 2.2: Overzicht van de mogelijkheden voor velden.

```
class Summary(invisibleParam: Int, val visibleParam: Int, var: mutableParam) {
  val extraVisibleField = 1
  private val extraInvisibleField = 2
  var extraExternallyMutableField = 3
  private var extraPrivateMutableField = 4
  println("done constructing object!")

  /* ... methods ... */
}
```

Methodeoproepen Het oproepen van een methode gebeurt net als in Java. Men kan echter ook een operatorsyntax gebruiken, die vooral handig is als er exact één argument is, omdat dit heel natuurlijk overkomt. Dit werkt *niet* met functies of procedures! Enkele voorbeelden met een tekenreeks:

```
val hello = "Hello world"
hello reverse
hello substring 1 // Most natural
hello substring (1)
hello substring (1,2)
```

Een dergelijke manier van oproepen zal van pas komen bij het bouwen van een domeinspecifieke taal.

2.2.2 Singleton en companion objecten

Het objectsysteem van Scala verschilt lichtjes van dat van Java:

- Er is geen aparte constructie voor interfaces. Met een *trait* kan men hetzelfde effect bereiken.
- Scala kent de notie *singleton object*. Dat is een uniek object, waarbij we met „uniek" bedoelen dat het het enige object is van die soort. Technisch gesproken definieert een singleton object geen type, daarom zeggen we niet dat een singleton object het enige object is van dat type.
- In Scala zijn er geen statische methoden of variabelen.
- Een *companion object* kan gedefinieerd worden bij elke klasse.

De volgende sectie gaat uitgebreid in op traits omdat ze een belangrijk instrument en studieobject zijn in deze thesis. In deze sectie bespreken we de andere verschillen.

De definitie van een singleton object kan net als een klasse methodes en velden bevatten. In tegenstelling tot een klasse is er automatisch één en slechts één instantie. Het is dus niet langer nodig om een klasse te definiëren en via het design pattern „singleton" ervoor te zorgen dat er slechts één instantie van die klasse kan worden gemaakt. De syntax is analoog aan de manier waarop men een klasse definieert. Laat ons bijvoorbeeld een deel van een eenvoudig online reservatiesysteem voor hotelkamers schrijven (Codefragment 2.3).

We willen niet dat een kamer twee maal geboekt wordt, dus zorgen we ervoor dat de beschikbaarheid door één enkel object wordt bijgehouden[1]. Om het voorbeeld eenvoudig te houden, zorgen we ervoor dat telkens het volgende kamernummer uit een wachtrij wordt teruggegeven. Daarnaast laten we toe een kamer te annuleren, waarna het bijhorende nummer weer achteraan de rij wordt geplaatst. Let goed op enkele andere syntactische verschillen met Java.

Codefragment 2.3: Onderdeel van een eenvoudig hotelreservatiesysteem.

```
object RoomManager {
val availableRooms = new Queue[Int]()
availableRooms ++= 1 to 20

def getRoom(): Option[Int] = {
  try {
    Some(availableRooms.dequeue)
  } catch {
    case _ ⇒ None
  }
}

def cancelRoom(roomNumber: Int) = {
  if (availableRooms contains roomNumber)
    throw new IllegalArgumentException("Room not booked, so cannot cancel.")
  if (roomNumber <= 0 || roomNumber > 20)
    throw new IllegalArgumentException("Invalid room number.")
  println("Cancelling... done")
  availableRooms enqueue roomNumber
}

}
```

Als er geen kamers meer beschikbaar zijn, krijgen we een None, anders een Just(x) waarbij x het kamernummer is:

```
scala> RoomManager.getRoom
res0: Option[Int] = Some(1)
scala> RoomManager.getRoom
res1: Option[Int] = Some(2)
scala> RoomManager cancelRoom 2
Cancelling... done
scala> RoomManager cancelRoom 0
java.lang.IllegalArgumentException: Invalid room number.
...
```

Het is mogelijk voor singleton objecten om traits in te mixen, maar in tegenstelling tot klassen kunnen ze geen *klassenparameters* hebben.

Een companion object is ook een singleton object, maar het heeft dezelfde naam als een klasse en is bovendien in hetzelfde bestand gedefinieerd. Een companion object heeft toegang tot de pri-

[1] Omwille van schaalbaarheid is dit in de praktijk misschien geen goed idee.

vate leden van zijn geassocieerde klasse en omgekeerd. Het is de plaats om methodes te definiëren die je in Java statisch zou maken, of om constanten te definiëren. Veronderstel bijvoorbeeld dat we een klasse CitroenDeuxChevaux hebben, die de bekende Citroën 2CV modelleert, dan zouden we algemene zaken als topsnelheid of paardenkracht kunnen definiëren in het companion object (Codefragment 2.4).

Codefragment 2.4: Voorbeeld van companion object: Citroën 2CV

```
object CitroenDeuxChevaux {
val TOP_SPEED_KM_PER_H = 120
val FORWARD_GEARS = 4
val HORSEPOWER = 2
}
class CitroenDeuxChevaux(val constructionYear: Int, val owner: String, val color: String) {
  def driveAt(speed: Int) {
    if (speed > CitroenDeuxChevaux.TOP_SPEED_KM_PER_H) {
      println("Cannot drive this fast... I'm a 2CV!")
    } else {
      println("Please release clutch and press gass pedal to the floor!")
    }
  }
}

val bobsCar = new CitroenDeuxChevaux(1980,"Bob","red")
println(bobsCar.owner + "'s car has " + CitroenDeuxChevaux.FORWARD_GEARS
        + " forward manual gears.")
```

Een belangrijke toepassing van companion objecten is het schrijven van factorymethodes. Veronderstel dat de constructor van een klasse de interne werking van die klasse reflecteert en dus niet geschikt is voor gebruik elders, dan kan men die constructor privaat maken en een factory methode schrijven. Scala voorziet speciale syntax waardoor deze methodes ook een aantrekkelijk alternatief vormen voor hulpconstructors.

Het speciale zit in de methodenaam: als je de factorymethode van object `X` de naam `apply` geeft, dan kun je een instantie maken door de argumenten van `apply` te vermelden na de klassennaam, en zonder `new`-sleutelwoord. Een dergelijke techniek heeft men toegepast bij veel collecties uit de standaardbibliotheek, waardoor deze in de taal lijken te zijn ingebakken. Zo kun je een lijst aanmaken op de volgende manieren:

```
val a = List(1,2,3)
val b = List.apply(1,2,3)
```

De code van de eerste regel wordt automatisch geëxpandeerd tot de vorm zoals op de tweede regel. In het geval van lijsten is er een klasse `List` met het bijhorend companion object. Het is echter niet mogelijk de constructor van de klasse te gebruiken om een lijst aan te maken omdat de klasse abstract is. Hoe de `apply`-methode precies geïmplementeerd is doet niet ter zake. Het is echter belangrijk om te weten dat hier bijvoorbeeld een subtype kan aangemaakt worden. In onze code komt dit mechanisme van pas bij solveronafhankelijkheid (Sectie 3.9).

2.2.3 Overerving met traits

Scala kent enkelvoudige overerving, maar heeft daarnaast een krachtig *traitmechanisme* gebaseerd op *mixins*. Vaak heeft men één of meerdere methodes die men in verschillende contexten kan hergebruiken. Een superklasse is echter niet altijd geschikt als plaats om eenheden van hergebruik te implementeren omdat een klasse een semantische eenheid is [17]. De klasse heeft een primaire rol als generator van instanties en heeft daarvoor een unieke plaats nodig in de

klassenhiërchie. Een eenheid van hergebruik moet niet alleen klein zijn, maar ook op willekeurige plaatsen toepasbaar [23]. Een klasse voldoet aan geen van beide voorwaarden.

Een trait is een fundamentele eenheid van codehergebruik. Hij encapsuleert methode- en velddefinities [16]. Een codevoorbeeld is te zien in Codefragment 2.5. Strategieën voor het selecteren van een variabele in ons constraint solving systeem kunnen door middel van traits gemeenschappelijk worden gedefinieerd voor gehele en Boolse variabelen. Een dergelijke strategie is alleen nuttig voor arrays van variabelen. In onze implementatie moet de functie `v` opgeroepen worden voor elke variabele in de array. De variabele die de laagste waarde oplevert, moet worden geselecteerd. Hierbij maken we ook gebruik van abstracte types met een bovengrens. Dat is niet verplicht, maar het is vaak een handige combinatie. Hier laat ze ons toe de strategieën onafhankelijk te definiëren van het type variabele en omgekeerd, en dit slechts later te verfijnen. We nemen bijvoorbeeld de klasse `InputOrder` en bepalen een deel van zijn gedrag (of, in dit geval, het gehele gedrag) door het inmixen van de traits `IntVarSelector` (typebepalend) en het strategiebepalende `VariableInputOrder`. Dan is het duidelijk dat `InputOrder` enkel voor gehele domeinvariabelen werkt. Ook elke andere strategie kan gewoon ingemixt worden.

Codefragment 2.5: Voorbeeld van het gebruik van traits.

```
protected trait VariableSelector {
  type VariableType <: DomainVariable
  def v(variable: VariableType): Double
}
trait IntVarSelector extends VariableSelector {
  type VariableType = IntVar
}
trait BoolVarSelector extends VariableSelector {
  type VariableType = BoolVar
}

protected trait VariableInputOrder extends VariableSelector {
  def v(variable: VariableType) = 0
}
case class InputOrder() extends IntVarSelector with VariableInputOrder
case class BoolInputOrder() extends BoolVarSelector with VariableInputOrder
```

Wanneer men meerdere traits tegelijk gaat inmixen is de volgorde belangrijk. Methodeoproepen worden gelineariseerd. De manier waarop die linearisatie gebeurt is vrij complex, maar voor deze thesis volstaat het te weten dat traits die meer naar rechts voorkomen in de broncode eerst effect hebben. De linearisatie maakt duidelijk dat er steeds maar één superklasse is. Traits vermijden dus heel wat problemen die voorkomen in talen met meervoudige overerving [23]: dergelijke overerving

- zorgt voor implementatieproblemen.
- leidt tot conceptuele problemen omdat de klasse gebruikt wordt als generator van instanties en eenheid van codehergebruik.
- is niet wijd geaccepteerd.

2.2.4 Case classes

Van bijzonder nut in deze thesis zijn de zogenaamde *case classes*. Hun doel is tweevoudig: enerzijds gebruiken we ze als lichtgewicht object waarop men kan *pattern matchen*, anderzijds maken ze deel uit van de ondersteuning voor *domeinspecifieke talen* (DSL's). Het eerste gebruik

is duidelijk geïnspireerd door de algebraïsche datatypes uit Haskell (of gelijkaardige concepten in andere functionele talen, bijvoorbeeld records uit de taal Oz 3).

Functionele talen hebben vaak een eenvoudige syntax voor pattern matching in combinatie met recordachtige structuren. Ook bij case classes is pattern matching eenvoudig. We geven een codevoorbeeld uit de connectie tussen de modulaire zoekheuristieken en de constraint solver JaCoP, waarover meer details in Hoofdstuk 3 op pagina 25. Het idee hier is dat we zelf solveronafhankelijke constraints voorzien waarop we kunnen pattern matchen om ze om te zetten naar objecten die specifiek zijn voor de constraint solver in kwestie (Codefragment 2.6).

Codefragment 2.6: Case classes als lichtgewicht objecten voor pattern matching.

```
/* From the solver independent code */

/** Supertype of all constraints **/
sealed abstract class Constraint
case class XeqC(x: DomainVariable, c: Int) extends Constraint
case class XeqY(x: DomainVariable, y: DomainVariable) extends Constraint
case class XltC(x: DomainVariable, c: Int) extends Constraint
case class XltY(x: DomainVariable, y: DomainVariable) extends Constraint
/* ... more cases ... */

/* From the JaCoP specific code */

def solverDependentConstraint(solverIndependent: Constraint) = solverIndependent match {
  case XeqC(x,c)  ⇒ new JC.XeqC(x./* ... */,c)
  case XeqY(x,y)  ⇒ new JC.XeqY(x./* ... */,y./* ... */)
  case XltC(x,c)  ⇒ new JC.XltC(x./* ... */,c)
  case XltY(x,y)  ⇒ new JC.XltY(x./* ... */,y./* ... */)
  /* ... more cases ... */
}
```

Het tweede gebruik van case classes komt veelvuldig van pas: een case class biedt ondersteuning voor ingebedde domeinspecifieke talen doordat het mogelijk is een instantie aan te maken van die klasse zonder het `new` keyword te gebruiken. Dat vermindert de visuele overhead: een afgeleide heuristiek ziet er dan uit als in Codefragment 2.7. Dankzij dit type klasse is het dus mogelijk heuristieken samen te stellen op een meer declaratieve manier. Merk op dat we in het voorbeeld ook nog syntactische suiker hebben gebruikt om && te kunnen schrijven.

Codefragment 2.7: Case classes ondersteunen DSL's.

```
/** Factory function for sWhile combinator. **/
def SWhile(cond: VarCond, sh1: SearchHeuristic, sh2: SearchHeuristic): SearchHeuristic = {
  val letVar = LetVar("_swhile")
  Let(letVar,1,
    If((letVar == 1) && cond,sh1,
      And(Assign("_swhile_stage0",0),sh2)))
}
```

2.2.5 Symbolische operatoren en impliciete conversies

We kunnen verder gaan in voorzien van een DSL dan het weglaten van `new`-sleutelwoorden aan de hand van case classes. Scala laat namelijk ook toe symbolische operatoren te gebruiken (sommige infix en andere prefix). We kunnen functies met een symbolische naam schrijven die uit hun argumenten een nieuwe heuristiek bouwen. Bovendien kunnen ook methodes van een object een symbolische naam hebben.

Definitie

Functies die niet behoren tot een object kunnen niet zomaar ergens in een bestand voor de compiler staan. De interactieve interpreter laat dit wel toe (ook in scripts). Ze moeten gedefinieerd worden in het *package object*. Dat geldt ook voor functies met een symbolische naam. De definitie moet dus als volgt:

```
package object myTestPackage {
  def and(a: SearchHeuristic, b: SearchHeuristic) = {
    SeqAnd(a,b)
  }
  def &(a: SearchHeuristic, b: SearchHeuristic) = {
    SeqAnd(a,b)
  }
}
```

Deze functies kunnen opgeroepen worden als and(a,b) en &(a,b) respectievelijk. Een infixnotatie is op die manier niet mogelijk.

Om een voorbeeld te geven van een methode met een symbolische naam, ontwikkelen we een klasse voor het representeren van complexe getallen (Codefragment 2.8).

Codefragment 2.8: Klasse voor complexe getallen.

```
case class ComplexNumber(real: Int, imaginary: Int) {
  def +(other: ComplexNumber): ComplexNumber = { /* ... */ }
  def +(other: Int): ComplexNumber = {
    ComplexNumber(this.real + other, this.imaginary)
  }
}
```

Beperkingen

Een nadeel in de ondersteuning van symbolische operatoren is wel dat de precedentie van de symbolen vastligt. Wanneer we onze symbolen kiezen op een manier die een hint geeft over hun betekenis, hebben we helaas nogal wat haakjes nodig. In Codefragment 2.9 geven we code die equivalent is met die uit Codefragment 2.7. We hebben hier opzettelijk overdreven om duidelijk te maken dat overmatig gebruik van syntactische suiker het geheel erg cryptisch kan maken.

Codefragment 2.9: Overmatig gebruik van symbolische operatoren.

```
/** Factory function for sWhile combinator. **/
def SWhile(cond: VarCond, sh1: SearchHeuristic, sh2: SearchHeuristic): SearchHeuristic = {
  val letVar = LetVar("_swhile")
  Let(letVar,1,(( letVar == 1) && cond)?
    (sh1 <> ((letVar := 0) & sh2)) )
}
```

Een tweede beperking van Scala is dat wanneer men een operator infix gebruikt, deze altijd beschouwd wordt als een methode van de linkeroperand [16]. Een voorbeeld maakt dit duidelijk. Beschouw opnieuw de klasse `ComplexNumber`. Mits de methode + die als argument een geheel getal aanvaardt, gedefinieerd is, kunnen we de volgende code schrijven:

```
val b = a + 1
val c = 1 + a // Problem
```

De eerste regel is equivalent met a.+(1). De tweede regel is problematisch: men verwacht dat + commutatief is en men dus net zo goed 1 + a mag schrijven. Dat vereist dat de klasse `Int` een methode + bevat met als signatuur **def** +(b: ComplexNumber): ComplexNumber. De bestaande klasse `Int` kun je echter in tegenstelling tot de zelfgedefinieerde klasse `ComplexNumber` niet uitbreiden.

De taal laat toe dit op te lossen door het gebruik van impliciete conversies. Veronderstel dat de volgende code zichtbaar is in de huidige scope, dan leidt de regel die eerder een compilatiefout veroorzaakte, niet langer tot problemen:

implicit def intToComplex(x: Int) = **new** Complex(x,0)

Achter de schermen wordt 1 + a door de conversie omgevormd tot **new** ComplexNumber(1,0) + a. Na de impliciete conversie van een geheel getal naar een complex getal kunnen opnieuw de bewerkingen voor complexe getallen gebruikt worden. Impliciete conversies moeten echter voorzichtig gebruikt worden, en er zijn ook beperkende regels voor die echter niet van belang zijn voor het verder verloop van deze thesis.

Hoewel impliciete conversies duidelijk een verbetering zijn ten opzichte van de situatie in Java, kunnen we toch situaties bedenken waarin we, voor de onderhoudbaarheid van de code, onze operator liever niet implementeren als methode op het linkerobject. We zouden liever iets kunnen schrijven als — ongeldige — code die volgt:

```
class ComplexNumberOperators {
  def ++(a: ComplexNumber, b: ComplexNumber) = {
    new ComplexNumber(a.real+b.real, a.imaginary+b.imaginary)
  }
  def test() {
    val a= ComplexNumber(1,1)
    val b = ComplexNumber(2,2)
    val c = a ++ b // Wrong
  }
}
```

Deze code, hoewel het zeer verleidelijk is om iets dergelijks te schrijven, werkt niet, en zal geïnterpreteerd worden als a.++(b). Ook wanneer men de functie ++ zou definiëren in het *package object* — of op topniveau in de interactieve Scala-interpreter, wat jammer genoeg niet toegelaten is in codebestanden — kan men die operator niet infix gebruiken. Zelfs met prefix moet men opletten welk symbool men gebruikt. Beschouw namelijk de volgende situatie in de interactieve interpreter:

```
scala> case class ComplexNumber(real: Int, imaginary: Int)
defined class ComplexNumber

scala> def +(a: ComplexNumber, b: ComplexNumber) = {
     |   ComplexNumber(a.real + b.real, a.imaginary + b.imaginary)
     | }
$plus: (a: ComplexNumber, b: ComplexNumber)ComplexNumber

scala> val a = ComplexNumber(1,1)
a: ComplexNumber = ComplexNumber(1,1)

scala> val b = ComplexNumber(2,2)
b: ComplexNumber = ComplexNumber(2,2)
```

```
scala> +(a,b)
<console>:12: error: value unary_+ is not a member of
                     (ComplexNumber, ComplexNumber)

scala> $plus(a,b)
res1: ComplexNumber = ComplexNumber(3,3)

scala> def ++(a: ComplexNumber, b: ComplexNumber) = {
     |   ComplexNumber(a.real + b.real, a.imaginary + b.imaginary)
     | }
$plus$plus: (a: ComplexNumber, b: ComplexNumber)ComplexNumber

scala> ++(a,b) // This is OK
res2: ComplexNumber = ComplexNumber(3,3)

scala> $plus$plus(a,b)
res3: ComplexNumber = ComplexNumber(3,3)
```

Dit inconsistente gedrag wijst op twee zaken: ten eerste is het duidelijk dat het waarschijnlijk niet echt de bedoeling is om top-level functies een symbolische naam te geven en ten tweede dat de ondersteuning voor DSL's nog heel wat beter kan. De reden dat + niet werkt en ++ wel, is dat één enkel plusteken behoort tot het kleine groepje symbolen dat ook prefix kan gebruikt worden.

2.2.6 Eersteklassefuncties

Typisch voor functionele programmeertalen zijn zogenaamde hogere-orde functies. Dit zijn functies die andere functies als argument toelaten. Een functie zoals in imperatieve programmeertalen bekend is, is dan een functie van eerste orde. Men kan dit enigzins vergelijken met functiepointers in C of beter, wat ze hadden moeten zijn. Hoewel dit voor de lezer met een achtergrond in objectgeoriënteerd programmeren een eigenaardig concept kan lijken, is het eigenlijk heel natuurlijk en bijzonder krachtig. Sterker nog: objectgeoriënteerd programmeren is onder de motorkap massaal van hogere orde [25].

In de context van Scala spreekt men ook wel over eersteklassefuncties. De reden hiervoor is dat overal waar andere types toegelaten zijn, kan je ook functietypes hebben, bijvoorbeeld als parameter, returntype, of als veld. Met andere woorden, functies zijn eersterangsburgers in de taal.

Hogere-orde functies spelen een rol bij het verminderen van codeduplicatie. Ze laten je namelijk, net als een „gewone" functie, toe de gemeenschappelijke delen van het algoritme eenmalig te definiëren en de specifieke delen mee te geven als parameter. Op die manier realiseert men een controleabstractie. Het lijkt dan alsof de functie in de taal is ingebakken. Men kan dus verder abstraheren dan met eerste-orde functies.

Een traditioneel, doch eenvoudig, voorbeeld is het sorteren van een lijst van objecten. Veronderstellen we bijvoorbeeld een lijst namen. Men kan die lijst in alfabetische volgorde sorteren, omgekeerd alfabetisch, op voornaam of achternaam, of elke andere manier die men maar kan bedenken. Een naïve implementatie is om de code te dupliceren. Het eigenlijke idee achter een sorteeralgoritme is namelijk altijd hetzelfde: „als naam x voor naam y hoort, zet hem er dan voor". Het enige wat verschilt is de manier waarop je beslist dat naam x voor naam y hoort, met andere woorden het sorteercriterium. Je wilt dit dus als parameter aan de sorteerfunctie kunnen meegeven. Maar wat doet dit criterium precies? Het neemt twee namen en geeft terug of de

eerste voor de tweede moet komen. Met andere woorden, het criterium neemt zelf parameters, en is dus een functie. In Java kan men dat effect emuleren door die functie te verpakken in een object dat men `Comparator` heet. Dat is niet meer dan een kunstgreep. In C is dit het klassieke voorbeeld voor een functiepointer [12]. Hogere-orde functies gaan echter veel verder dan functiepointers[2].

In functionele programmeertalen werkt men graag met lijsten. Hoewel multi-paradigma heeft Lisp, wat staat voor „list processing", een sterk functionele inslag. Er zijn dan ook heel wat controleabstracties voor het werken met lijsten ontwikkeld. Het zijn stuk voor stuk hogere-orde functies en ik omschrijf ze wel eens als het Zwitsers zakmes van de functionele programmeur. Ook in Scala zijn dit prima voorbeelden. In de volgende paragrafen bespreken we er slechts twee ter illustratie.

Map De *map*-functie roept een door de gebruiker mee te geven functie op voor elk element uit een lijst en verzamelt de resultaten in een nieuwe lijst. Het verhogen van elk getal in een lijst met 10 gaat dan in Scala als volgt:

```
scala> List(1,2,3) map (_ + 10)
res0: List[Int] = List(11,12,13)
```

Dit is meteen een voorbeeld met een *anonieme functie*. We kunnen dit ook uitschrijven, wat duidelijk maakt dat de functie *map* wel degelijk van tweede orde is.

```
def plusTen(number: Int) = number + 10
val initialList = List(1,2,3)
initialList.map(plusTen)
```

Filter De *filter*-functie neemt een lijst (in Scala is dit het object waarop je de functie oproept) en een *predicaat* en weerhoudt die elementen uit de lijst die niet aan het predicaat voldoen. Om bijvoorbeeld de negatieve getallen uit een lijst weg te laten, schrijft men in Scala:

```
scala> List(1,-1,2,-2) filter (_ > 0)
res0: List[Int] = List(1,2)
```

Als we dit voluit schrijven, ziet de code er zo uit:

```
def strictlyPositive (number: Int) = number > 0
val initialList = List(1,−1,2,−2)
initialList . filter ( strictlyPositive )
```

Ook voor tal van andere datastructuren maken dergelijke hogere-orde functies deel uit van de standaardbibliotheek voor collecties in Scala. Die bibliotheek komt ook in onze thesis zeer goed van pas. Daarnaast schrijven we ook zelf tal van hogere-orde functies, juist omdat ze controleabstracties realiseren. Meestal doen we dit zelf onbewust.

2.2.7 Gecurryde functies

Gecurryde functies (naar de logicus Haskell B. Curry) zijn een belangrijk concept in functionele programmeertalen. In sommige talen, waaronder Haskell, zijn gecurryde functies zelfs standaard. Ze liggen aan de basis van het definiëren van nieuwe controleabstracties en Scala gaat hier vrij ver in. Aan het einde van deze sectie zullen we in staat zijn abstracties te ontwerpen die eruit zien alsof ze in de taal zijn ingebakken.

[2]Voor de geïnteresseerden volstaat de sectie over *closures* in een willekeurig boek over functioneel programmeren, naast wat kennis over de layout van de stack in C-achtige talen.

Een gecurryde functie heeft in Scala meer dan één parameterlijst. Laat ons verderwerken aan het voorbeeld met de Citroën 2CV uit Sectie 2.2.2: veronderstel dat we voor een leerling-chauffeur het gepaste motorgeluid willen afspelen gegeven de versnelling waarin de auto zich bevindt en de snelheid waarmee hij rijdt (Codefragment 2.10). Voor de eenvoud laten we enkele versnellingen weg. We maken meteen twee parameterlijsten. Dat lijkt nu raar, maar straks kunnen we daar enkele leuke voorbeelden mee geven.

Codefragment 2.10: Voorbeeld voor currying: de leerling-chauffeur

```
def playMotorSound(gear: Int)(currentSpeed: Int) {
  (gear,currentSpeed) match {
    // Leave out neutral, reverse and 3rd gear for simplicity
      case (1,x) if x <= 0 ⇒println("IIII...")
      case (1,x) if x < 10 ⇒ println("rrrr")
      case (1,_) ⇒ println("WROAR")

      case (2,x) if x < 10 ⇒ println("splut splut...")
      case (2,x) if x > 40 ⇒ println("WROAR")
      case (2,_) ⇒ println("rrrr")

      case (_,x) if x < 60 ⇒ println("splut splut...")
      case (_,x) if x >= 60 ⇒println("WROAR")
  }
}
```

Deze functie kan dus als volgt gebruikt worden:

```
scala> playMotorSound(4)(90)
WROAR
```

Veronderstel nu dat we gespecialiseerde functies willen voor elke versnelling, dan kunnen we deze eenvoudig schrijven:

```
def soundFirstGear = playMotorSound(1) _
def soundOverdriveGear = playMotorSound 4 _
```

Het type van beide functies is (Int) ⇒ (). Je kunt ze gebruiken net als elke andere functie:

```
scala> soundOverdriveGear(90)
WROAR
```

Veronderstel dat we nu ook een synoniem willen voor onze oorspronkelijke functie, dan kan o.a. met **val** drive = playMotorSound _. Wanneer we dit interactief uitvoeren zien we het type van `playMotorSound`: (Int) ⇒ (Int) ⇒ ().

Er zijn heel wat situaties waarin dit mechanisme nuttig kan gebruikt worden. Maar het kan nog beter: in Scala mag je de ronde haakjes van elke parameterlijst waarin exact één parameter voorkomt vervangen door accolades. Dus elk van volgende oproepen is ook toegelaten:

drive{4}{90}
drive{4}(90)
drive(4){90}

In de praktijk tref je meestal de laatste aan, maar dan ietsje anders gespatieerd: drive(4) { 90 }. Merk op dat dit heel hard gaat lijken op ingebouwde constructies als „if" of „while". Dat effect is nog groter als het tweede argument een *anonieme functie* is. Op die manier kun je zelf controleabstracties bouwen die net in Scala ingebakken lijken.

2.2.8 Flexibele modifiers en imports

Scala heeft meer flexibele modifiers dan Java. Er is echter geen modifier voor pakketzichtbaarheid (de impliciete default in Java): de afwezigheid van een modifiers betekent publieke toegang. Zichtbaarheid binnen een pakket `X` kan echter bekomen worden door de flexibiliteit. Voor pakkettoegang schrijf je dan `private[X]`. Andere combinaties met flexibiliteit zijn ook mogelijk. Ze zijn nuttig wanneer klassen of pakketten genest worden. Veronderstel dat we een klasse `B` definiëren in een klasse `A` dan zijn ondermeer ook de volgende combinaties geldig binnen klasse `B`:

private[A] **protected**[B] **private**[**this**]

Ook alle andere combinaties zijn geldig. Voor `private[this]` bestaat geen Java tegenhanger. De betekenis is toegang enkel binnen dit object; toegang vanuit een ander object van hetzelfde type is dus niet mogelijk.

Voor deze thesis zijn de flexibele imports van Scala op zich niet zo belangrijk. In Sectie 4.1 suggereren we echter een gelijkaardige syntax voor een uitbreiding van de traits. Voor de lezer die niet vertrouwd is met Scala , voorzien we dan ook een korte uitleg in Appendix B op pagina 55.

De lezer beschikt nu over alle nodige voorkennis van Scala om de meer technische hoofdstukken aan te pakken. Vooraleer we daarmee beginnen, geven we eerst aan hoe praktische problemen met ons systeem voor modulaire zoekheuristieken kunnen aangepakt worden.

2.3 Programmeren met modulaire zoekheuristieken

Ons systeem voor modulaire zoekheuristieken bestaat enerzijds uit de verzameling heuristieken zelf, anderzijds uit variabelen, een verzameling beperkingen en ondersteunende klassen. Bij de variabelen heeft men *domeinvariabelen* en *letvariabelen*. De domeinvariabelen zijn die variabelen waarvoor een oplossing moet worden gevonden. Letvariabelen ondersteunen het zoeken naar die variabelen. Ze kunnen bijvoorbeeld worden worden gebruikt als tellervariabele in een repeterende heuristiek, of voor het bijhouden van statistieken zoals de diepte van de boom of het aantal nodes. Domeinvariabelen zijn ofwel geheel of Bools. Ze kunnen zowel alleen als in arrays voorkomen. Voorbeelden zijn te vinden in Codefragment 2.11. De methodes die we illustreren zijn zowel bruikbaar voor gehele als Boolse variabelen (of arrays indien het een arraymethode betreft).

Codefragment 2.11: Het gebruik van domeinvariabelen en constraints.

```
1  /* Make a new integer domain variable with domain [1,10]. */
   val foo = IntVar(1,10)
3  /* Make another integer domain variable. */
   val foo2 = IntVar(5,10)
5  /* Make a new boolean domain variable. */
   val bar = BoolVar()
7  /* Make a new array of three integer domain variables with domain [1,10]. */
   val arrFoo = IntVarArgs(3,1,10)
9  /* Make a new array of three boolean domain variables. */
   val arrBar = BoolVarArgs(3)
11
   /* Has this variable been assigned? */
13 val baz = bar.assigned
   /* Get the value if it's safe. */
15 if (baz)
       println("Bar = " + bar.value)
17 /* Get the lower and upper bound. */
```

```
  val lower = foo.lowerBound
19 val upper = foo.upperBound

21 /* Get second variable in the array. */
  val second = arrFoo(1)
23 /* Get array size. */
  val size = arrFoo.size
25
  /* Constraints */
27 val constraint1 = foo := 5
  val constraint2 = foo != foo2 // alternative: =/=
29 val constraint3 = foo < 5
  val constraint4 = foo >= foo2
```

Daarnaast geeft Codefragment 2.11 enkele voorbeelden van het aanmaken van constraints. De constraints die hier geïllustreerd worden, hebben allemaal een symbolische infixnotatie. Daardoor is hun betekenis intuïtief duidelijk. Ze kunnen ook voluit geschreven worden als volgt:

```
  val constraint1b = XeqC(foo,5)
2 val constraint2b = XneqY(foo,foo2)
  val constraint3b = XltC(foo,5)
4 val constraint4b = XgteqY(foo,foo2)
```

Veel meer binaire constraints zijn beschikbaar volgens hetzelfde benoemingspatroon, en hebben een symbolische notatie. Unaire en ternaire constraints hebben nooit een symbolische notatie. Door het aanmaken van een constraint wordt hij overigens nog niet afgedwongen: daarvoor is er de methode `imposeConstraint` van de klasse `Space`.

Een overzicht van de primitieve heuristieken is te vinden in Tabel 2.4 op pagina 21. Hun betekenis wordt gedefinieerd in [20]. Daarnaast zijn er in ons systeem heel wat voorbeelden van nuttige afgeleide heuristieken gedefinieerd. Hun implementatie gebeurt in functie van de primitieven. Merk op dat de letheuristiek niet-primitief werd geïmplementeerd, hoewel hij conceptueel wel als primitief is te beschouwen. Een implementatie als primitief is evenzeer mogelijk, en waarschijnlijk performanter, maar vereist meer code.

2.3.1 Symbolische notatie

Voor de combinatie van ingebouwde combinatoren tot complete heuristieken is het mogelijk een symbolische notatie te gebruiken. In Tabel 2.1 tonen we welke symbolen beschikbaar zijn en wat hun betekenis is. De precedentie van deze symbolen verloopt volgens de Scala standaardregels (Appendix A op pagina 54). Ook voor het opbouwen van condities zijn enkele symbolen gedefinieerd (Tabel 2.2).

Daarnaast zijn de gekende symbolen `<` (kleiner dan) en `>` (groter dan) beschikbaar voor vergelijking van een *letvariabele* en een getal, of een letvariabele en andere letvariabele. Voor een letvariabele en een getal zijn er ook nog kleiner-of-gelijk (`<=`), groter-of-gelijk (`>=`) en gelijkheid (`==`). Bij de vergelijking van een domeinvariabele met een letvariabele zijn ook alle gekende symbolen van toepassing.

Zoals hierboven reeds vermeld, hebben ook binaire constraints een symbolische notatie. We hebben deze opgenomen in de tabel met beschikbare constraints (Tabel 2.3). Hierbij staan de letters x en y voor een domeinvariabele, c voor een constante en staat `xs` voor een array van domeinvariabelen.

Tabel 2.1: Beschikbare syntactische suiker voor combinatoren en zijn betekenis.

Symbool	Combinator
`A & B`	`SeqAnd(A,B)`
`A | B`	`SeqOr(A,B)`
`!A`	`Once(A)`
`c ? A <> B`	`If(c,A,B)`
`x := i`	`IntAssign(x,i)`
`x := y`	`LetAssign(x,y)`
`x := d`	`DVarAssign(x,y)`
`+x`	`Incr(x)`
`%`	`Prune()`

Tabel 2.2: Beschikbare syntactische suiker voor condities en zijn betekenis.

Symbool	Conditie
`@@`	`TrueCond()`
	`FalseCond()`
`c1 && c2`	`AndCond(c1,c2)`

gekende vergelijkingsoperatoren

Tabel 2.3: Beschikbare constraints en eventuele symbolische notatie.

Type constraint	Constraint	Symbool
Binair x+c	XeqC(x, c)	`:=`
	XneqC(x, c)	`!=` of `=/=`
	XltC(x, c)	`<`
	XgtC(x, c)	`>`
	XlteqC(x, c)	`<=`
	XgteqC(x, c)	`>=`
Binair x+y	XeqY(x, y)	`:=`
	XneqY(x, y)	`!=` of `=/=`
	XltY(x, y)	`<`
	XgtY(x, y)	`>`
	XlteqY(x, y)	`<=`
	XgteqY(x, y)	`>=`
Unair	Alldiff(xs)	
Ternair	Distance(x, y, z)	

Tabel 2.4: Overzicht van de primitieve zoekheuristieken.

Naam	**Omschrijving**	**Syntax**
integer search	Systematisch zoeken over de gegeven integere variabelen `v` met de gegeven variabeleselectiestrategieën en waardeselectiestrategie.	`IntSearch(assign,v,varSelStrats,valSelStrat)`
boolean search	Systematisch zoeken over de gegeven Boolse variabelen `v` met de gegeven variabele- en waardeselectiestrategieën.	`BoolSearch(v,varSelStrat,valSelStrat)`
prune	Snoei de zoekboom onder de huidige node weg.	`Prune()`
fail	Faal de huidige zoekheuristiek.	`Fail()`
fail to prune	Zet in heuristiek `s` een faling om in een snoeioperatie.	`FailToPrune(s)`
integer assign	Assigneer integer `i` aan letvariabele `v`.	`IntAssign(v,i)`
let assign	Assigneer de waarde van letvariabele `v2` vermeerderd met de optionele waarde `o` aan variabele `v1`.	`LetAssign(v1,v2,o)`
domain var assign	Assigneer de waarde van domeinvariabele `dv` vermeerderd met de optionele waarde `o` aan letvariabele `lv`.	`DVarAssign(lv,dv,o)`
median assign	Assigneer de mediaan van de twee waarden bekomen uit letvariabelen `v2` en `v3` aan letvariabele `v1`.	`AssignMed(v1,v2,v3)`
multiply	Vermenigvuldig de waarde van letvariabele `v` met de opgegeven multiplier `m`.	`Mult(v,m)`
luby	Stel letvariabele `v1` gelijk aan `c*luby(a)` waarbij `a` de waarde is van `v2` (nummerreeks).	`Luby(v1,v2,c)`
print	Druk de waarde van variable of array `a` als `s` een oplossing vindt.	`Print(a,s)`
post	Leg conditie `c` op in elke node bekomen tijdens het uitvoeren van heuristiek `s`.	`Post(c,s)`
if	Als conditie `c` waar is, zoek volgens heuristiek `s1`, anders zoek volgens `s2`.	`If(c,s1,s2)`

Naam	**Omschrijving**	**Syntax**
binary and	Voer eerst heuristiek `s1` uit, in geval van succes voer `s2` uit, anders faal.	`SeqAnd(s1,s2)`
and	Voer eerst de eerste heuristiek in de lijst `l` uit, in geval van succes voer de tweede uit, enzovoort.	`SeqAnd(l)`
or	Voer eerst heuristiek `s1` uit, bij faling probeer `s2`.	`SeqOr(s1,s2)`
restart	Herstart zoekheuristiek `s` zolang conditie `c` waar is.	`Restart(c,s)`
let	Initialiseer de gegeven letvariabele `v` op de opgegeven waarde `i` en voer daarna heuristiek `s` uit.	`Let(v,i,s)`
depth count statistic	Houd tijdens het uitvoeren van heuristiek `s` de diepte van de boom bij in de opgegeven letvariabele `v`.	`DepthCount(v,s)`
node count statistic	Houd tijdens het uitvoeren van heuristiek `s` het aantal nodes in de boom bij in de opgegeven letvariabele `v`.	`NodeCount(v,s)`
fail count statistic	Houd tijdens het uitvoeren van heuristiek `s` het aantal falingen bij in de opgegeven letvariabele `v`.	`FailCount(v,s)`
discrepancy count statistic	Houd tijdens het uitvoeren van heuristiek `s` het aantal discrepanties bij in de opgegeven letvariabele `v`.	`DiscrCount(v,s)`

2.4 Voorbeeldprogramma's

Ter illustratie van de mogelijkheden, bespreken we het probleem van de Golomb linealen. Een Golomb lineaal is een verzameling posities op een imaginair lineaal zodanig dat de afstand tussen elke twee posities verschillend is [3, 4]. Zonder verlies aan algemeenheid kan men de eerste positie op nul kiezen. Het aantal markeringen noemen we de orde, de lengte is wat we er intuïtief onder verstaan. Een Golomb lineaal noemen we optimaal als er geen korter lineaal van dezelfde orde bestaat. Dit is een rekenintensief optimalisatieprobleem dat we kunnen oplossen met modulaire zoekheuristieken (Codefragment 2.12).

In de klasse `GolombRuler` maken we een model voor het probleem. Twee heuristieken om het probleem op te lossen aan de hand van het model staan in de objecten `GolombBab` en `GolombRestartBab`. Op regel 9 van de code maken we een array aan voor de markeringen op het lineaal, en op regel 11 voor de verschillen. Op regel 13 leggen we de beperking op dat de eerste markering nul is. De notatie met := is syntactische suiker voor de constraint `XltC`. In de lus van regel 15 zorgen we dat de markeringen geordend zijn. Opnieuw gebruiken we syntactische suiker: < komt veel beter over dan XltY(marks(i),marks(i + 1)). Vervolgens leggen we op dat differences (k) = marks(j) − marks(i).

Opleggen dat de variabelen in een array allen een verschillende waarde moeten aannemen komt vaak voor en kan met de `Alldiff` constraint (regel 27). Als kost nemen we de lengte van het lineaal, dus de plaats van de laatste markering. Op regels 45 en 50 maken we de twee heuristieken aan. Ze hebben echter zoveel gemeenschappelijk dat het nuttig is een trait te definiëren (regels 38 tot 44). Het sleutelwoord **lazy** bij print zorgt ervoor dat er geen defaultwaarde **null** wordt gebruikt voor searchHeuristic maar dat we wachten om deze variabele te initialiseren tot ze nodig is. Wanneer we een instantie aanmaken worden de variabelen van de ingemixte traits en superklasses namelijk eerst geïnitialiseerd, tenzij we voorzorgen treffen. Jammer genoeg wordt dit vaak vergeten. De reden waarom we het eigenlijke zoeken (regels 47 en 53) niet kunnen opnemen in de trait, is exact dezelfde.

Codefragment 2.12: Code voor het probleem van de Golomb linealen.

```
package searchCombinators.examples
import searchCombinators._
/* Example programs finding optimal Golomb rulers. */
/* Model for Golomb Ruler */
class GolombRuler(size: Int, space: Space) {
  // Avoid overflow
  assert(size < 32,"Size > 31 would lead to overflow")
  val guessUpperBound = (1 << (size+1)) - 1
  val marks = IntVarArgs(size,0,guessUpperBound,space)
  val numDifferences = (size*size - size)/2
  val differences = IntVarArgs(numDifferences,1,guessUpperBound,space)
  // Assume first mark to be zero
  space.imposeConstraint(marks(0) := 0)
  // Order marks
  for (i ← 0 until size-1) {
    space.imposeConstraint(marks(i) < marks(i+1))
  }
  // Setup difference constraints
  var k = 0
  0 until size-1 foreach { i ⇒
    i+1 until size foreach { j ⇒
      {space.imposeConstraint(Distance(marks(j),marks(i),differences(k)))
      k += 1
      }
```

```
    }
  }
  space.imposeConstraint(Alldiff( differences ))
  // Symmetry breaking
  if (size > 2) {
    space.imposeConstraint(differences(0) < differences(numDifferences-1))
  }

  def getCost(): IntVar = {
    marks(size-1)
  }
}
/** Common code for making example run */
trait GolombCommon {
  val space = Space()
  val model = new GolombRuler(3,space)
  def searchHeuristic: SearchHeuristic
  lazy val print = Print(searchHeuristic, model.getCost())
}
/** Finds optimal Golomb ruler using branch-and-bound. */
object GolombBab extends App with GolombCommon {
  val searchHeuristic = Bab(model.getCost(),IntSearch(model.marks,List(InputOrder()),BisectLow()))
  Runner.run(print,space)
}
/** Finds optimal Golomb ruler using restart branch-and-bound. */
object GolombRestartBab extends App with GolombCommon {
  val searchHeuristic = RestartBab(model.getCost(),IntSearch(model.marks,List(InputOrder()),
    BisectLow()))
  Runner.run(print,space)
}
```

Hoofdstuk 3

Technische uitwerking

In dit hoofdstuk bespreken we grondig de technische details van onze implementatie. Een implementatie van dergelijke omvang wordt niet in één stuk geschreven: in een eerste fase hebben we de bouwstenen waaruit zoekheuristieken bestaan, geïmplementeerd zonder de koppeling te maken met een constraint solver library. We hebben deze dus vervangen door een „dummy".

Aan het begin hebben we geprobeerd de aanpak van de bestaande code in C++ zo dicht mogelijk te benaderen in Scala. Op die manier maakten we ons niet alleen vertrouwd met de code en de nieuwe taal, maar het liet ons ook toe heel wat ineleganties aan het licht te brengen. Vervolgens hebben we het ontwerp aangepast naar iets wat niet alleen meer geschikt is voor Scala, maar ook zo goed mogelijk voldoet aan onze doelstellingen. Deze transformaties bespreken we in hetgeen volgt.

In een tweede fase hebben onze dummy constraint solver vervangen door een solveronafhankelijk systeem, en voorzagen we de koppeling met een specifieke constraint solver library, JaCoP.

3.1 Macrostructuur van het uiteindelijke systeem

Op het hoogste niveau bestaat de code uit een solveronafhankelijk deel, en een solverafhankelijk deel. Het solveronafhankelijk deel voorziet het volgende:

- de basisklassen waarop zoekheuristieken gebouwd worden,
- ten tweede een implementatie van primitieve zoekheuristieken waarvan de kracht zich slechts uit door ze te gaan combineren, en
- een implementatie van afgeleide heuristieken op basis van de vorige.

De basis waarop het gehele systeem rust, voorziet zelf geen implementatie van de typische zaken die een standaard constraint solver doet, maar voorziet wel een solveronafhankelijke interface. Het solverafhankelijk deel voorziet dan in die zaken die ontbreken. We voorzien in de basis bijvoorbeeld geen implementatie van constraint variabelen, maar definiëren wel een interface waaraan deze moeten voldoen.

De primitieve zoekheuristieken hebben nogal wat gemeenschappelijke kenmerken, waar we gebruik maken van traits om codeduplicatie te vermijden. Ze zijn dan ook gestructureerd zoals wordt afgebeeld in Figuur 3.1 op de volgende pagina. Aan de top van de structuur staat de abstracte klasse `SearchHeuristic`. Codefragment 3.1 op de pagina hierna toont de meest belangrijke methodes van deze klasse. Deze methodes vormen samen een *message protocol*. Dat protocol wordt in meer detail besproken in Sectie 3.2.

De figuur toont enkele primitieve heuristieken in het zwart. Alle primitieve heuristieken erven over van `SearchHeuristic` maar voor de overzichtelijkheid hebben we enkele (zwart afgebeelde) overervingspijlen weggelaten. De grijze pijlen met bolletje op het uiteinde duiden *traitinclusie* aan. Een primitieve heuristiek moet vanzelfsprekend het volledige message protocol ondersteunen. Maar daarvoor kan de heuristiek zoveel mogelijk traits gebruiken. Deze implementeren elk een stukje van het protocol, namelijk één of meerdere methodes. Elke trait is ook een partiële implementatie van `SearchHeuristic`. Dat duiden we aan met de grijze overervingspijlen.

Codefragment 3.1: De belangrijkste methodes van `SearchHeuristic`: de boodschappen uit een message protocol.

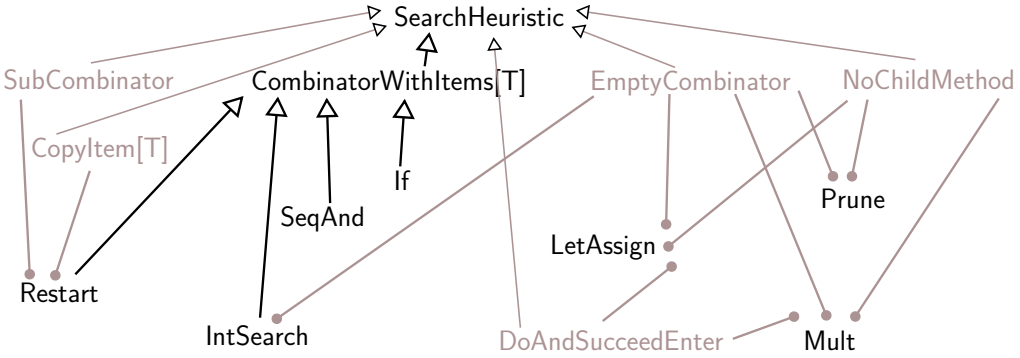
```
abstract class SearchHeuristic {

  /* ... Syntactic sugar and helper methods ... */

  def next(currentNode: Node, nextNode: Node)
  def enter(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator, top:
    SearchHeuristic, currentNode: Node): Int
  def child(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator, top:
    SearchHeuristic, currentNode: Node, childNumber: Int): Option[Node]
  def stat(node: Node)
  def init(node: Node, vm: VarAccessor#VM)
  def copy(oldNode: Node, newNode: Node)
  def steal(oldNode: Node, newNode: Node)
  def disposeNode(toDispose: Node) { /* Deliberately empty */ }
}
```

Afgeleide heuristieken ontstaan door de primitieve heuristieken te gaan combineren. Een belangrijke doelstelling van deze thesis is om dat proces zo gebruiksvriendelijk mogelijk te maken. We hebben er enkele voorgedefinieerd in het pakketobject van het pakket `searchCombinators`. Deze zijn algemeen bruikbaar. Bij het modelleren van een probleem maakt men ook afgeleide heuristieken, zij het nogal probleemspecifieke.

In de hieropvolgende secties gaan we dieper in de werking van het systeem en op de structuur van de combinatoren.



Figuur 3.1: Structuur van enkele primitieve zoekheuristieken.

3.2 Het message protocol en zijn implementatie

Zoekheuristieken kunnen met elkaar interageren met een message protocol besproken in [20]. Dit protocol zorgt voor modulariteit. De belangrijkste boodschappen zijn `enter` en `child`. In een praktische implementatie zijn echter meer boodschappen nodig, vooral voor boekhouding.

Als een combinator de boodschap `enter` ontvangt, voert hij zijn specifieke taak uit op de huidige node van de zoekboom. Die taken zijn van heel uiteenlopende aard: het incrementeren van een zekere variabele, het opleggen van een nieuwe constraint, of op basis van een test een subcombinator op een bepaalde manier oproepen zijn maar enkele voorbeelden.

De `child`-methode bepaalt hoeveel kindnodes de huidige node nodig heeft. Die kindnodes worden dan aangemaakt en bewaard om er later op hun beurt `enter` op op te roepen. Dat ogenblik hangt af van de manier waarop men de zoekboom wenst te doorlopen.

De `stat`-methode is er ter ondersteuning van combinatoren die een statistiek bijhouden. Veronderstel dat een statistische combinator in zijn `enter`-methode een waarde heeft bijgehouden waaruit de uiteindelijke statistiek kan berekend worden, dan kan die naverwerking gebeuren in de `stat`-methode.

De boodschap `init` laat een combinator toe zich te initialiseren voor hij de eerste `enter`-boodschap ontvangt. Een combinator kan hier bijvoorbeeld een boekhoudkundige teller op nul zetten of iets anders dat nodig is voor een correcte werking.

De methodes `copy` en `steal` zijn nodig als men parallel wil gaan zoeken. Onze code ondersteunt dit niet. De boodschap `disposeNode` signaleert dat de combinator alle boekhoudkundige informatie die hij eventueel zou bijhouden over de node in kwestie mag weggooien.

In onze Scala implementatie zijn we er in geslaagd de methode `itemSize` zoals die aanwezig is in de C++ code te elimineren. Deze methode stond in voor het bepalen van het aantal bytes nodig voor het bewaren van de „items” van de combinator in kwestie.

Wat een „item” precies is, is combinatorafhankelijk; in zijn meest algemene vorm kunnen we het omschrijven als „state die nodig is om de goede werking van de combinator te verzekeren, maar door de veronderstellingen die successief gemaakt worden bij constraint programming (en de daarbij horende backtracking) niet als gewone state kan geïmplementeerd worden”.

In Scala houdt elke combinator zijn eigen items bij, waardoor we niet alleen deze methode kunnen weglaten, maar vooral heel wat casts vermijden. In C++ werden deze items op een centrale plaats bijgehouden. Dat is gemakkelijker voor het manuele geheugenbeheer, maar daardoor verloren we wel de specifieke typeinformatie.

Het weglaten van de methode is belangrijk omdat de drempel om een nieuwe heuristiek te implementeren hierdoor verlaagt.

3.3 Indeling in pakketten

De meeste code behoort tot het pakket `searchCombinators`. De klassen `Node` en `NodeAllocator` vormen hierbij de voornaamste uitzondering. We willen dat een nieuwe instantie van `Node` slechts kan aangemaakt worden door `NodeAllocator`. Dat doen we door beide klassen in een afzonderlijk subpakket `node` te plaatsen, en daarna de constructor van `Node` privaat voor dat pakket te maken.

De voorbeeldproblemen bevinden zich in het pakket `searchCombinators.examples`. De koppeling met de JaCoP constraint solver behoort tot `searchCombinators.jacopConnection`.

3.4 Microstructuur van een combinator

De microstructuur van een combinator heeft rechtstreekse invloed op de moeilijkheidsgraad om een nieuwe combinator te schrijven. De kenmerken van een goede microstructuur zijn de volgende:

- we moeten zo weinig mogelijk methodes implementeren,
- de methodes zijn zo kort mogelijk,
- de methodes van een combinator bevatten onderling geen gelijkaardige code,
- code komt niet voor in meerdere combinatoren.

De eerste twee punten hebben voor een groot deel te maken met het gedetailleerd design van het systeem en de programmeertaal waarin men schrijft. We hebben gezien dat we door aanpassingen in het design methodes kunnen weglaten of eenvoudiger aan de benodigde data raken, maar ook dat dit vrij programmeertaalafhankelijk is. Uiteraard blijft de architectuur van het systeem op hoog niveau hetzelfde.

Vermijden dat methodes binnen één combinator gelijkaardige code bevatten kan door middel van controleabstracties. Verschillende manieren om deze te realiseren werden besproken in Hoofdstuk 2. We hebben deze technieken ook toegepast in onze implementatie. We bespreken dit in Sectie 3.5. Algemeen kan men stellen dat technieken uit functioneel programmeren hier het mechanisme bij uitstek zijn.

Het laatste kenmerk van goede microstructuur is dat code niet voorkomt in meerdere combinatoren. Dat is het meest opvallende probleem met de C++ code en we besteden er dan ook veel aandacht aan. De manier om dit op te lossen is overerving. We hebben reeds geargumenteerd dat door het grote aantal combinaties noch enkelvoudige noch meervoudige overerving geschikt zijn (Sectie 2.2.3). Methodes zijn kleine eenheden, dus gebruiken we een lichtgewicht eenheid van hergebruik: de trait.

Het gemakkelijke geval is wanneer de code in twee combinators exact hetzelfde realiseert. In dat geval stopt men de methode in een trait. Meestal hebben combinators meer dan één methode gemeenschappelijk zodat men grotere traits met meerdere methodes kan bouwen. Op die manier bekomen we voor ons systeem een volledige hiërarchie van traits, zoals reeds werd afgebeeld in Figuur 3.1.

In die figuur hebben we niet alle traits weergegeven. We hebben ook nog een aantal traits die zorgen voor een controleabstractie. Dat is het geval waarin een methode in twee (of meer) combinatoren niet exact hetzelfde realiseert maar toch iets gelijkaardigs. Het idee is dan het gemeenschappelijke in een trait te stoppen en de verschilpunten als vereisten voor de trait. De combinator die een dergelijke trait inmixt moet dan nog de verschilpunten aangeven door enkele methodes te implementeren. Deze methodes zijn echter veel korter dan de oorspronkelijke methode.

De moeilijkheid zit in het ontdekken van gelijkaardig gedrag en hoe men de verschilpunten moet definiëren om een algemeen bruikbare trait te bekomen. Om dit te verduidelijken beginnen we met een situatieschets en bespreken dan de trait die deze situatie opvangt.

Bij de implementatie van `enter` komt het vaak voor dat eerst de supercombinator moet worden opgeroepen. Als die combinator leidt tot een oplossing moet er dan een bijkomende actie gebeuren. Dat controleverloop hebben we geabstraheerd in de trait `OnEnterSolution`. In Codefragment 3.2 kan men dan zien dat men de `enter`-methode niet meer hoeft te implementeren, maar wel een methode `onEnterSolution`.

Codefragment 3.2: Met een controleabstractie kunnen we de implementatie van `enter` vereenvoudigen.

```
trait OnEnterSolution extends SearchHeuristic {
  abstract override def enter(/* ... */): Int = {
    val returnVal = super.enter(sm,na,top,currentNode)
    if (returnVal == SearchHeuristic.Solution) {
      onEnterSolution()
    }
    returnVal
  }
  def onEnterSolution()
}
```

We illustreren het nut aan de hand van de printcombinator. De lezer kan Codefragment 1.1 op pagina 2 vergelijken met die in Codefragment 3.3. Door de trait vermijden we het expliciet testen van de returnwaarde, wat voorspelbare code is. Hierdoor winnen we aan leesbaarheid en is de programmeur sneller in staat hoofd- van bijzaak te onderscheiden.

Codefragment 3.3: De printcombinator illustreert het nut van controleabstractie.

```
package searchCombinators
import node._
case class Print(
  protected val subCombinator: SearchHeuristic,
  variableOrArray: DomainVariableOrArray,
  visualizer : Visualizer = Visualizer
  ) extends SearchHeuristic with SubCombinator with OnEnterSolution {
  // First OnEnterSolution!
  def onEnterSolution() {
      val stringBuilder = new StringBuilder()
      /* ... Fill stringBuilder ... */
      visualizer.shout(this,stringBuilder.toString)
  }
}
```

In Codefragment 3.4 zien we iets gelijkaardigs: de childoproep van de onderliggende combinator kan ofwel een kindnode opleveren, ofwel niet. Alleen als er een kindnode is, moet de actieve combinator nog een bijkomende actie ondernemen. Dat kunnen we maar beter abstraheren. De combinator die trait `OnChild` inmixt, moet niet langer de methode `child` implementeren, maar in de plaats daarvan de methode `onChild`. In die methode moet dan de bijkomende actie staan die moet worden ondernomen als er effectief een kindnode is.

Codefragment 3.4: We abstraheren een vaakvoorkomend controlepatroon in de childmethode.

```
trait OnChild extends SearchHeuristic {
  abstract override def child(/* ... */): Option[Node] = {
    val maybeChild = super.child(sm,na,top,currentNode,childNumber)
    maybeChild match {
      case Some(ch) ⇒onChild(currentNode,ch)
      case None     ⇒ ;
    }
```

```
8       maybeChild
    }
10  def onChild(currentNode: Node, childNode: Node)
}
```

We willen ook opmerken dat zowel bij `OnChild` als `OnEnterSolution` de uiteindelijke returnwaarde niet beïnvloed wordt door de bijkomende actie van de actieve combinator. Deze actie gebeurt dus enkel en alleen voor haar *side effect*. Men zou de ideeën uit de vorige paragrafen ook kunnen zien als een implementatie van *advice* uit *aspect-oriented programming*.

3.5 Controleabstractie via anonieme functies

Naast vaak voorkomende patronen over meerdere combinatoren heen, hebben we soms ook patronen binnen de verschillende methodes van één enkele combinator. Hiervoor zijn traits minder geschikt omdat ze een complete methode implementeren waarbij eventueel een nieuwe methode door de inmixende klasse moet geïmplementeerd worden. We bespreken de „if" combinator om de situatie concreet te maken.

In de implementatie van de „if" combinator moet een conditievlag worden opgehaald uit een node, en gebeurt naargelang de status van die vlag, een actie op de ene subcombinator of op de andere (Figuur 3.5). Een dergelijke controleabstractie wordt geïmplementeerd als reguliere methode, maar heeft een functie als één van zijn argumenten. In dit geval gaat het om een generieke functie, maar dit is niet altijd nodig. Hier is de returnwaarde van de abstractie die van de functie, en die kan ook () (lees „unit") zijn.

Bij de toepassing van `ifCondition` in de andere methodes zien we anonieme functies. Ter herinnering: de parameters van een dergelijke functie staan voor de pijl, en het statement (eventueel een blokstatement) erna. De underscore is ook hier een dummywaarde.

Codefragment 3.5: Controleabstractie via anonieme functie.

```
case class If
2   (private val cond: VarCond,
     private val then: SearchHeuristic,
4    private val `else`: SearchHeuristic)
    (implicit nodeAllocator: NodeAllocator)
6   extends CombinatorWithItems[Boolean](nodeAllocator) {

8   /* .. methods not using control abstraction left out ... */

10  /** Control abstraction simplifying the implementation. */
    private def ifCondition[T](nodeContainingFlag: Node,
12    action: Function2[SearchHeuristic,Boolean,T]): T = {
      val conditionFlag = lookupItem(nodeContainingFlag)
14    if (conditionFlag) {
        action(`else`, conditionFlag)
16    } else {
        action(then,conditionFlag)
18    }
    }
20
    def child(/* ... */): Option[Node] = {
22    ifCondition(currentNode, (sub,conditionFlag) ⇒ {
          val maybeChild = sub.child(/* ... */)
24        /* Another control abstraction */
          ifChild(maybeChild) {insert(_, conditionFlag)}
```

```
26          maybeChild
          } )
28    }

30    def stat(node: Node) = {
        ifCondition(node,(sub,_) ⇒ sub.stat(node))
32    }

34    def copy(oldNode: Node, newNode: Node) = {
        val conditionFlag = ifCondition(oldNode,
36          (sub, conditionFlag)
            ⇒  { sub.copy(oldNode,newNode); conditionFlag } )
38      insert(newNode,conditionFlag)
      }
40  }
```

3.6 Implementatie van afgeleide heuristieken

Afgeleide heuristieken kunnen als functie geïmplementeerd worden. Scala laat niet toe dat functies geen methode zijn van een object. Er bestaat echter een zogenaamd pakketobject waarin topniveau functies kunnen worden geplaatst (zie Sectie 2.2.5 op pagina 12). Een voorbeeld werd reeds gegeven in Codefragment 2.7 op pagina 12. In datzelfde object wordt een deel van de syntactische suiker gedefinieerd.

3.7 Impliciete parameters

Enkele combinatoren hebben meer parameters dan we zouden willen in een declaratieve syntax. Wat we verwachten voor de „if" combinator is dat hij een conditie en twee subcombinatoren neemt als argument. Eigenlijk is dat niet zo: hij heeft ook een instantie nodig van `NodeAllocator`.

Een situatie waarin er echter meer dan één dergelijk object nodig zou zijn, lijkt eerder zeldzaam. Toch is een singletonobject hier eerder drastisch. Daarom gebruiken we een impliciete klassenparameter. Daardoor hoeven we geen instantie van `NodeAllocator` mee te geven en gaat de compiler op zoek naar een geschikt object. Men zou dit als een favoriet of geprivilegieerd object kunnen beschouwen [16]. Willen we wel een instantie meegeven, dan moet dat in een tweede argumentenlijst.

De theorie achter het mechanisme is als volgt: als een argumentenlijst niet opgegeven wordt gaat Scala op zoek naar objecten in scope die kunnen gebruikt worden. Om problemen te vermijden moeten die waarden wel als impliciet zijn gemarkeerd. De impliciete parameters moeten in een aparte argumentenlijst staan. In functionele programmeerkringen heet dit *currying* [18] (zie Sectie 2.2.7 op pagina 16). Impliciete parameters zijn erg krachtig maar moeten toch met de nodige voorzichtigheid worden gebruikt om geen ongewenste effecten te creëren. Merk ook op dat *implicits* niet hetzelfde zijn als meer bekende default parameters.

Een codevoorbeeld maakt het concept impliciete parameter duidelijk. Veronderstel dat we een methode hebben die online eten bestelt bij ons vast fastfoodrestaurant (in [16] gaat het om cafeïnehoudende dranken; Codefragment 3.6).

Codefragment 3.6: Voorbeeld van een impliciete parameter.

```
class Food(val what: String, val quantity: Int)
```

```
2
object FoodOrderer {
4   def order(deliveryTime: String)(implicit what: Food) {
      /* Actually order food ... */
6     println("Ordered " + what.what + " for delivery at " + deliveryTime + ".")
    }
8 }

10 implicit val pizza = new Food("pizza",1)
```

Veronderstel ook dat je favoriete menu pizza is. Dan kun je de methode oproepen met enkel het tijdstip van gewenste levering om pizza te bestellen en met twee argumenten, elk in een aparte lijst, als je iets anders wil eten:

```
FoodOrderer.order("18:00")
FoodOrderer.order("18:00")(new Food("French fries",1))
```

3.8 Uitgewerkte voorbeelden

In deze sectie bespreken we aan de hand van voorbeelden wat men kan bereiken met de technieken die we hiervoor besproken hebben. Voor een aantal geselecteerde combinatoren beginnen we met een heel eenvoudige implementatie en passen achtereenvolgens de gepaste technieken toe.

3.8.1 Voorbeeld: incrementcombinator

Dit voorbeeld is bedoeld om de effectiviteit van traits te illustreren. Daarnaast toont het ook het belang van een goed systeemontwerp en welke rol refactoring hierin kan spelen. De `Incr`-combinator heeft een eenvoudige functionaliteit: hij verhoogt de waarde van een opgegeven gebruikersvariabele met één. In eerste instantie kunnen we dit als volgt coderen:

Codefragment 3.7: `Incr` zonder traits

```
/* This version does not use traits at all. */
2 class Incr(
    variableName: String,
4   debugName: String
    ) extends SearchHeuristic(debugName)
6     with EmptyCombinator
      // NoChild takes effect first
8     with NoChildMethod
    {
10  private var variableStackPointer: Node#ItemID = _
    /** The enter-message: increments the variable and succeeds.
12    */
    override def enter(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator,
14    top: SearchHeuristic, currentNode: Node): Int = {
      val IntItem(variableValue) = currentNode.item(variableStackPointer)
16    currentNode.depositItem(variableStackPointer,IntItem(variableValue + 1))
      SearchHeuristic.Solution
18  }
    /** Gets the index of the variable named at construction time in the given
20   * environment.
      */
22  override def init(node: Node, vm: VarAccessor#VM, environment: LetBinding)
     = {
24    variableStackPointer = environment(variableName)
```

```
  }
  def child(sm: StateManager, na: NodeAllocator, top:
  SearchHeuristic, currentNode: Node, childNumber: Int): Option[Node] = {
    throw new RuntimeException("child should not be called.")
  }
  /* Empty methods shown intentionally... */
  def next(currentNode: Node, nextNode: Node) {}
  def stat(node: Node) {}
  def copy(oldNode: Node, newNode: Node) {}
}
```

Deze code bevat heel wat lege methodes, en ook de `child`-methode doet niks nuttigs. Zoals we eerder hebben vermeld komt dit ook bij de andere combinatoren vaak voor. We voorzien in een tweede versie dan ook enkele traits. Door de overeenkomsten met andere combinatoren te bekijken, zien we dat we het best twee traits definiëren: `EmptyCombinator` en `NoChildMethod`. De code ziet er nu uit als volgt:

Codefragment 3.8: `Incr` met eenvoudige traits

```
/* This version uses some simple traits. */
class Incr(
  variableName: String,
  debugName: String
  ) extends SearchHeuristic(debugName)
    with EmptyCombinator
    // NoChild takes effect first
    with NoChildMethod
  {
  private var variableStackPointer: Node#ItemID = _
  /** The enter-message: increments the variable and succeeds.
    */
  override def enter(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator,
    top: SearchHeuristic, currentNode: Node): Int = {
    /* ... */
  }
  /** Gets the index of the variable named at construction time in the given
   * environment.
    */
  override def init(node: Node, vm: VarAccessor#VM, environment: LetBinding)
   = {
    variableStackPointer = environment(variableName)
  }
}
```

In deze code vallen vooral de methode `init` en de private variabele `variableStackPointer` op. Behalve voor een programmeur die volledig vertrouwd is met de rest van het systeem, is het helemaal niet duidelijk wat hier de bedoeling van is. Ook de code van `enter` is ingewikkeld om gewoon de waarde van een variabele met eentje te verhogen.

De oorzaak hiervan is de manier waarop men aan de variabele om te incrementeren moet komen: men geeft de naam van de variabele mee in de constructor van de `Incr`-combinator. In de `init`-methode zoekt men dan aan de hand van de naam op wat de index is in een tabel van variabelen. Dat moet slechts éénmaal gebeuren en is vooral interessant wanneer men manueel het geheugen moet gaan beheren. We zitten hier immers nog erg dicht bij een letterlijke vertaling van de C++ code waarvan we vertrokken zijn voor onze implementatie in Scala.

Scala heeft echter garbage collection, dus is er eigenlijk geen reden meer om de variabelen te gaan bijhouden in een tabel. In plaats van de naam van de letvariabele door te geven, kunnen we veel beter het object dat de variabele voorstelt (m.a.w. de variabele zelf) doorgeven. Dat vereist enkele ingrijpende aanpassingen in de basisklassen waarop het systeem gebouwd is waarop we hier niet verder ingaan. Het eindresultaat is echter belangrijk: de `enter`-methode is opeens veel leesbaarder en de `init`-methode is leeg:

Codefragment 3.9: `Incr` met directe variabelentoegang

```
case class Incr
  (variable: LetVar)
  extends SearchHeuristic
  with EmptyCombinator
  // NoChild takes effect first
  with NoChildMethod
  {
  /* Simplified enter */
  override def enter(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator,
    top: SearchHeuristic, currentNode: Node): Int = {
    variable.value += 1
    SearchHeuristic.Solution
  }
  override def init(node: Node, vm: VarAccessor#VM, environment: LetBinding)
   = { /* Deliberately empty */ }
}
```

Het is nu voor de hand liggend om ook de lege `init`-methode op te nemen in een trait. Uit vergelijking met andere combinatoren blijkt dat de trait `EmptyCombinator` hier — zoals verwacht — prima voor voldoet. Merk ook op dat op dat we van de klasse een *case class* gemaakt hebben. In dit specifiek geval is de bedoeling hiervan vooral instanties te kunnen maken zonder het `new`-keyword nodig te hebben.

Door verdere herwerkingen op systeemniveau zijn we er vervolgens in geslaagd het aantal parameters van de `enter`-methode te reduceren tot vier (met ook meer duidelijke namen voor de types). De signatuur ziet er dan uit als volgt:

def enter(sm: StateManager, na: NodeAllocator, top: SearchHeuristic, currentNode: Node): Int

Hoewel dit waarschijnlijk de code is die de meeste programmeurs zouden schrijven, kunnen we nog één stap verder gaan. Voor de `Incr`-combinator alleen is dit eerder overdreven, maar wanneer de andere combinatoren bekijken zien we dat het volgende patroon vaak terugkeert:

- Doe een bepaalde handeling.
- Geef de constante `SearchHeuristic.Solution` terug.

Wie een basiskennis functioneel programmeren heeft, herkent in de eerste stap van het patroon meteen een zogenaamd *side effect*. De returnwaarde van `enter` ligt vast bij deze en vijf andere combinatoren (zie Appendix C). Het enige verschil is het neveneffect. Dat is ook de absolute kerntaak van de `Incr`-combinator: „verhoog een letvariabele met één".

Patronen helpen de bedoeling van iets duidelijk te maken. Denken we maar aan de talloze *design patterns* die behoren tot de standaardkennis van elke programmeur [11] of aan de idiomen die in elke programmeertaal aanwezig zijn. Daarom introduceren we een extra trait `DoAndSucceedEnter`. De naam is geïnspireerd op het Haskell sleutelwoord dat voor neveneffecten wordt gebruikt: `do`. We kunnen dan de `Incr`-combinator als volgt schrijven:

Codefragment 3.10: `Incr` met een patroon voor neveneffecten

```
case class Incr
  (variable: LetVar)
  extends SearchHeuristic
  with EmptyCombinator
  with NoChildMethod
  // DoAndSucceedEnter takes effect first
  with DoAndSucceedEnter
  {
  /* The side effect for enter before returning solution constant. */
  def enterSideEffect() {
    variable.value += 1
  }
}
```

Wanneer we nu onze allereerste versie met de laatste vergelijken, zien we een belangrijke code-reductie: van 26 regels naar elf (zonder commentaar).

3.8.2 Voorbeeld: if-combinator

Dit voorbeeld toont het nut van controleabstracties. Daarnaast vormt het een goede illustratie van de eeuwige tradeoff tussen efficiëntie en onderhoudsgemak. De `If`-combinator is een complexe combinator waar elke methode zeer specifiek is. Bijgevolg kunnen we met traits niets doen om delen van de code weg te abstraheren. Om het geheel leesbaar te houden, focussen we op de `enter` en `child` methodes. In een eerste implementatie, die bijna een letterlijke vertaling is van de implementatie in C++, ziet de code voor die methodes er uit als volgt:

Codefragment 3.11: Eerste versie van een deel van `If`

```
class If(private val cond: VarCond, private val then: SearchHeuristic, private
val `else`: SearchHeuristic, debugName: String) extends SearchHeuristic(debugName) {
  private var environment: LetBinding = _
  def enter(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator, top:
  SearchHeuristic, currentNode: Node): Int = {
    if (currentNode.item(stackPointer).toInt == 0) {
      // combinator condition true
      then.stat(currentNode)
      if (!cond.evaluate(sm,vm,currentNode)) {
        currentNode.item(stackPointer).updateInt(1) // flag true = cond. false
        then.dispose(/* ... */)
        `else`.init(/* ... */)
        `else`.enter(/* ... */)
      } else {
        then.enter(/* ... */)
      }
    } else { // combinator condition false
        `else`.enter(/* ... */)
    }
  }
  def child(sm: StateManager, vm: VarAccessor#VM, na: NodeAllocator, top:
  SearchHeuristic, currentNode: Node, childNumber: Int): (Node,Boolean) = {
    var tuple: (Node,Boolean) = (null,false) // dummy
    val conditionFlag = currentNode.item(stackPointer).toInt
    if (conditionFlag == 0) {
      tuple = then.child(/* ... */)
    } else {
      tuple = `else`.child(/* ... */)
    }
```

```
    if (tuple._2) {
31    tuple._1.items += (stackPointer -> new Item(conditionFlag))
    }
33  tuple
  }
35 /** Return maximum size of item this heuristic requires (including
   * sub-heuristics) and compute stack-pointers, i.e. _sp
37   */
  def itemSize(): Int = {
39  stackPointer = then.itemSize max 'else'.itemSize
    1 + stackPointer
41 }

43 /* Left out other methods for reasons of space and clarity. */
}
```

. Deze code is op het eerste zicht Chinees. In de eerste plaats komt dit door de manier waarop de combinator zijn lokale toestand bijhoudt. In C++ is er geen garbage collectie en kan men om efficiëntieredenen op erg laag niveau werken. Dit manifesteert zich hier in de variabele `stackPointer` en in de methode `itemSize`. De bedoeling van de methode `itemSize` is het aantal bytes terug te geven dat deze combinator nodig heeft voor het opslaan van zijn lokale toestand. Dit aantal bytes wordt dan in C++ voorzien op het adres aangewezen door `stackPointer`. In Scala is dit de index in een tabel; een onelegante manier van werken.

We willen speciaal wijzen op regel 6 van de code: een `If`-combinator slaat (in deze versie) een gehele waarde op. Apart van het feit dat een Boolse waarde volstaat en duidelijker is, verliezen we door de opslag van alle lokale state in een centrale tabel de typeinformatie! Hierdoor moeten we in regel 6 een cast doen (verborgen in `toInt`). We willen dit zoveel mogelijk vermijden.

Een naïve oplossing is om een tabel te voorzien voor elk type van lokale state die moet worden bijgehouden. Maar eigenlijk willen we dat andere programmeurs gemakkelijk combinatoren kunnen toevoegen, mogelijks met een type van lokale state waar we geen tabel voor hebben. Dit is dus niet goed genoeg. Kunnen we het type van het item op de één of andere manier mee opslaan in een algemene tabel? In Haskell bestaat een heterogene lijst die HList heet [1, 2, 13]. Dit is echter een moeilijke aanpak...

Het kan eenvoudiger door elke combinator zijn eigen lokale state te laten bijhouden. Het typerende voor lokale state is dat ze verschilt per node van de zoekboom. We moeten er dus wel op letten om de state niet langer bij te houden wanneer die node niet langer nodig is (zolang er referenties naar zijn, kan de garbage collector zijn werk niet doen). We hebben dit voorzien in de superklasse `CombinatorWithItems[T]`, al kan het eenvoudiger met zogenaamde *zwakke referenties*. De typeparameter in de superklasse is het type van de lokale state. Op de systeembrede aanpassingen waardoor de types in de volgende versie van de code anders heten, en er ook minder parameters zijn, gaan we hier niet dieper in. De code voor de methodes `enter` en `child` ziet er dan uit als volgt:

Codefragment 3.12: Deel van `If` met superklasse

```
1 case class If
  (private val cond: VarCond,
3  private val then: SearchHeuristic,
   private val 'else': SearchHeuristic)
5 (implicit nodeAllocator: NodeAllocator)
  extends CombinatorWithItems[Boolean](nodeAllocator) {
7 def enter(/* ... */): Int = {
    val conditionFlag = lookupItem(currentNode)
9   if (!conditionFlag) {
```

```
        then.stat(currentNode)
11      if (!cond.evaluate(sm,currentNode)) {
          insert(currentNode,true) // flag true = cond. false
13        `else`. init(currentNode)
          `else`. enter(sm,na,top,currentNode)
15      } else { then.enter(/* ... */) }
      } else { `else`. enter(/* ... */) }
17  }
    def child(/* ... */): Option[Node] = {
19    var maybeChild: Option[Node] = None // dummy
      val conditionFlag = lookupItem(currentNode)
21    if (!conditionFlag) {
        maybeChild = then.child(sm,na,top,currentNode,childNumber)
23    } else { maybeChild = `else`.child(/* ... */) }
      ifChild(maybeChild) {insert(_, conditionFlag)}
25    maybeChild
    }
27  /* Other methodes left out for reasons of space and clarity. */
}
```

We beschikken dus over de methodes `insert` en `lookupItem` om om te gaan met de lokale state. Bemerk ook de controleabstractie `ifChild`.

Wanneer we echter alle methodes van `If` zien, bemerken we dat het ophalen van de conditievlag, en het uitvoeren van een actie op de ene of de andere subcombinator naargelang die vlag, een terugkerend patroon is. Dit particulier voorbeeld wordt besproken in Sectie 3.5 op pagina 30. De lezer die onze eerste versie van de `child`-methode vergelijkt met de versie die aldaar te vinden is, zal bemerken dat we opnieuw een niet onaanzienlijke codereductie teweeg hebben gebracht.

3.9 Solveronafhankelijkheid

We hebben reeds vermeld dat onze implementatie solveronafhankelijk is. Om een solver te koppelen aan ons systeem moeten een aantal traits geïmplementeerd worden. In Codefragment 3.13 wordt een eerste set methodes vermeld. De belangrijkste methode hier is die voor het opleggen van constraints, `imposeConstraint`. Wanneer we een bijkomende veronderstelling toevoegen aan een space, moeten we hier later kunnen op terugkeren om een volledige gevalsanalyse mogelijk te maken (zie de voorkennis in Hoofdstuk 2 op pagina 4). Daarbij komt de klasse `StateManager` in het spel.

Codefragment 3.13: Elke koppeling moet een space voorzien.

```
trait Space {
2   def fail ()
    def failed (): Boolean
4   def imposeConstraint(constraint: Constraint)
}
```

In de voorkennis hebben we ook vermeld dat we bij het evalueren van een zoekheuristiek een boomstructuur bekomen. Een dergelijke boom bevat nodes, gemodelleerd door klasse `Node`. Elke node bevat eigenlijk niet veel meer dan de toestand waarin de de variabelen zich bevinden en de constraints, dus in feite een soort space. De toestand is afhankelijk van de gebruikte constraint solver. Laat ons dit het type `State` noemen. De taken van `StateManager` zijn dan:

1. Gegeven een instantie van `State`, vind de bijhorende space zoals die er op het moment van aanmaken van de instantie uitzag.

2. Gegeven een `State` en een bijkomende veronderstelling (klasse `Branch`), maak een nieuwe instantie van `State` aan.

Hoe deze taken precies gebeuren zal afhangen van de gebruikte constraint solver. Sommige systemen laten toe een diepe kopie te nemen van een space (bijvoorbeeld Gecode). Andere constraint solvers laten toe wijzigingen aan één enkele space ongedaan te maken (bijvoorbeeld JaCoP). We bekomen dus de methodes zoals opgelijst in Codefragment 3.14

Codefragment 3.14: De trait `StateManager`.

```
1 trait StateManager {
    def space(state: State): Space
3   def child(parent: State, branch: Branch): State
    def release(state: State)
5   def rootSpace: Space
  }
```

De implementatie van `StateManager` voor de performante koppeling met JaCoP is goed te begrijpen (Codefragment 3.15). Deze koppeling werkt enkel wanneer de zoekboom doorlopen wordt in diepte-eerst volgorde. Drie gevallen zijn mogelijk (regels 4 tot 11):

1. De vorige node in de boom heeft geen kinderen. We wensen de space te bepalen voor zijn broer. We blijven dus op dezelfde diepte in de boom.

2. De vorige node in de boom heeft minstens één kind waarvoor we nu de space wensen te bepalen. We gaan een niveau dieper in de boom.

3. We hebben de volledige kinderboom van een node x doorzocht. De vorige node z was dus de meest rechtse afstammeling van x. Node x heeft echter nog een broer y. Ten opzichte van node z kan y vele niveaus hoger liggen.

In het eerste geval verwijderen we de wijzigingen aangebracht door de vorige node (regel 13) en blijven op hetzelfde niveau. We posten de beslissing die genomen werd op de tak naar de huidige node (regel 15). Merk op dat de rootspace geen broers heeft, maar aan het begin van het programma, deze methode ook wordt opgeroepen voor de rootspace. Op dat speciale geval moeten we expliciet testen (regel 12).

Het tweede geval is het eenvoudigst. We verhogen het niveau (regels 19 en 20) waarna we de beslissing die werd genomen op de tak naar de huidige node posten.

In het laatste geval verwijderen we alle beslissingen die genomen werden tot op het niveau van de oorspronkelijke node x (regels 24 tot 28). Daarna verwijderen we ook die wijzigingen en posten diegene horend bij de nieuwe node y.

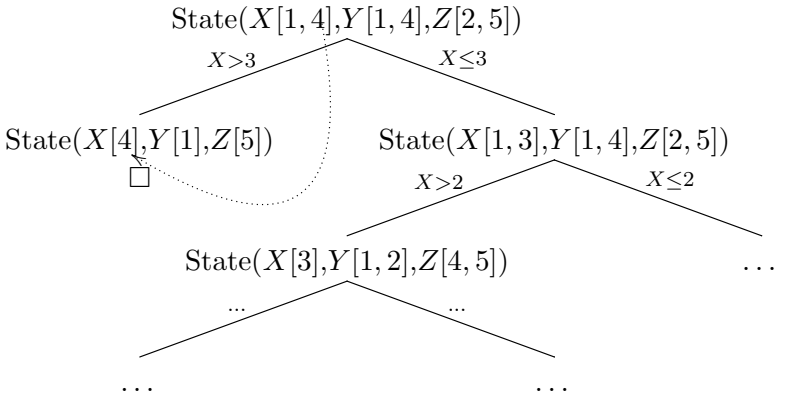
Codefragment 3.15: Beheer van spaces in de koppeling met JaCoP.

```
  class StateManagerEfficientImpl(val rootSpace: Space) extends StateManager {
2   val jacopSpace = rootSpace.asInstanceOf[SpaceJaCoPImpl].delegate
    var currentLevel = 0
4   def space(state: State): Space = {
      state.level match {
6       case level if level == currentLevel     ⇒ sameLevel()
        case level if level == currentLevel + 1 ⇒ oneLevelDown()
8       case level if level < currentLevel      ⇒ levelsUp()
        case _ ⇒ throw new RuntimeException(/* ... */)
10    }
      def sameLevel() = {
```

```
12        if (currentLevel > 0) {
            jacopSpace.removeLevel(currentLevel)
14          jacopSpace.setLevel(currentLevel)
            state.branch.post(rootSpace)
16        }
      }
18    def oneLevelDown() = {
        currentLevel += 1
20      jacopSpace.setLevel(currentLevel)
        state.branch.post(rootSpace)
22    }
      def levelsUp() = {
24      while (currentLevel > state.level) {
          jacopSpace.removeLevel(currentLevel)
26        jacopSpace.setLevel(currentLevel - 1)
          currentLevel -= 1
28      }
        jacopSpace.removeLevel(currentLevel)
30      jacopSpace.setLevel(currentLevel)
        state.branch.post(rootSpace)
32    }
      rootSpace
34  }
    /* ... Left out other methods ... */
36 }
```

We illustreren de vorige paragrafen aan de hand van Figuur 3.2, een licht aangepaste kopie van Figuur 2.1 op pagina 5. Een instantie van `State` is een blackbox waarvan we de echte inhoud niet kennen, maar met de methode `space` van `StateManager` kunnen we er wel een instantie van `Space` uit afleiden. De pijl in stippellijn geeft aan dat we een nieuwe `State` kunnen afleiden uit een voorgaande met de methode `child`.

Het exacte type van de variabelen is ook solverafhankelijk. Voor elke solver moeten we immers onze methodes mappen op die van de variabelen van de solver. We voorzien dus in een wrapper. Het probleem hier is dat we de gebruikerscode absoluut niet afhankelijk willen maken de solver. Dus wanneer we een variabele aanmaken is het exacte type solverafhankelijk maar we mogen dit type niet vermelden in de gebruikerscode. Dit lijkt paradoxaal.



Figuur 3.2: Een voorbeeldboom ter illustratie van de werking van `StateManager`.

Toch is er een vrij eenvoudige oplossing: bij elke trait of abstracte klasse die definieert wat een concrete implementatie van dit type variabele moet kunnen, definiëren we een companion object met een factorymethode (Sectie 2.2.2). In die methode maken we de variabele aan met het concrete type dat hoort bij de gekoppelde solver (Codefragment 3.16).

Codefragment 3.16: Companion object om solveronafhankelijk variabelen aan te maken.

```
object IntVar {
  def apply(lowerBound: Int, upperBound: Int, space: Space, name:String): IntVar = {
    new IntVarJaCoPImpl(lowerBound,upperBound,space,name)
  }
}
abstract class IntVar // ...
```

Door deze manier van werken, is het dus — net als bij een lijst uit de standaardbibliotheek — slechts mogelijk op de volgende manieren gehele domeinvariabelen aan te maken, waarbij de tweede manier zeer zeldzaam is:

```
val a = IntVar(0,10,space,"test")
val a = IntVar.apply(0,10,space,"test2")
```

Hoewel deze techniek vermijdt dat we overal gebruikerscode moeten aanpassen, zijn het enkele methodes in het solveronafhankelijke pakket die toch solverafhankelijk zijn. Dat komt door de beperking dat companion objecten in hetzelfde bestand moeten worden gedefinieerd als de klasse of trait waarbij ze horen. Ook dit probleem is te omzeilen door delegatie, maar erg omslachtig. We hebben er dan ook voor geopteerd dit niet te doen.

In de volgende sectie bespreken we de impact van de volgorde waarin we de boom doorlopen.

3.10 Doorlopen van de zoekboom

De zoekboom uit Figuur 3.2 kan op verschillende manieren doorlopen worden. Bekende strategieën zijn *depth first search* (DFS) en *breadth first search* (BFS). De component die boom doorloopt heten we *search engine* [19] en staat orthogonaal op de variabele- en waardeselectiestrategie. De strategie zelf heet *queuing strategy*.

Wanneer we gebruik maken van JaCoP als constraint solver is depth first search veel efficiënter. Dat komt omdat JaCoP niet toelaat een space te kopiëren. Veronderstel dat we breadth first search gebruiken, dan moeten we iedere keer vertrekken vanaf de rootspace, die wijzigen door het toepassen van de branches in de juiste volgorde en nadat we klaar zijn met deze node de wijzigingen aan de rootspace opnieuw ongedaan maken. Dat is natuurlijk erg onefficiënt wanneer de boom diep wordt, wat hij ook is, behalve voor triviale problemen. Met depth first search kunnen we eerst een volledig pad van de rootnode naar een bladnode verkennen vooraleer we wijzigingen ongedaan moeten maken.

Wanneer we een diepe kopie van een space kunnen maken is het eigenlijk gemakkelijker. Anderzijds is het kopiëren op zich duurder dan het bijhouden van een verschil met de ouderspace. Daarom kunnen we ook een hybride strategie gebruiken die slecht een kopie neemt op bijvoorbeeld elk achtste diepteniveau.

Voor JaCoP hebben we zowel een generieke implementatie die elke queuing strategy aankan, in het bijzonder BFS, en een veel performantere specifiek voor DFS.

3.11 Domeinvariabelen

Omdat we zowel Boolse als integere variabelen hebben, en daarnaast ook arrays van die twee types, die nogal wat gemeenschappelijk hebben, komen ook hier traits in combinatie met abstracte types goed van pas. De solverafhankelijke structuur wordt gereflecteerd in de solverafhankelijke code. Op solveronafhankelijk niveau staat aan de top van de hiërarchie de trait `DomainVariableOrArray` (Figuur 3.3, waarbij opnieuw de grijze pijlen staan voor traitinclusie). Deze trait groepeert singleton domeinvariabelen en arrays. Dit overkoepelend type laat toe combinatoren met beide types te laten werken. De trait declareert de methodes `size` en `getIndex`. Voor enkelvoudige variabelen is de grootte altijd één.

Merk op dat de methode `getIndex` een singleton variabele teruggeeft, maar dat we op dit punt het type nog niet kennen. Wat we wel weten is dat het een singleton is. Daarom definiëren we een zogenaamd *abstract type member* en garanderen dat het zeker een singleton variabele is met een *upper bound*:

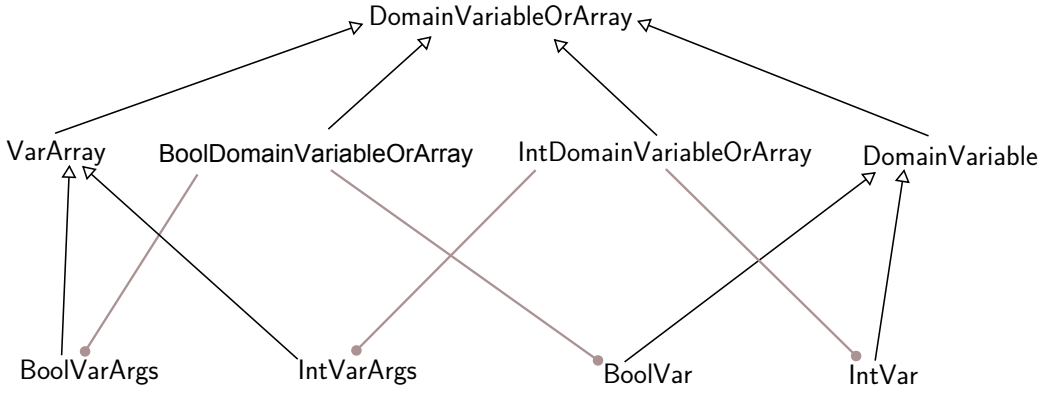
type SingletonVariableType <: DomainVariable

Singleton domeinvariabelen mixen de subtrait `DomainVariable` in; arrays gebruiken `VarArray`. Een enkelvoudige domeinvariabele heeft bijvoorbeeld een onder- en bovengrens. Een array moeten we kunnen indexeren. Scala laat toe het te laten lijken alsof de arrays in de taal zijn ingebakken: we kunnen ons eigen arraytype indexeren als `myArray(i)` door het implementeren van een methode `apply`. Daarnaast declareren we ook een methode `foreach` die een functie als argument neemt. Deze methode kunnen we gebruiken met een functionele for-lus.

Orthogonaal aan bovenstaande onderverdeling hebben we het onderscheid tussen integere variabelen en Boolse variabelen. Integere variabelen, zij het nu singleton of array, mixen trait `IntDomainVariableOrArray` in. Boolse variabelen gebruiken `BoolDomainVariableOrArray`. Deze traits definiëren geen nieuwe methoden, maar kunnen nu wel het abstracte membertype uit `DomainVariableOrArray` toekennen.

Op basis van de bovenstaande traits kunnen we nu alle gewenste types afleiden. Die afleidingen, met o.a. `BoolVar` en `IntVarArgs`, moeten nog steeds solveronafhankelijk zijn. Daarom zijn het traits of abstracte klassen. In Codefragment 3.17 tonen we `IntVar` omdat deze voorziet in syntactische suiker voor condities.

Codefragment 3.17: De abstracte klasse IntVar voorziet in syntactische suiker.



Figuur 3.3: Structuur van de domeinvariabelen.

```
1 abstract class IntVar extends DomainVariable with IntDomainVariableOrArray {
    def degree(): Int
3   /* ... left out some similar methods ... */
    def ranges(): List[IntVarRange]
5   def domainSize(): Int
    def <(y: LetVar): VarCond = CmpCondDVar(this,y,-1)
7   def <=(y: LetVar): VarCond = CmpCondDVar(this,y,-1,1)
    /* ... left out > and >= ... */
9 }
```

Hoofdstuk 4

Evaluatie

In dit hoofdstuk evalueren we zowel de geschiktheid van Scala als implementatietaal, als onze eigen code in vergelijking met de oorspronkelijke implementatie in C++, en in vergelijking met standaardgebruik van de constraint solver JaCoP. Hierbij komen aspecten kijken als performantie, onderhoudbaarheid, platformonafhankelijkheid, codeomvang en dergelijke meer. We formuleren de plus- en minpunten van Scala, en geven ideeën voor eventuele toekomstige aanpassingen aan de taal.

4.1 Scala als geschikte implementatietaal

Persoonlijk vinden we het meest jammer dat de precedentie van de operatoren bepaald wordt door het eerste karakter. Het is hierdoor vaak niet mogelijk én een voor de hand liggend symbool te kiezen, én haakjes zoveel mogelijk te vermijden. Dit doet vaak afbreuk aan het gevoel dat we een eigen taal hebben gedefinieerd binnen het Scala-framework. Ter illustratie geven we in Bijlage A op pagina 54 een overzichtstabel van de precedenties. Een verwant, maar voor deze thesis van minder belang, probleem is dat de associativiteit bepaald wordt door het al of niet gelijk zijn aan een dubbelpunt. In het eerste geval hebben we rechtsassociativiteit.

Verder is het vaak niet mogelijk syntactische suiker te implementeren daar waar het voor de onderhoudbaarheid van de code het meest nodig is. Dat komt omdat een infixoperator moet worden gedefinieerd als een methode van zijn linkeroperand, met de rechteroperand als parameter. De mogelijkheid tot een definitie met beide operandi als parameters ontbreekt, maar is volgens ons belangrijk om alle syntactische suiker op een centrale plaats te groeperen. De groepering is niet noodzakelijk verwant met het type van het linkeroperand.

Scala biedt een uitgebreide API waarvan vooral het collectieframework bijzonder van pas kwam. Bijzondere vermelding verdient de aanwezigheid van onmuteerbare (*immutable*) collecties. In combinatie met *hogere-orde functies* als `map` en `foreach` waren we in staat verschillende algoritmen beknopt te coderen. In Codefragment 4.1 tonen we een onderdeel van de implementatie van een *publish-subscribe* mechanisme uit de klasse `NodeAllocator`.

Codefragment 4.1: Illustratie van hogere-orde functies: de notificatiecomponent van de klasse `NodeAllocator`.

```
1 private val subscribers = collection.mutable.Set[SearchHeuristic]()
  /* ... */
3
  /** Notify all subscribers. */
5 private def notifyCombinators(about: Node) {
    subscribers.foreach(_.disposeNode(about))
7 }
```

Het traitmechanisme is veel krachtiger dan standaard enkelvoudige overerving. Dit werd reeds geïllustreerd in Codefragment 2.5 op pagina 11 en Figuur 3.1 op pagina 26. Toch zijn hier enkele beperkingen: zo is het niet mogelijk een trait meer dan één keer in te mixen. Hiervoor zouden we een soort hernoemingsmechanisme nodig hebben, waarvan de syntax zou kunnen lijken op die voor het hernoemen van klassen en pakketten (zie Bijlage B op pagina 55).

Het pakketmechanisme is veel flexibeler dan we gewoon zijn uit Java. Dit komt de algemene structuur van de code ten goede. Hiermee verwant zijn de rijke toegangsmodifiers. Zichtbaarheid van een klasse of methode tot in een zeker pakket, komt van pas in de klasse `Node` en in de implementatie van de koppeling met de constraint solver. Er mogen immers slechts objecten van het type `Node` aangemaakt worden door de klasse `NodeAllocator` (Codefragment 4.2). In de koppeling willen we de solverafhankelijke constraint variabele verbergen voor ons systeem, terwijl het wel noodzakelijk is eraan te kunnen in de solverspecifieke klasse voor het opleggen van constraints.

Codefragment 4.2: Rijke toegangsmodifiers kunnen vermijden dat `Node` overal kan geïnstantieerd worden.

```
1  package searchCombinators.node {
     import searchCombinators._
3    /* Featuring constructor only accessible in some surrounding package */
     class Node private[node] {
5      var state: State = _ // Getter and setter
     }
7
     class NodeAllocator {
9
       /* ... */
11
       def allocate(): Node = {
13       if (buffer.isEmpty)
           new Node // Can make new instance of Node here.
15       else {
           val node = buffer.pop()
17         assert(node.state == null)
           node
19       }
       }
21   }
   }
23 package searchCombinators {
     /* Cannot make new instance of Node here. */
25 }
```

Het is ook niet mogelijk een *companion object* (een singleton object dat hoort bij een klasse) elders te definiëren dan in het bestand waar ook de klasse werd gedefinieerd. Enerzijds kunnen we dit begrijpen: een companion object heeft toegang tot de private leden van de bijhorende klasse. Het gaat dus om een geprivilegieerde toegang die bij misbruik de encapsulering kan doorbreken en aldus de uitbreidbaarheid van het systeem in gevaar brengen. Anderzijds is het vergelijkbare mechanisme *friend class* uit C++ veel flexibeler.

4.2 Evaluatie van onze implementatie

In deze sectie evalueren we onze eigen code in haar meest belangrijke aspecten.

4.2.1 Effectiviteit van traits

Traits zijn een efficiënte manier om veelvoorkomende patronen te extraheren. We implementeerden 21 basiscombinatoren. De patronen die hierin voorkomen hebben we verzameld in zeven traits. Per combinator worden gemiddeld 1,76 traits ingemixt, met een standaardafwijking van 1,06. Soms worden er tot drie traits gebruikt. Hun kracht komt inderdaad voor het grootste deel uit hun combinatie, net als bij de combinatoren zelf. Voor een gedetailleerd overzicht verwijzen we naar Appendix C.

Het is opvallend welke de combinators die geen traits inmixen: `SeqAnd`, `SeqOr` en `If`. Deze traits hebben namelijk meer dan één subcombinator en zijn het moeilijkst om te implementeren. Het is dan ook niet verwonderlijk dat patronen daar minder voorkomen. Traits kunnen bovendien maar één enkele keer ingemixt worden.

Hoe effectief is het nu om zeven traits te hebben? Gemiddeld moeten na inclusie van de traits die van toepassing zijn nog 2.13 berichten uit het message protocol geïmplementeerd worden (standaardafwijking 1,73). Van de 21 basiscombinatoren hebben er echter 13 maar één enkele methode meer te implementeren. Het gaat om de entermethode, of een methode die in de plaats daarvan kan worden geïmplementeerd. Deze methode bepaalt namelijk het meest het karakteristieke gedrag van de combinator. Bij de combinatoren zonder traits moeten nog zes methodes geïmplementeerd worden, in vergelijking met de zeven uit de C++ code[1]. Dat komt door het herwerken van het design.

4.2.2 Omvang van de codebase

De oorspronkelijke C++ code telt ongeveer 3250 lijnen code zonder voorbeelden. Onze Scala implementatie telt er ongeveer 1900 inclusief de koppeling met JaCoP. De koppeling neemt hiervan 215 lijnen voor zijn rekening. Het gaat hier om fysische lijnen. Aan het begin van deze thesis hebben we ervoor gekozen een lijn Scala code maximaal 80 karakters breed te maken. Dit is te weinig: naast het feit dat de meeste objectgeoriënteerde talen langere lijnen opleveren dan talen uit het structurele paradigma, heeft Scala voor constructors en velddefinities een zeer hoge informatiedichtheid. Met andere woorden, er moet veel op een logische lijn komen. Daarenboven zorgt mijn persoonlijke keuze voor lange identifiers, in tegenstelling tot die in de oorspronkelijke code in C++, er nog meer voor dat deze getallen slechts een indicatie geven van de grootteorde. Jammer genoeg zijn op het moment van schrijven geen tools beschikbaar voor het tellen van logische lijnen code die Scala ondersteunen.

Toch moeten we toegeven dat we een forsere totale codereductie hadden verwacht. We kunnen verschillende oorzaken aanwijzen waarom dit niet het geval is:

- Solveronafhankelijkheid vraagt om een complexer design met meer klasses en types dan nodig zijn een systeem dat slechts gemakkelijk kan samenwerken met één enkele constraint solver, zoals het systeem uit de oorspronkelijke C++ code.
- Het toevoegen van een gebruiksvriendelijke domeinspecifieke taal vraagt extra code in vergelijking met een systeem waar gebruiksvriendelijkheid totaal ontbreekt.
- Door het ontbreken van vele lusconstructies zoals ze voorkomen in de meeste imperatieve talen, is het in Scala soms moeilijk om complexe combinatoren zoals *and*, *restart* of *if* te schrijven. Het is niet altijd voor de hand liggend om lussen die bij deze combinatoren voorkomen om te zetten naar een lus in functionele stijl.

[1] In de oorspronkelijke C++ code was er nog een achtste methode die enkel gebruikt wordt voor parallelle werking. Deze hebben we niet meegeteld om een eerlijke vergelijking met onze sequentiële code toe te laten.

Uit bovenstaande lijst blijkt duidelijk dat de toename van de codeomvang zich situeert in de backend van het systeem. Het is echter veel nuttiger te kijken naar dat deel waar de gebruikers meest in contact mee zullen komen. Op dat vlak hebben we wel degelijk zeer goede resultaten geboekt. Het vraagt namelijk veel minder code om een gemiddelde combinator toe te voegen, om ze te combineren in nieuwe heuristieken, en vooral ook om *constraint satisfaction* problemen met die heuristieken te gaan oplossen. Bijvoorbeeld: een oplossing voor het all-interval probleem kan je implementeren in ongeveer dertig lijnen Scala-code, terwijl het je er in C++ zeventig kost (althans met het C++ systeem waarop wij ons hebben gebaseerd).

4.2.3 Performantie

We hebben enkele performantiemetingen uitgevoerd om ons systeem te vergelijken met een meer traditionele aanpak. De problemen die we hiervoor hebben gebruikt zijn het „all interval"-probleem en dat van het „Golomb lineaal". We hebben de probleemgrootte laten toenemen en hierbij telkens meerdere metingen verricht.

We hebben zowel metingen verricht met het uitprinten van oplossingen, als zonder printen. De tijd voor het printen is niet te verwaarlozen: voor het „all interval"-probleem was in ons bereik van parameterwaarden de tijd met printen voor een zekere argumentgrootte n ruwweg gelijk aan de tijd om het probleem met grootte $n + 1$ op te lossen zonder printen. In hetgeen volgt gaan we uit van de pure rekentijd.

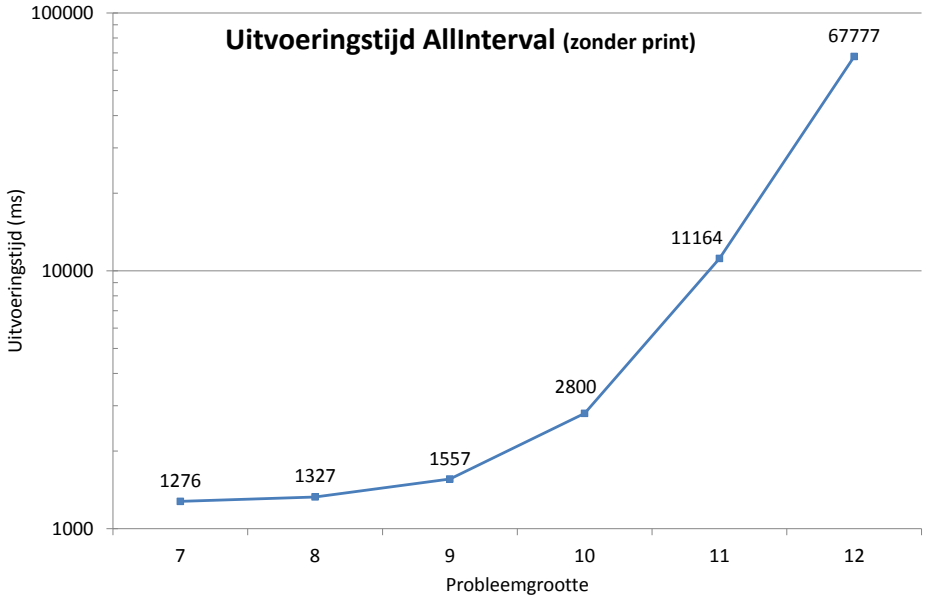
Algemene koppeling Voor het „all interval"-probleem hebben we metingen verricht met grootte van vijf tot en met twaalf. De koppeling die hier gebruikt werd is de algemene koppeling die werkt voor elke strategie voor het doorlopen van de boom. Deze kunnen we echter weinig performant veronderstellen omdat er voortdurend naar de rootspace wordt gebacktrackt. Argumentgroottes kleiner dan zeven zijn niet relevant. Bij grootte elf duurt het oplossen ongeveer 11 seconden. Op dat punt begint het de uitvoeringstijd te tellen. Bij grootte twaalf duurt het reeds meer dan een minuut (zie Figuur 4.1 en Tabel 4.1).

Tabel 4.1: Gemiddelde performantie (ms) voor het "all interval"-probleem.

Grootte	Algemene koppeling	Performante koppeling	JaCoP Scala
7	1.276,1	1.265,1	
8	1.327,5	1.295,2	
9	1.557,5	1.365,1	
10	2.800,0	1.635,4	379,7
11	11.164,0	2.597,3	1.185,3
12	67.777,1	8.273,5	5.124,8
13		44.143,9	

We hebben de performantie van dit probleem ook gemeten op een traditionele implementatie waarbij we gebruik maken van de JaCoP constraint solver met bindings voor Scala. Deze implementatie is veel performanter: voor grootte elf duurt het gemiddeld nog maar 1085 ms. Maar ook hier gaat deze tijd snel de hoogte in: bij grootte twaalf is de gemiddelde uitvoeringstijd al 5124 ms. We kunnen concluderen dat onze implementatie met deze koppeling zo traag is dat ze niet bruikbaar is voor het oplossen van een realistisch probleem. Performantie was dan ook niet de focus van deze thesis.

Ook voor het probleem van het Golomb-lineaal is onze uitvoeringssnelheid drastisch traag (Figuur 4.2 en Tabel 4.2). Hierbij gebruiken we een complexere strategie dan bij „all interval",



Figuur 4.1: Performantie voor het „all interval" probleem.

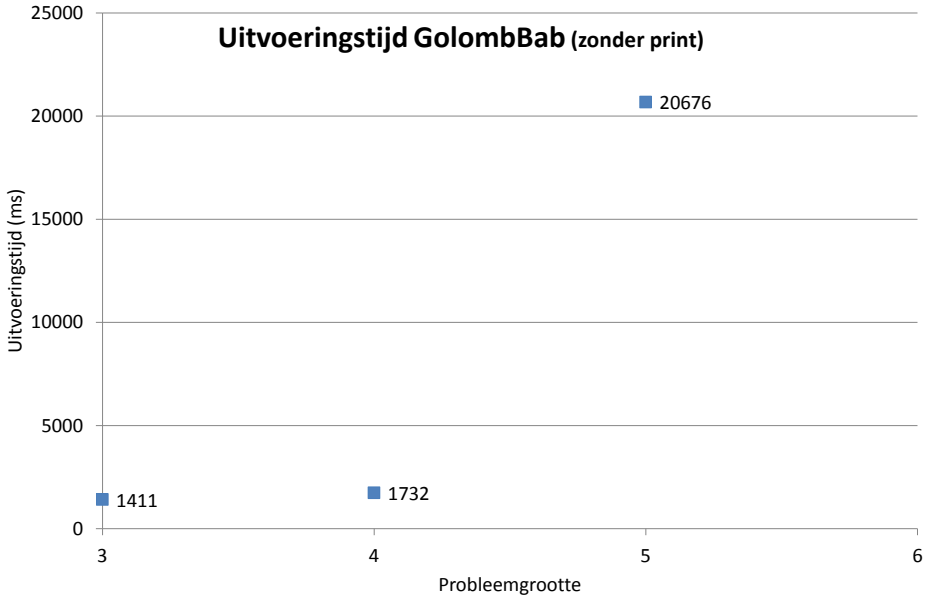
namelijk een *branch-and-bound* combinator. We hebben metingen verricht met argumentgroottes van 3 tot en met 5. De precieze uitvoeringstijd van argumentgrootte zes is niet relevant omdat deze toch niet praktisch is, maar om het exponentiële karakter te illustreren willen we toch even vermelden dat we het proces maar hebben stopgezet na twee uur rekenen.

Tabel 4.2: Gemiddelde performantie (ms) voor het Golombprobleem.

Grootte	Algemene koppeling	Performante koppeling	JaCoP Scala
3	1.411,2	1.274,1	96,6
4	1.732,3	1.393,5	89,5
5	20.676,1	2.009,6	97,3
6			112,8
9			1.022,3
10			6.900,3
11			128.505,5

Ter vergelijking: een argumentgrootte elf levert geen enkel probleem op voor de traditionele aanpak met Scala binding (gemiddeld 128505.5 ms, i.e. ongeveer twee minuten). Toch zullen we niet veel verder meer raken als we kijken naar de vorige metingen: argumentgrootte 10 duurde slechts gemiddeld 6900 ms.

Diepte-eerstkoppeling Specifiek voor het geval waarin de zoekboom wordt doorlopen volgens de diepte-eerststrategie hebben we ook een koppeling geschreven waarbij we voor het construeren van de space op zekere diepte niet steeds beginnen vanaf de root space. Voor deze koppeling kunnen we dan ook een veel betere performantie verwachten. Voor het „all interval" probleem is dit zeker het geval (Figuur 4.3) bij argumentgroottes vanaf elf. De uitvoeringstijd is nu iets minder dan drie seconden, tegenover eerder elf seconden. Het probleem van grootte



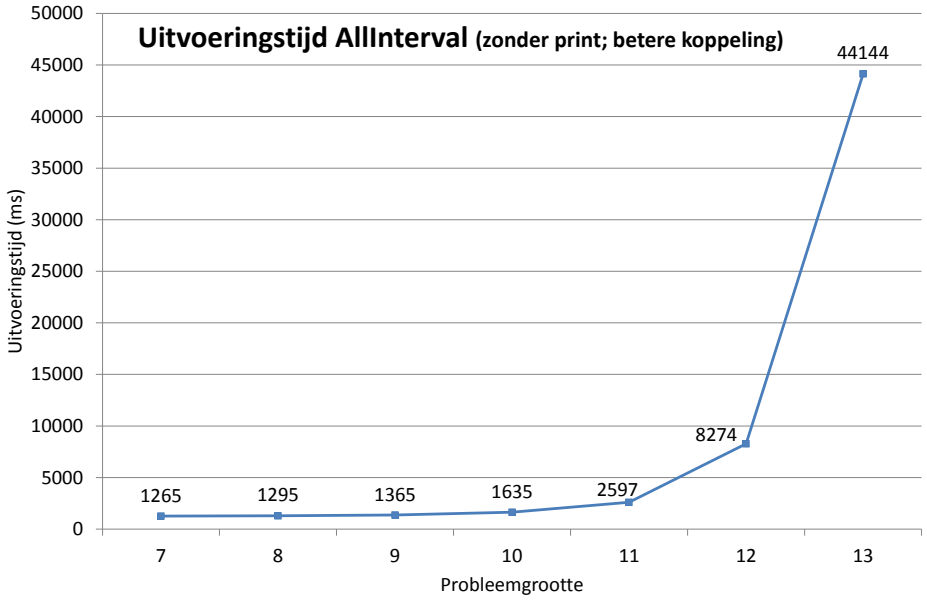
Figuur 4.2: Performantie voor het probleem van de Golomblinealen.

dertien is nu ook op te lossen in ongeveer 44 seconden. Voor kleinere problemen valt de winst minder goed op. Voor argumenten elf tot en met dertien merken we dat de oplossingstijd in dezelfde grootteorde ligt als die bij een traditionele oplossing. Voor het Golombprobleem valt de verbeterde performantie vooral op bij argumentgrootte vijf. Voor de kleinere instanties van het probleem is de winst eerder beperkt. Ondanks de verbetering is het programma nog steeds grootteordes trager dan bij een traditionele oplossing.

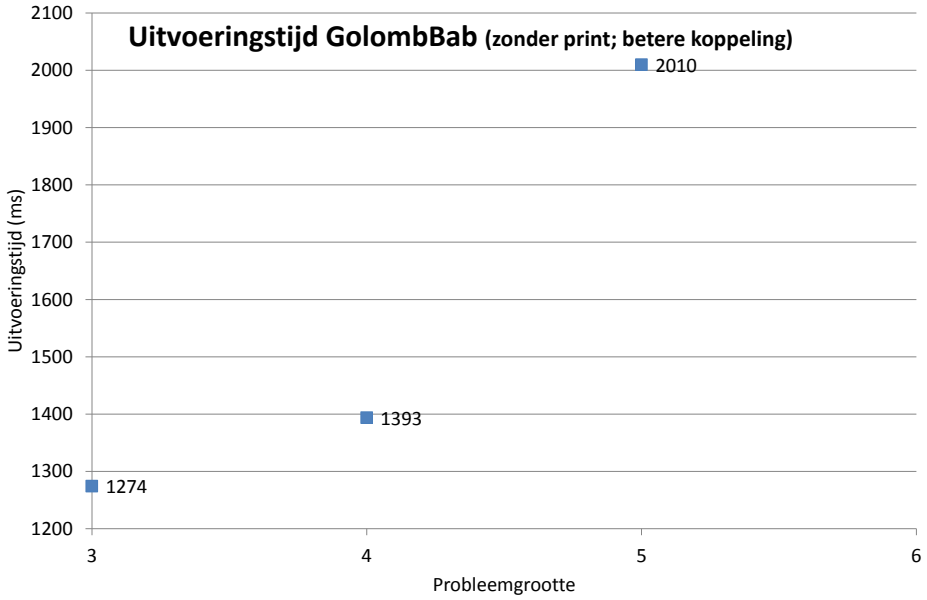
Vergelijking met Java en C++ De moeilijkheid ligt dus voor een belangrijk deel in het voorzien van een goede koppeling met de constraint solver. Voor het meten van de invloed van Scala op het resultaat, hebben we ook enkele metingen gedaan voor het Golomb probleem in Java (waarin JaCoP is ontwikkeld). We bemerken dat de Scalabindings voor de JaCoP constraint solver toch voor enige overhead zorgen (Figuur 4.5). We hebben er dus goed aan gedaan voor onze koppeling geen gebruik te maken te maken van deze bindings. Het is namelijk mogelijk rechtstreeks te werken met de Java-klassen.

Wanneer we onze resultaten vergelijken met die over de oorspronkelijke implementatie in C++, merken we dat de onze implementatie grootteordes trager is dan deze in C++ [20]. We gebruiken echter een andere constraint solver. De paper stelt ook dat combinatoren zonder runtime overhead werken ten opzichte van een traditionele implementatie met de Gecode constraint solver. Voor onze implementatie is dat niet het geval. Voor het „all interval" probleem, waarvoor we een eenvoudige *integer search* combinator gebruiken, liggen de resultaten al in dezelfde grootteorde. Mits wat aandacht voor de performantie, hopen we voor al te diepe combinatorbomen een fair resultaat te kunnen bereiken.

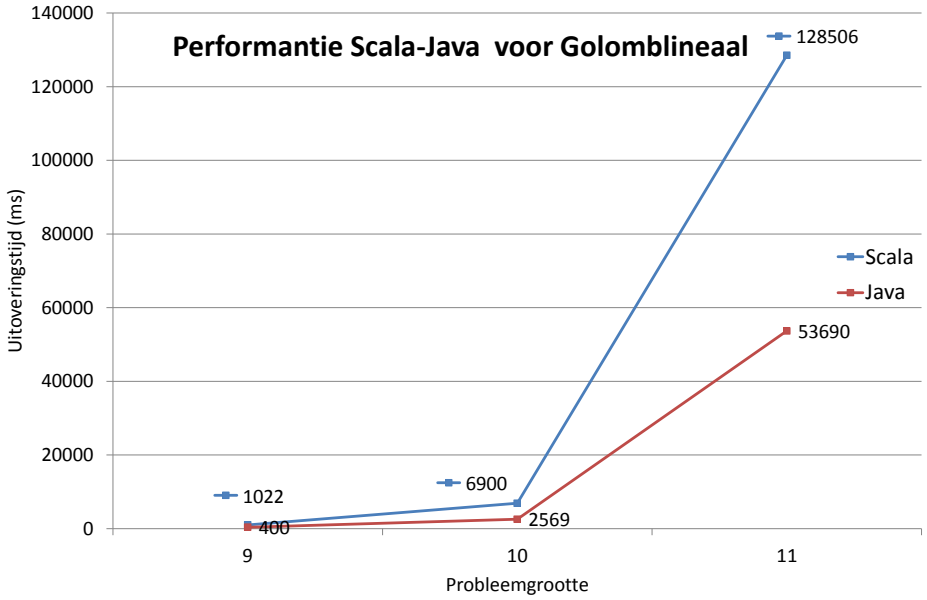
Onze performantieresultaten zijn te verklaren door de hoeveelheid indirecties die nodig zijn in het solverafhankelijke deel om onze constraints en variabelen om te zetten naar solverspecifieke klassen en door de casts die hiermee gepaard gaan. We kunnen dit aantal proberen te verminderen, eventueel door een deel van de solveronafhankelijkheid anders te implementeren. We



Figuur 4.3: Performantie voor het „all interval" probleem met specifieke koppeling.



Figuur 4.4: Performantie voor het probleem van de Golomblinealen met specifieke koppeling.



Figuur 4.5: Performantie van JaCoP in Java en met Scalabindings.

merken ook dat diep geneste combinatoren tot problemen leiden. Dit vraagt om verbeteringen in de Scala-compiler.

Hoofdstuk 5

Conclusies en toekomstig werk

In dit werk hebben we de implementatie besproken van modulaire heuristieken in Scala. Daarbij hebben we niet alleen een nieuw design ontwikkeld, maar hebben we ook de positieve en negatieve kanten van Scala leren kennen. We bespreken eerst het design; in Sectie 5.2 hebben we het specifiek over de taal Scala zelf. We eindigen met enkele suggesties voor toekomstig werk.

5.1 Een elegante implementatie

Vertrekkend van een implementatie in C++, hebben we het design aangepast. Door het gebruik van traits zijn we erin geslaagd vaak terugkerende patronen in de code van de combinatoren te abstraheren, en aldus codeduplicatie te vermijden. We hebben gezien dat dit mogelijk is voor een groot aantal combinatoren. Alleen voor de allermoeilijkste combinators die twee of meer subcombinatoren hebben, helpt deze techniek niet om de hoeveelheid implementatiewerk te verminderen.

Door de combinatie van traits in de juiste volgorde, bij sommige combinatoren tot drie stuks, zijn we erin geslaagd de hoeveelheid nog zelf te schrijven code te herleiden tot een minimum. Bij zes op de 21 geïmplementeerde basiscombinatoren, moeten we nog enkel de entermethode, of een vervangende methode implementeren. Deze bepaalt het gedrag van de combinator immers het meest.

In die gevallen waar patronen eerder typerend zijn voor één enkele combinator, over zijn verschillende methodes heen, dan over verschillende combinatoren, helpen traits niet. Hier kunnen we dankbaar gebruik maken van Scala's mogelijkheid zelf controleabstracties te schrijven. In het bijzonder is dit te danken aan de anonieme functies die in de taal worden ondersteund.

Doorheen het verloop van de thesis, hebben we ontdekt dat traits niet enkel van pas komen bij de combinators zelf. Ook in alle ondersteunende klassen, zoals die voor condities en variabelen, kunnen ze succesvol worden aangewend. Dit overervingssysteem gaat vaak hand in hand met het gebruik van abstracte typemembers en bovengrenzen.

We zijn erin geslaagd het ontwerp zo aan te passen om casts zoveel mogelijk te vermijden. In het solveronafhankelijk deel van de code komen er alvast geen voor. Ons systeem moet gekoppeld worden aan een constraint solver, maar de keuze staat in principe vrij, mits de constraint solver aan enkele lichte voorwaarden voldoet. De gebruikersgedefinieerde code moet geenszins aangepast worden wanneer een andere solver gekoppeld wordt. Omwille van zijn implementatie in Java en goede documentatie, hebben wij ervoor gekozen de koppeling te maken met de JaCoP constraint solver.

In de praktijk hangt de performantie van het systeem voor een belangrijk deel af van de manier waarop men de koppeling realiseert. Daarom kiest men liefst voor een constraint solver waar

men zelf zoveel mogelijk kan aan wijzigen, hetzij via een goede API, hetzij via een licentie die toelaat aan de code te sleutelen.

Het modelleren van nieuwe zoekheuristieken kan erg beknopt gebeuren. Door het gebruik van case klassen en gebruikersgedefinieerde operatoren lijkt het alsof de combinatoren ingebakken zijn in de taal Scala. We hebben met andere woorden een domeinspecifieke taal ontwikkeld. Een klein minpunt hierbij is dat we de precedentie van de operatoren niet kunnen aanpassen.

Om het systeem praktisch bruikbaar te maken, moet er nog heel wat verbeteren aan de performantie. Het gebruik van een profiler is dan ook aangewezen om uit te maken welke elementen van het design voor de grootste vertragingen zorgen.

Samenvattend hebben we de volgende punten bereikt:

- Door het gebruik van de geavanceerde overervingsvorm traits hebben we codeduplicatie vermeden. Deze duplicatie was niet te vermijden met standaard enkelvoudige overerving. Tegelijk hebben de problemen die gepaard gaan met meervoudige overerving vermeden.
- Scala is een taal met garbage collection, waardoor we — in tegenstelling tot de oorspronkelijke C++ code — niet meer zelf instaan voor het geheugenbeheer. Dit draagt sterk bij tot het verlagen van de drempel om een nieuwe zoekheuristiek te implementeren.
- Door het schrijven van boilerplate code overbodig te maken, en door het hoge abstractieniveau van Scala, kan een nieuwe zoekheuristiek sneller, dus goedkoper, en betrouwbaarder geïmplementeerd worden. Als gevolg is het mogelijk meer onderzoek te doen naar de meest geschikte heuristiek voor het oplossen van een constraint satisfaction probleem. Aangezien deze problemen vaak voorkomen in allerlei praktische situaties, is het zo goed mogelijk kunnen oplossen van deze problemen van cruciaal belang.
- Dankzij de traits is een expliciete implementatie van mixin inheritance niet langer nodig. We kunnen de zoekspecificaties statisch samenstellen.
- Mits voldaan aan enkele minimale vereisten, kan de code zonder veel moeite gekoppeld worden aan om het even welke constraint solver.
- De solveronafhankelijke code bevat geen casts. Dit is een indicatie dat er in ons design geen grote fouten zijn gemaakt.
- We hebben terugkerende patronen uit de code te weten elimineren. Hierdoor is onze code gemakkelijker te onderhouden.

5.2 Scala is veelzijdig en flexibel

Scala in het algemeen is een interessante taal. Haar typeinferentie en hoge niveau maken het programmeren aangenamer, want minder typewerk voor triviale zaken. Naast de geavanceerde overerving, wil ik in het bijzonder ook wijzen op de concepten uit functioneel programmeren die in deze taal zijn geïntegreerd: onwijzigbare („immutable”) variabelen, currying, anonieme functies, zelfgedefinieerde controleabstracties, algemene patroonmatching, en dergelijke, blijken toevallig de geknipte oplossing voor heel wat problemen.

Daarentegen kan ik me voorstellen dat de langere leercurve en zaken als het ontbreken van een imperatieve „for” lus maken dat Scala misschien niet voor iedereen is weggelegd. Sommige erg ingewikkelde lussen zijn ook net iets eenvoudiger uit te drukken op een imperatieve manier, terwijl voor andere de functionele manier veel korter en duidelijker is.

Scala's typesysteem is erg uitgebreid en flexibel. Het is echter niet altijd zo gemakkelijk om te begrijpen waarom een bepaald ontwerp niet typecheckt. Door de flexibiliteit denkt men soms abusievelijk te maken te hebben met een dynamisch getypeerde taal en bedenkt men — als vanzelf — oplossingen die enkel in die talen mogelijk zijn. Zoals bij alle programmeertalen helpt ervaring om deze fouten te vermijden.

We hopen dat de ontwikkelaars van Scala in een toekomstige versie aandacht zullen besteden aan het verbeteren van de ondersteuning voor domeinspecifieke talen. Een belangrijke tekortkoming is het feit dat de precedentie van de operatoren niet gewijzigd kan worden. Er is alvast zeer veel nuttig werk verricht, getuige hiervan het zeer uitgebreide en nuttige collectieframework.

5.3 Toekomstig werk

Modulaire zoekheuristieken laten toe effectief eigen zoekheuristieken te definiëren. Gezien het belang van die heuristieken in tal van problemen, verwachten we dan ook dat dit modulaire ontwerp alleen maar aan belang zal toenemen. Want een toename in de rekencapaciteit van computers, zal ook leiden tot de vraag grotere problemen aan te pakken. Voor wat betreft het systeem zelf, zou het interessant zijn de koppeling te maken met een visualisatietool als Cp-Viz [24, 8]. Op die manier zouden we de actieve combinator en de zoekboom kunnen visualiseren, en hieruit eventueel optimalisaties van de zoekheuristiek kunnen vinden.

Performantieverbetering is nog duidelijk nodig voor onze code. Hierbij willen we natuurlijk zo weinig mogelijk raken aan de elegantie van het design, maar vooral niet aan de interface naar onze gebruikers toe. Het succes van een bibliotheek heeft immers voor een groot deel te maken met gebruiksvriendelijkheid.

Ten derde zou het nuttig kunnen zijn om zekere automatische transformaties uit te voeren op heuristieken. Denken we maar aan twee binaire „and"-combinatoren genest in elkaar. Deze zouden we kunnen vervangen door één enkele ternaire „and"-combinator.

Tot slot hopen we op meer ingebouwde ondersteuning binnen de verschillende constraint solvers. De Gecode FlatZinc interpreter is hiervan alvast een goed voorbeeld [9].

Bijlage A

Precedentie van de operatoren in Scala

In deze appendix geven we de precedentie van de operatoren in Scala, zoals te vinden in [16]. De precedentie wordt bepaald door het eerste symbool, tenzij de operator eindigt op een gelijkheidsteken, en geen vergelijkingsoperator is. In dat geval spreken we over *assignatieoperatoren*. Deze hebben de laagste precedentie. Tabel A.1 geeft de volgorde van hoog naar laag.

Tabel A.1: Precedentie van de operatoren in Scala in dalende volgorde. Karakters op dezelfde lijn hebben gelijke precedentie.

Alle speciale karakters die niet hieronder staan vermeld
`* / %`
`+ -`
`:`
`= !`
`< >`
`&`
`^`
`|`
Alle letters
Alle assignatieoperatoren

Bijlage B

Scala's flexibele importdirectieven

In deze bijlage schetsen we de mogelijkheden en de syntax van het importmechanisme van Scala. De belangrijkste verschillen met Java zijn [16]:

- `import`-clauses mogen overal voorkomen
- ze kunnen refereren naar objecten en naar hele pakketten
- ze laten toe het geïmporteerde of onderdelen ervan te hernoemen
- ze laten toe onderdelen van het geïmporteerde te verbergen

In Codefragment B.1 tonen we enkele clauses die dit zullen verduidelijken. In de eerste lijn importeren we de klassen `IntVar` en `BoolVar`. In de tweede regel importeren we alles uit het pakket `searchCombinators`. Iets gelijkaardigs doen we in de volgende regel, maar we vereisen een prefix `JC` voor elk van de geïmporteerde leden. In regel vier hernoemen we enkele klassen, en importeren we de rest niet. Daaronder importeren ook de rest van het pakket onder de oorspronkelijke naam. Tot slot importeren we alles uit pakket `vim` behalve de — voor ons onnuttige[1] — klasse `EmacsEmulation` (hernoemen naar de „don't care" waarde `_`).

Codefragment B.1: Voorbeelden van flexibele importclauses.

```
1 import searchCombinators.{IntVar, BoolVar}
  import searchCombinators._
3 import JaCoP.{constraints ⇒JC}
  import JaCoP.core.{Store ⇒JStore, FailException ⇒JFailException }
5 import JaCoP.core.{Store ⇒JStore, IntVar ⇒JIntVar, _}
  import vim.{EmacsEmulation ⇒_, _}
```

[1] Hoewel we ook Emacs erkennen als een bijzonder krachtige tool in de handen van de juiste persoon...

Bijlage C

Detailgegevens effectiviteit van traits

In deze appendix geven we de details over de effectiviteit van traits. We tonen welke boodschappen nog door elke basiscombinator moeten worden geïmplementeerd (Tabel C.1 en C.2). In de tabellen hebben we enkel de relevante combinatoren weergegeven. Daarnaast tonen we waar elke trait precies gebruikt wordt (Tabel C.3 en C.4).

Tabel C.1: Expliciete implementatie van berichten per combinator: standaardberichten

	Bericht					
Combinator	**enter**	**child**	**next**	**init**	**copy**	**stat**
AssignMed						
BoolSearch	x	x				
Fail	x					
FailToPrune	x					
If	x	x	x	x	x	x
Incr						
IntAssign						
IntSearch	x	x	x	x		
LetAssign						
Luby						
Mult						
Post	x					
Print						
Prune	x					
Restart	x	x	x			
SeqAnd	x	x	x	x	x	x
SeqOr	x	x	x	x	x	x
DepthCount				x		x
NodeCount	x					
FailCount	x					
DiscrCount				x		x
Aantal combinators						
21	12	6	5	6	3	5

Tabel C.2: Expliciete implementatie van berichten per combinator: vervangende berichten

Combinator	enterSideEffect	onEnterSolution	onChild
AssignMed	x		
Incr	x		
IntAssign	x		
LetAssign	x		
Luby	x		
Mult	x		
Print		x	
DepthCount			x
DiscrCount			x
Aantal combinators			
21	6	1	2

Tabel C.3: Gebruik van traits in de verschillende combinatoren (deel 1).

Combinator	EmptyCombinator	NoChildMethod	DoAndSucceedEnter
AssignMed	x	x	x
BoolSearch	x		
Fail	x	x	
Incr	x	x	x
IntAssign	x	x	x
IntSearch	x		
LetAssign	x	x	x
Luby	x	x	x
Mult	x	x	x
Post			
Print			
Prune	x	x	
Aantal combinators			
21	10	8	6

Tabel C.4: Gebruik van traits in de verschillende combinatoren (deel 2).

Combinator	SubCombinator	OnEnterSolution	CopyItem	OnChild
FailToPrune	x			
If				
Post	x			
Print	x	x		
Prune				
Restart	x		x	
DepthCount	x			x
NodeCount	x			
FailCount	x			
DiscrCount	x		x	x
Aantal combinators				
21	8	1	2	2

Bibliografie

[1] HaskellWiki on heterogenous collections (13 mei 2012), `http://www.haskell.org/haskellwiki/Heterogenous_collections`

[2] The HList package on Hackage (13 mei 2012), `http://hackage.haskell.org/package/HList`

[3] Wikipedia over Golomb linealen (15 maart 2012), `http://en.wikipedia.org/wiki/Golomb_ruler`

[4] Wolfram MathWorld over Golomb linealen (15 maart 2012), `http://mathworld.wolfram.com/GolombRuler.html`

[5] The Gecode constraint solver website (18 mei 2012), `http://www.gecode.org/`

[6] The JaCoP constraint solver website (18 mei 2012), `http://www.jacop.eu/`

[7] The Scala programming language website (18 mei 2012), `http://www.scala-lang.org/`

[8] Cp-Viz visualization platform website (21 maart 2012), `http://cpviz.sourceforge.net/`

[9] MiniZinc with Search Combinators (21 maart 2012), `http://www.gecode.org/flatzinc.html`

[10] G. Bracha; W. Cook, Mixin-based inheritance, SIGPLAN Not., 25(10), (1990), 303–311

[11] E. Gamma; R. Helm; *et al.*, Design Patterns: Elements of Reusable Object-Oriented Software, 1e druk, Addison-Wesley, Massachusetts (1994)

[12] A. Kelley; I. Pohl, A Book on C: Programming in C, 4e druk, Addison-Wesley Professional, Massachusetts

[13] O. Kiselyov; R. Lämmel; *et al.*, Strongly typed heterogeneous collections, Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004), 96–107

[14] B. Meyer, Object-Oriented Software Construction, Prentice Hall (2000)

[15] N. Nethercote; P. J. Stuckey; *et al.*, MiniZinc: towards a standard cp modelling language, Proceedings of the 13th international conference on Principles and practice of constraint programming, CP'07, Springer-Verlag, Berlin, Heidelberg (2007), 529–543

[16] M. Odersky; L. Spoon; *et al.*, Programming in Scala, 2e druk, Artima Press, California (2010)

[17] M. Odersky; M. Zenger, Scalable component abstractions, SIGPLAN Not., 40(10), (2005), 41–57

[18] M. Schönfinkel, Über die Bausteine der mathematischen Logik, Mathematische Annalen, 92, (1924), 305–316

[19] T. Schrijvers; G. Tack; *et al.*, Search Combinators, persoonlijk overhandigd – ongepubliceerd

[20] T. Schrijvers; G. Tack; *et al.*, Search combinators, Proceedings of the 17th international conference on Principles and practice of constraint programming, CP'11, Springer-Verlag, Berlin, Heidelberg (2011), 774–788

[21] T. Schrijvers; M. Triska; *et al.*, Tor: Extensible Search with Hookable Disjunction, persoonlijk overhandigd – ongepubliceerd

[22] C. Schulte; G. Tack; *et al.*, Modeling and Programming with Gecode, 3.6.0e druk, `http://www.gecode.org/doc/3.6.0/MPG.pdf` (2011)

[23] N. Shärli; S. Ducasse; *et al.*, Traits: Composable Units of Behavior, Tech. rapp. (2002)

[24] H. Simonis; P. Davern; *et al.*, A generic visualization platform for cp, Proceedings of the 16th international conference on Principles and practice of constraint programming, CP'10, Springer-Verlag, Berlin, Heidelberg (2010), 460–474

[25] P. Van Roy; S. Haridi, Concepts, Techniques, and Models of Computer Programming, MIT Press, Cambridge (2004)

Lijst van figuren

Lijst van tabellen