*9*

## Constraints

*Constraints are Boolean expressions that must be true for every program expressed with a specific language. Together with type systems, which are discussed in the next chapter, they ensure the static semantics of a language. This chapter introduces the notion of constraints, some considerations regarding languages suitable for expressing constraints, and provides examples with our tools.*

As we explained in the DSL Design part of the book, not all programs that conform to the structure (grammar, AS, meta model) of a language are valid. Language definitions include further restrictions that cannot be expressed purely by structure. Such additional restrictions are typically called constraints.

|  |  |
| --- | --- |
| An error message is reported if the expression evaluates to **false** ("**expr** does not hold!"). Constraints are typically associated a particular language concept ("for each instance of con- |  |
| cept **C**, **expr-with-C** must hold")2. There are two major kinds of constraints we can distinguish: well-formedness and type |  |

Constraints are Boolean conditions that have to evaluate to **true** in order for the model to be correct ("does **expr** hold?")1.

systems. Examples for well-formedness constraints include:

* Uniqueness of names in lists of elements (e.g., functions in a namespace).
* Every non-start state of a state machine has at least one incoming transition.
* A variable is defined before it is used (statement ordering).

Type system rules are different in that they verify the correctness of types in programs, e.g., they make sure you don’t as-

sign a **float** to an **int**. In expression languages particularly, type calculation and checking can become quite complicated, and therefore warrant special support. This is why we distinguish between constraints in general (covered in this chapter) and type systems (which we cover in the next chapter).

Constraints can be implemented with any language or framework that is able to query a model and report errors to the user. To make expressing constraints efficient3, it is useful if the lan-

guage has the following characteristics:

* It should be able to effectively navigate and filter the model. Support for path expressions (as in **aClass.operations. arguments.type** as a way to find out the types of all arguments of all operations in a class) is extremely useful.
* Support for higher-order functions is useful, so that one can write generic algorithms and traversal strategies.
* A good collections language, often making use of higherorder functions, is very useful, so it is easily possible to filter collections, create subsets or get the set of distinct values in a list.
* Finally, it is helpful to be able to associate a constraint declaratively with the language concept (or structural pattern) for whose instances it should be executed.

Here is an example constraint written in a pseudo-language:

|  |
| --- |
| **constraint for**:  Class **expression**:  **this**.operations.arguments.type.filter(ComplexNumber).isNotEmpty &&  !**this**.imports.any(i|i.name == "ComplexNumberSupportLib") **message**:  "class "+**this**.name+" uses complex numbers, "+  "so the ComplexNumberSupportLib must be imported" |

Some kinds of constraints require specialized data structures to be built or maintained in sync with the program. Examples include dead code detection, missing returns in some branches of a method’s body, or read access to an uninitialized variable. To be able to find these kinds of errors statically, a dataflow graph has to be constructed from the program. It models the various execution paths through a (part of a) program. Once a dataflow graph is constructed, it can be used to check whether there exists a path from program start to a variable read without coming across a write to the same variable. We show an example of the use of a data flow graph in the MPS example

(Section 9.2).

### 9.1 Constraints in Xtext

Just like scopes, constraints are implemented in Java or any other JVM language4. Developers add methods to a validator

class generated by the Xtext project wizard. In the end, these validations plug into the EMF validation framework5.

A constraint checking method is a Java method with the following characteristics: it is public, returns **void**, can have an arbitrary name, it has a single argument of the type for which the check should apply, and it has the **@Check** annotation. For example, the following method is a check that is invoked for all instances of **CustomState** (i.e. not for start states and background states). It checks that each such state can actually be reached by verifying that it has incoming transitions (expressed via a **ChangeStateStatement**):

|  |
| --- |
| @Check(CheckType.NORMAL) **public void** checkOrphanEndState( CustomState ctx ) {  CoolingProgram coopro = Utils.ancestor(ctx, CoolingProgram.**class**); TreeIterator<EObject> all = coopro.eAllContents(); **while** ( all.hasNext() ) { EObject s = all.next();  **if** ( s **instanceof** ChangeStateStatement ) {  ChangeStateStatement css = (ChangeStateStatement) s; **if** ( css.getTargetState() == ctx ) **return**; } }  error("no transition ever leads into this state",  CoolingLanguagePackage.eINSTANCE.getState\_Name());  } |
| The method retrieves the cooling program that owns the **ctx** state, then retrieves all of its descendants and iterates over them. If the descendant is a **ChangeStateStatement** and its **targetState** property references the current state, then we return: we have found a transition leading into the current state. If we don’t find one of these, we report an error. An error report contains the error message, a severity (**INFO**, **WARNING**, | | |  |
| **ERROR**), the element to which it is attached6, as well as the par- | | |  |
| ticular feature[[1]](#footnote-1) of that element that should be highlighted. The | | |  |

**CheckType.NORMAL** in the annotation defines when this check should run:

* **CheckType.NORMAL**: run when the file is saved.
* **CheckType.FAST**: run after each model change (more or less after each keypress).
* **CheckType.EXPENSIVE**: run only if requested explicitly via the context menu.

Note that neither Xtext nor any of the other tools supports impact analysis by default. Impact analysis is a strategy for finding out whether a particular constraint can potentially be affected by a particular change, and only evaluating the constraint if it can. Impact analysis can improve performance if this analysis is faster than evaluating the constraint itself. For local constraints this is usually not the case. Only for non-local constraints that cover large parts of the model (and possibly require loading additional fragments), is impact analysis important. Xtext uses a pragmatic approach in the sense that these constraints must be marked as **EXPENSIVE** by the user and run only on request (over lunch, during nightly build). As an example, let us get back to the example about orphan states. The implementation of the constraint checks orphan-ness separately for each state. In doing so, it gets all descendants of the cooling program *for each state*. This can be a scalability problem for larger programs. To address this issue, one would write *a single constraint* for the whole cooling program that identifies all orphan states in one or maybe two scans through the program. This constraint could then be marked as **EXPENSIVE** as programs get really big8.

|  |
| --- |
| **checking rule** stateUnreachable {  **applicable for concept** = State **as** state **do** { **if** (!state.initial && state.ancestor<Statemachine>.  descendants<Transition>.  where({~it => it.target == state; }).isEmpty) {  **error** "orphan state - can never be reached" -> state;  }  }  } |

|  |  |
| --- | --- |
|  |  |
| *9.2 Constraints in MPS*  *9.2.1 Simple Constraints*  MPS’ approach to constraints is very similar to Xtext’s9. The main difference is that the constraint is written in BaseLanguage, an extended version of Java that has some of the features that makes constraints more concise. Here is the code for the same *state unreachable* constraint, which we can make use of in the state machines extension to C: |  |

Currently there is no way to control when a constraint is run10,

it is decided based on some MPS-internal algorithm which tracks changes to a model and reevaluates constraints as neces-

sary. However, pressing **F5** in a program or explicitly running the model checker forces all constraints to be reevaluated.

#### 9.2.2 Dataflow

As we have said earlier, dataflow analysis can be used to detect dead code, null access, unnecessary **if**s (because it can be shown statically that the condition is always true or false) or read-before-write errors. The foundation for data flow analysis is the *data flow graph*. This is a data structure that describes the flow of data through a program’s code. Consider the following example:

|  |
| --- |
| **int** i = 42; j = i + 1;  someMethod(j); |
| The **42** is "flowing" from the **init** expression in the local variable declaration into the variable **i** and then, after adding **1**, into **j**, and then into **someMethod**. Data flow analysis consists of two tasks: building a data flow graph for a program, and then performing analysis on this data flow graph to detect problems in the program.  MPS comes with predefined data structures for representing data flow graphs, a DSL for defining how the graph can be derived from language concepts (and hence, programs) and a set |  |
| of default analyses that can be integrated into your language11. |  |
| We will look at all these ingredients in this section. |  |

*Building a Data Flow Graph* Data flow is specified in the *Dataflow* aspect of language definitions. There you can add data flow builders (DFBs) for your language concepts. These are programs expressed in MPS’ data flow DSL that build the data flow graph for instances of those concepts in programs.

Here is the DFB for **LocalVariableDeclaration**.

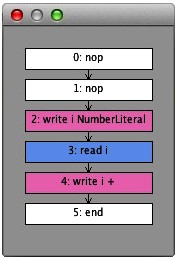
|  |
| --- |
| **data flow builder for** LocalVariableDeclaration {  (node)->**void** { **if** (node.init != **null**) { **code for** node.init **write** node = node.init  } **else** { **nop**  }  }  } |

If the **LocalVariableDecaration** has an **init** expression (it is optional!), then the DFB for the **init** expression has to be executed using the **code for** statement. Then we perform an actual data flow definition: the **write node = node.init** spec-

|  |  |
| --- | --- |
| ifies that write access is performed on the current node. The statement also expresses that whatever value was in the **init** expression is now in the node itself. If there is no **init** expression, we still want to mark the **LocalVariableDeclaration** node as visited by the data flow builder using the **nop** state- |  |
| ment – the program flow has come across this node12. |  |
| To illustrate a **read** statement, we can take a look at the **LocalVariableRef** expression which read-accesses the variable it references. Its data flow is defined as **read node.var**, where **var** is the name of the reference that points to the referenced variable.  In an **AssignmentStatement**, we first execute the DFB for |  |

|  |
| --- |
| **data flow builder for** AssigmentStatement {  (node)->**void** { **code for** node.rvalue  **write** node.lvalue = node.rvalue }  } |

|  |
| --- |
| **void** trivialFunction() {  int8 i = 10; i = i + 1;  } |

the **rvalue** and then "flow" the **rvalue** into the **lvalue** – the purpose of an assignment:

For a **StatementList**, we simply mark the list as visited and then execute the DFBs for each statement in the list. We are now ready to inspect the data flow graph for the simple function below. Fig. 9.1 shows the data flow graph.

|  |  |
| --- | --- |
| Most interesting data flow analysis has to do with loops and branching. So specifying the correct DFBs for things like **if**, **switch** and **for** is important. As an example, we look at the DFB for the **IfStatement**. We start with the obligatory **nop** to mark the node as visited. Then we run the DFB for the condition, because that is evaluated in all cases. Then it becomes interesting: depending on whether the condition is true or false, we either run the **thenPart** or we jump to where the **else if** parts begin. Here is the code so far: |  |

**nop**

**code for** node.condition

**ifjump after** elseIfBlock // elseIfBlock is a label defined later **code for** node.thenPart

{ **jump after** node }

The **ifjump** statement means that we may jump to the specified label (i.e. we then execute the **else if**s). If not (we just "run over" the **ifjump**), then we execute the **thenPart**. If we execute the **thenPart**, we are finished with the whole **IfStatement**

– no **else if**s or **else** parts are relevant, so we jump after the current node (the **IfStatement**) and we’re done. However, there is an additional catch: in the **thenPart**, there may be a **return** statement. So we may never actually arrive at the **jump after node** statement. This is why it is enclosed in curly braces: this says that the code in the braces is optional, so if the data flow does not visit it, that’s fine (and no *dead code* error is reported).

Let’s continue with the **else if**s. We arrive at the label **elseIfBlock** if the condition was **false**, i.e. the above **ifjump** actually happened. We then iterate over the **elseIf**s and execute their DFB. After that, we run the code for the **elsePart**, if there is one. The following code can only be understood if we know that, if we execute one of the **else if**s, then we jump *after the whole* **IfStatement**. This is specified in the DFB for the **ElseIfPart**, which we’ll illustrate below. Here is the rest of the code for the **IfStatement**’s DFB:

**label** elseIfBlock **foreach** elseIf **in** node.elseIfs {

**code for** elseIf

} **if** (node.elsePart != **null**) {

**code for** node.elsePart

}

We can now inspect the DFB for the **ElseIfPart**. We first run the DFB for the condition. Then we may jump to after that **else if**, because the condition may be false and we want to try the next **else if**, if there is one. Alternatively, if the condition is true, we run the DFB for the body of the **ElseIfPart**. Then two things can happen: either we jump to after the whole

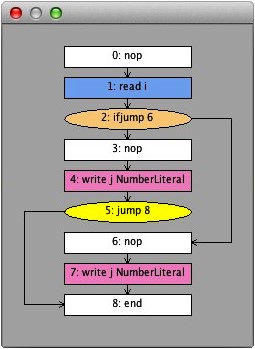
**IfStatement** (after all, we have found an **else if** that is true), or we don’t do anything at all anymore because the current **else if** contains a **return** statement. So we have to use the curly braces again for the jump to after the whole **if**. The code is below, and an example data flow graph is shown in figure

Fig. 9.2.

|  |
| --- |
| **code for** node.condition **ifjump after** node **code for** node.body  { **jump after** node.ancestor<IfStatement> } |

The DFB for a **for** loop makes use of the fact that loops can be represented using conditional branching. Here is the code:

**code for** node.iterator **label** start **code for** node.condition **ifjump after** node



**code for** node.body **code for** node.incr **jump after** start

We first execute the DFB for the **iterator** (which is a subconcept of **LocalVariableDeclaration**, so the DFB shown above works for it as well). Then we define a label **start** so we can jump to this place from further down. We then execute the **condition**. Then we have an **ifjump** to after the whole loop (which covers the case in which the condition is false and the loop ends). In the other case (where the condition is still true) we execute the code for the **body** and the **incr** part of the **for** loop. We then jump to after the **start** label we defined above.

|  |  |
| --- | --- |
| *Analyses* MPS supports a number of data flow analyses |  |
| out of the box13. The following utility class uses the unreach- |  |
| able code analysis: |  |

|  |
| --- |
| **public class** DataflowUtil { **private** Program prog;  **public** DataflowUtil(node<> root) { // build a program object and store it prog = DataFlow.buildProgram(root);  }  **public void** checkForUnreachableNodes() {  // grab all instructions that  // are unreachable (predefined functionality) sequence<Instruction> allUnreachableInstructions =  ((sequence<Instruction>) prog.getUnreachableInstructions());  // remove those that may legally be unreachable sequence<Instruction> allWithoutMayBeUnreachable = allUnreachableInstructions.where({~instruction => !(Boolean.TRUE.equals(instruction.  getUserObject("mayBeUnreachable"))); });  // get the program nodes that correspond // to the unreachable instructions sequence<node<>> unreachableNodes = allWithoutMayBeUnreachable. select({~instruction => ((node<>) instruction.getSource()); });  // output errors for each of those unreachable nodes **foreach** unreachableNode **in** unreachableNodes { **error** "unreachable code" -> unreachableNode; }  }  } |

The class builds a **Program** object in the constructor. **Program**s are wrappers around the data flow graph and provide access to a set of predefined analyses on the graph. We will make use of one of them here in the **checkForUnreachableNodes** method. This method extracts all unreachable nodes from the graph (see comments in the code above) and reports errors for them. To actually run the check, we call this method from a non-typesystem rule for C functions:

|  |
| --- |
| **checking rule** check\_DataFlow {  **applicable for concept** = Function **as** fct **overrides false do** {  **new** DataflowUtil(fct.body).checkForUnreachableNodes();  }  } |

### 9.3 Constraints in Spoofax

Spoofax uses *rewrite rules* to specify all semantic parts of a language definition. In this section, we first provide a primer on rewrite rules. Next we show how they can be used to specify constraints in language definitions14.

|  |  |
| --- | --- |
|  |  |
| *9.3.1 Rewrite Rules*  Rewrite rules are functions that operate on terms, transforming one term to another. Rewrite rules in Spoofax are provided as part of the Stratego program transformation language. A basic rewrite rule that transforms a term pattern **term1** to a term pattern **term2** has the following form: |  |

rule-name: term1 -> term2

Term patterns have the same form as terms: any term is a legal term pattern. In addition to the basic constructors, string literals, integer literals, and so on, they also support variables (e.g., **v** or **name**) and wildcards (indicated by **\_**). As an example, the following rewrite rule rewrites an **Entity** to the list of properties contained in that entity:

|  |
| --- |
| get-properties:  Entity(name, properties) -> properties |
| So, for an **Entity("User", [Property("name", String)])**, it binds **"User"** to the variable **name**, and **[Property("name", "String")]** to the variable **properties**. It then returns the collection **properties**. While rewrite rules can be viewed as functions, they have one important difference: they can be de- | | |  |
| fined multiple times for different patterns15. In the case of | | |  |
| **get-properties**, we could add another definition that works | | | overloading. |

for property access expressions:

|  |
| --- |
| get-properties:  PropAccess(expr, property) -> property |

Rules can have complex patterns. For example, it is possible to write a rule that succeeds only for entities with *only* a **name** property[[2]](#footnote-2):

|  |
| --- |
| is-name-only-entity:  Entity(\_, [Property("name", "String")]) -> True() |

Rewrite rules can be invoked using the syntax **<rule-name> term**17. The angle brackets make it easy to distinguish rule in-

vocations from terms, and makes it possible to use invocations in term expressions.

Stratego provides a **with** clause that can be used for additional code that should be considered for rewrite rules. The **with** clause is most commonly used for assignments and calls to other rules. As an example, we can write the rule above using a **with**. This rule assigns the value of **get-properties** to a variable **result** and returns that as the result value of the rule:

|  |
| --- |
| invoke-get-properties:  Entity(name, properties) -> result **with** result := <get-properties> Entity(name, properties) |
| Rules can also have conditions. These can be specified using | | |  |
| **where**18. These clauses typically use the operators listed in the | | |  |
| following table: | | |  |

|  |  |
| --- | --- |
| **Expression** | **Description** |
| **<e> t** | Applies **e** to **t**, or fails if **e** is unsuccessful. |
| **v := t** | Assign a term expression **t** to a variable **v**. |
| **!t => p** | Match a term **t** against a pattern **p**, or fail. |
| **not(e)** | Succeeds if **e** does not succeed. |
| **e1; e2** | Sequence: apply **e1**. If it succeeds, apply **e2**. |
| **e1 <+ e2** | Choice: apply **e1**, if it fails apply e2 instead. |

An example of a rule with a **where** clause is the following:

|  |
| --- |
| has-properties:  Entity(name, properties) -> True() **with** properties := <get-properties> Entity(name, properties);  **where** not(!properties => []) |
| This rule only succeeds for entities where the where condition | |  |
| **not(!properties => [])** holds19. That is, it succeeds as long | |  |
| as an entity does not have an empty list (indicated by **[]**) of properties. Rewrite rules can have any number of **where** and **with** clauses, and they are evaluated in the order they appear. | |  |
| Like functions or methods in other languages, rewrite rules can have parameters. Stratego distinguishes between parameters that pass other rules and parameters that pass terms, using a vertical bar to separate the two separate lists20. The Stratego | |  |
| standard library provides a number of higher-order rules, i.e. | |  |

rules that take other rules as their argument. These rules are used for common operations on abstract syntax trees: for example, **map(r)** applies a rule **r** to all elements of a list:

|  |
| --- |
| get-property-types:  Entity(\_, properties) -> types **with** types := <map(get-property-type)> properties  get-property-type:  Property(\_, type) -> type |

Rules like **map** specify a *traversal* on a certain term structure: they specify how a particular rule should be applied to a term and its subterms. Rules that specify traversals are also called *strategies*21. In Spoofax, strategies are used to control traversals

|  |  |
| --- | --- |
| *9.3.3 Index-Based Constraint Rules*  Some constraint rules interact with the Spoofax index23. One |  |
| way to do this is to use URI annotations on the abstract syntax. |  |

in constraints, transformation, and code generation.

#### 9.3.2 Basic Constraint Rules

Spoofax uses rules with the name **constraint-error** to indicate constraints that trigger errors, **constraint-warning** for warnings, and **constraint-note** for notes. To report an error, warning or information note, these rules have to be overwritten for the relevant term patterns. The following example is created by default by the Spoofax project wizard. It simply reports a note for any module named **example**:

|  |
| --- |
| constraint-note:  Module(name, \_) -> (name, "This is just an example program.") **where**  !name => "example" |
| The condition checks if the module **name** matches the string **"example"**. The rule returns (via its right-hand side) a tuple with the tree node where the marker should appear and a string message that should be shown. All constraint rules have this form.  Most constraint rules use string interpolation for error messages. Interpolated strings have the form **$[...]** where variables can be escaped using **[...]**. The following example uses | | |  |
| string interpolation to report a warning22. | | |  |

|  |
| --- |
| constraint-warning:  Entity(theName, \_) -> (theName,  $[Entity [theName] does not have a capitalized name]) **where** not(<string-starts-**with**-capital> theName) |

These are placed on each reference and definition. For example, a reference to a Mobl variable **v** is represented as **Var("v")**. With an annotation, it reads as follows:

Var("v"{[Var(),"v","function","module"]})

The annotation is added directly to the name, surrounded with curly braces24. Unresolved references are represented by terms

|  |  |
| --- | --- |
| such as the following (notice the **Unresolved** term, surrounding the namespace): |  |

order. Var("u"{[Unresolved(Var()),"u","function","module"]})

|  |
| --- |
| constraint-error:  Entity(name, \_) -> (name, $[Duplicate definition]) **where** defs := <index-lookup-all> name;  <gt> (<length> defs, 1) |

In most statically typed languages, references that cannot be statically resolved indicate an error. The following constraint rule reports an error for these cases:

|  |  |
| --- | --- |
| constraint-error:  x -> (x, $[Unable to resolve reference.]) **where**  !x => \_{[Unresolved(t) | \_]} | |
| This rule matches any term **x** in the abstract syntax, and reports | | | |  | |
| an error if it has an **Unresolved** annotation25. Note how the | | | |  | |
| pattern **\_{[Unresolved(t) | \_]}** matches any term (indicated by the wildcard **\_**) that has a list annotation where the head of the list is **Unresolved(t)** and the tail matches **\_**.  In addition to annotations, the Spoofax index provides an API for inspecting naming relations in programs. The following table shows some of the key rules the index provides. | | | |  | |
| **index-uri** | | Gets the URI of a term. | | |
| **index-namespace** | | Gets the namespace of a term. | | |
| **index-lookup** | | Returns the first definition of a reference. | | |
| **index-lookup-all** | | Returns all definitions of a reference. | | |
| **index-get-files-of** | | Gets all files a definition occurred in. | | |
| **index-get-all-in-file** | | Gets all definitions for a given file path. | | |
| **index-get-current-file** | | Gets the path of the current file. | | |

We can use the index API to detect duplicate definitions. In most languages, duplicate definitions are always disallowed. In the case of Mobl, duplicate definitions are not allowed for functions or entities, but they are allowed for variables, just as in JavaScript. The following constraint rules checks for duplicate entity declarations:

This rule matches any entity declaration. Then, it fires a helper rule **is-duplicates-allowed**. Next, the constraint rule determines all definition sites of the entity name. If the list has more than one element, the rule reports an error. This is checked by comparing the length of the list with **1** by calling the **gt** ("greater than") rule. More sophisticated constraints and error messages can be specified using a type system, as we show in the next chapter.

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)