Das *autOlat*-Protokoll
Version 0.1.0

Bertram Felgenhauer

3. August 2010

Inhaltsverzeichnis

1	Begriffe	3
2	Datentypen	3
2.1	Basistypen	3
2.2	Aliase	4
2.3	Documented α	4
2.4	Signed α	4
2.5	Instance	4
2.6	ServerInfo	5
2.7	TaskTree	5
2.8	Version	5
2.9	TaskDescription	6
2.10	ScoringOrder	6
2.11	Either $\alpha \beta$	6
3	RPC-Aufrufe	6
3.1	get_server_info	7
3.2	get_task_types	7
3.3	get_task_description	7
3.4	verify_task_config	8
3.5	get_task_instance	8
3.6	grade_task_solution	9
4	Protokoll	10
4.1	XML-RPC	10
4.1.1	Basistypen	10
4.1.2	Strukturtypen	10
4.2	Abbildung der Datentypen in XML-RPC	10
4.2.1	Basistypen	10
4.2.2	komplexe Datentypen	10
4.2.3	Records	11
4.2.4	Tupel	11
4.2.5	Either	11

1 Begriffe

Server Programm, das als Anbieter von Aufgabensemantik für verschiedene Aufgabentypen fungiert. Es nimmt Anfragen vom *Client* entgegen, und beantwortet diese.

Client Nutzer des *Servers*. Dies kann z.B. eine Lernplattform sein, über die Lernenden Zugang zu den vom Server implementierten Aufgaben bekommt.

Aufgabentyp Ein *Server* kann Semantik für verschiedene Aufgabentypen anbieten. Beispiele wären das Lösen linearer Gleichungssysteme oder das Faktorisieren einer Zahl. Im Allgemeinen umfasst jeder Aufgabentyp viele konkrete Aufgabenstellungen.

Aufgabenkonfiguration Zusammen mit dem *Aufgabentyp* definiert die Aufgabenkonfiguration die Rahmenparameter für die zugehörigen *Aufgabeninstanzen*. Dies könnte z.B. die Anzahl der Gleichungen in einem linearen Gleichungssystem sein.

Aufgabeninstanz Die Aufgabeninstanz ist eine konkrete Aufgabenstellung, also beispielsweise ein lineares Gleichungssystem. Der *Server* kann diese Anhand einer *Aufgabenkonfiguration* und zugehörigem *Aufgabentyp* erzeugen. Die Aufgabeninstanz kann dem Endnutzer präsentiert und von diesem gelöst werden.

Aufgabenlösung Eine Aufgabenlösung stellt einen Lösungsversuch für eine *Aufgabeninstanz* dar. Diese kann vom *Server* auf Korrektheit geprüft werden.

Bewertung Der *Server* stellt fest, ob die Lösung korrekt war oder nicht.

Größe Neben der eigentlichen *Bewertung* gibt es die Möglichkeit, Aufgabenlösungen eine zusätzliche Zahl, die *Größe* zuzuordnen, beispielsweise die Anzahl der Lösungsschritte einer Lösung. Diese ist von sekundärer Bedeutung - eine korrekte Lösung mit 100 Schritten ist genauso richtig wie eine mit 20 Schritten. Man kann die Zahl aber für Bestenlisten für die jeweilige Aufgabe verwenden.

2 Datentypen

2.1 Basistypen

Typ	Bedeutung
String	UTF-8-kodierte Zeichenkette
Int	vorzeichenbehafteter 32-bit Ganzzahlwert
Double	64-bit Gleitkommazahl
$[\alpha]$	Liste von Elementen vom Typ α
(α, β)	Paar von Werten vom Typ α und β
(α, β, γ)	Tripel

2.2 Aliase

Alias	Typ	Bedeutung
Name	String	uninterpretierter Name
Task	String	Name eines Aufgabentyps
Seed	String	vom Nutzer abhängiger String zur Generierung von Aufgaben
Signature	String	vom Server erzeugte Signatur
Config	String	Aufgabenkonfiguration
Description	String	XML-Dokument mit textueller Beschreibung
Solution	String	Aufgabenlösung

2.3 Documented α

```
data Documented  $\alpha$  = Documented {  
    contents ::  $\alpha$ ,  
    documentation :: Description  
}
```

Beschreibung

Dieser Typ dient der Annotation eines Wertes mit einer textuellen Beschreibung für den Nutzer.

2.4 Signed α

```
data Signed  $\alpha$  = Signed {  
    contents ::  $\alpha$ ,  
    signature :: Signature  
}
```

Beschreibung

Dieser Typ dient der Annotation eines Wertes mit einer Signatur. Die Signatur wird vom Server geprüft, wenn ein vom Server gelieferter Wert vom Client an den Server zurückgegeben wird. Für den Client ist die Signatur ohne Bedeutung; er muss sie allerdings aufbewahren.

2.5 Instance

```
data Instance = Instance {  
    tag :: String,  
    contents :: String  
}
```

Beschreibung

Dieser Typ beschreibt zusammen mit einem Aufgabentyp eine Aufgabeninstanz. Wie die Datenfelder genau verwendet werden, obliegt dem Server.

2.6 ServerInfo

```
data ServerInfo = ServerInfo {
    protocol_version :: Version,
    server_name     :: Name,
    server_version  :: Version
}
```

Beschreibung

Werte dieses Typs enthalten den Servernamen und dessen Version, sowie die Protokollversion, die der Server unterstützt. Siehe [Version \(2.8\)](#).

2.7 TaskTree

```
data TaskTree = Task {
    task_name :: Task
}
| Category {
    category_name :: Name,
    sub_trees    :: [TaskTree]
}
```

Beschreibung

Ein Wert dieses Typs enthält entweder den Namen eines Aufgabentyps, oder beschreibt eine ganze Unterkategorie von Aufgaben. In diesem Fall enthält er den Namen der Kategorie sowie eine Liste von Teilaufgaben und weiteren Unterkategorien.

2.8 Version

```
data Version = Version {
    major :: Int,
    minor :: Int,
    micro :: Int
}
```

Beschreibung

Dieser Typ beschreibt eine Versionsnummer. Versionsnummern bestehen aus drei Komponenten. Dabei gilt folgende Konvention:

- Bei inkompatiblen Änderungen soll die major-Version erhöht werden.
- Bei abwärtskompatiblen Änderungen soll die minor-Version erhöht werden.
- Bei kompatiblen Änderungen soll die micro-Version erhöht werden.

2.9 TaskDescription

```
data TaskDescription = TaskDescription
    task_sample_config :: Documented Config,
    task_scoring_order :: ScoringOrder
```

Beschreibung

Werte dieses Typs beschreiben einen Aufgabentyp näher. Dabei wird eine Beispielkonfiguration zurückgeliefert, die mindestens syntaktisch korrekt sein soll, dabei nach Möglichkeit auch eine sinnvolle Aufgabenstellung ergibt.

Daneben wird angegeben, in welcher Reihenfolge Punkte für Toplisten ausgewertet werden sollen. Siehe `ScoringOrder` (2.10).

2.10 ScoringOrder

```
data ScoringOrder = Increasing
                  | None
                  | Decreasing
```

Beschreibung

Sortierreihenfolge für Bestenlisten.

- `Decreasing` – absteigende Sortierung. Das heißt, eine größere Größe ist besser als eine kleinere.
- `Increasing` – aufsteigende Sortierung. Kleinere Größen sind besser.
- `None` – keine Angabe, z.B. weil die Sortierreihenfolge von der Aufgabenstellung abhängt, oder weil die berechnete Größe nicht variiert.

2.11 Either α β

```
data Either  $\alpha$   $\beta$  = Left  $\alpha$ 
                  | Right  $\beta$ 
```

Beschreibung

`Either  $\alpha$   $\beta$` stellt eine Alternative zwischen einem Wert vom Typ α oder einem Wert vom Typ β dar.

3 RPC-Aufrufe

Die Aufrufe sind an Haskell angelehnt in der Form

```
function_name :: ParameterTyp1 -> ParameterTyp2 -> ResultatTyp
dargestellt.
```

3.1 `get_server_info`

`get_server_info` :: `ServerInfo`

Zweck

Abfrage von grundlegenden Serverinformationen. Dazu gehört insbesondere auch die unterstützte Protokollversion, so dass Inkompatibilitäten erkannt werden können.

Parameter

keine

Rückgabewert

Der Aufruf liefert Name und Version des Servers sowie die implementierte Version des Protokolls zurück. Siehe `ServerInfo` (2.6).

3.2 `get_task_types`

`get_task_types` :: `[TaskTree]`

Zweck

Dieser Aufruf liefert die vom Server unterstützten Aufgabentypen. Diese sind thematisch in einem Baum strukturiert, um die Auswahl auf Seite des Anwenders zu unterstützen.

Parameter

keine

Rückgabewert

Ein Baum (bzw. Wald) von kategorisierten Aufgabentypen. Siehe `TaskTree` (2.7)

3.3 `get_task_description`

`get_task_description` :: `Task -> TaskDescription`

Zweck

Mit diesem Aufruf können nähere Informationen zu einem Aufgabentyp erfragt werden. Dazu gehört unter anderem eine Beispielkonfiguration für die Aufgabe.

Parameter

- Der Aufgabentyp, dessen Information abgefragt werden soll.

Rückgabewert

Die Aufgabenbeschreibung. Siehe `TaskDescription` (2.9).

3.4 `verify_task_config`

```
verify_task_config :: Task -> Config
  -> Either Description (Signed (Task, Config))
```

Zweck

Der Aufruf gibt dem Server die Möglichkeit, eine Aufgabenkonfiguration zu einem Aufgabentyp auf Korrektheit bzw. Plausibilität zu prüfen. Wenn Fehler gefunden werden, werden diese in Textform zurückgeliefert. Anderenfalls wird die Aufgabenkonfiguration signiert und zur weiteren Verwendung in `get_task_instance` (3.5) zurückgegeben.

Parameter

- der Aufgabentyp
- die zu prüfende Aufgabenkonfiguration

Rückgabewert

Je nach Resultat der Prüfung wird zurückgegeben:

- `Left error`, wobei `error` eine Beschreibung des Fehlers ist.
- `Right signed`, wobei `signed` die signierte Aufgabenkonfiguration inklusive Aufgabentyp ist.

3.5 `get_task_instance`

```
get_task_instance :: Signed (Task, Config) -> Seed
  -> (Signed (Task, Instance), Description, Documented Solution)
```

Zweck

Mit diesem Aufruf wird zu einer gegebenen geprüften Aufgabenkonfiguration (die Signatur dient als Zertifikat für die Prüfung) eine Aufgabeninstanz erzeugt.

Parameter

- Eine signierte Aufgabenkonfiguration. Siehe `verify_task_config` (3.4)
- Eine nutzerabhängige Zeichenkette zur Initialisierung eines Zufallszahlengenerators. Damit ist es möglich, individuelle Aufgaben zu erzeugen.

Rückgabewert

Es wird ein Tripel zurückgegeben. Darin sind folgende Werte enthalten:

- Eine signierte Aufgabeninstanz. Die darin enthaltenen Daten sollen dem Anwender nicht angezeigt werden.
- Eine textuelle Aufgabenstellung für den Nutzer.
- Eine Lösungsvorlage mit Dokumentation.

3.6 `grade_task_solution`

```
grade_task_solution :: Signed (Task, Instance) -> Solution  
-> Either Description (Documented Double)
```

Zweck

Hiermit können Aufgabenlösungen überprüft werden. Ähnlich wie bei `verify_task_config` (3.4) wird entweder ein Fehler mit Erläuterung zurückgegeben, oder die Größe der Lösung zusammen mit einem Kommentar.

Parameter

- Eine signierte Aufgabeninstanz, wie von `get_task_instance` (3.5) zurückgegeben.
- Eine Aufgabenlösung.

Rückgabewert

Je nach Resultat der Bewertung wird zurückgegeben:

- `Left hint` bei erfolgloser Lösung, wobei `hint` eine Begründung dafür angibt, warum die Lösung falsch war.
- `Right size` bei erfolgreicher Lösung, wobei `size` die Größe der Lösung mit einem Kommentar enthält.

4 Protokoll

4.1 XML-RPC

Die RPC-Aufrufe werden per XML-RPC-Protokoll übertragen. XML-RPC basiert auf HTTP. Ein RPC-Aufruf ist ein POST-Request an die URL des *autO-lat*-Service, bei dem die Parameter und Rückgabewerte XML-kodiert übertragen werden. Details zur Kodierung gibt es unter <http://www.xmlrpc.com/>.

4.1.1 Basistypen

Die folgenden XML-RPC-Basistypen werden verwendet:

- `string` – Zeichenketten – für `String` und `Int`
- `double` – Gleitkommazahlen – für `Double`
- `boolean` – Wahrheitswerte – für `Bool`

4.1.2 Strukturtypen

- `array` – Felder – für Listen (`[ $\alpha$ ]`) und komplexe Datentypen.
Verkürzte Notation: `[ a, b, ... ]`
- `struct` – records – für komplexe Datentypen
Verkürzte Notation: `{ k1: v1, k2: v2, ... }`

4.2 Abbildung der Datentypen in XML-RPC

4.2.1 Basistypen

Typ	Beispiel	XML-RPC-Typ	Beispiel
String	"abc"	string	"abc"
Int	123	string	"123"
Double	123.0	double	123.0
Bool	True	boolean	true
<code>[α]</code>	<code>["a", 42]</code>	array	<code>["a", 42]</code>

Werte von Typaliasen werden wie die entsprechenden Basistypen kodiert.

4.2.2 komplexe Datentypen

Komplexe (algebraische) Datentypen haben einen Namen, Typparameter, falls sie generisch sind, und eine oder mehrere Ausprägungen. Jede Ausprägung ist ein Record oder ein Tupel. Die jeweils verwendete Ausprägung bestimmt die Kodierung der Werte.

```
data Name  $\alpha$  = Alternative1
  | Alternative2
  ...
```

4.2.3 Records

Records haben einen Namen und benannte Datenfelder. Beispiel:

```
data ... = Record {  
  field1 ::  $\alpha$ ,  
  field2 ::  $\beta$   
}
```

Sie werden als `struct` kodiert, die eine weitere `struct` enthält:

```
{  
  "Record": {  
    "field1": ...,  
    "field2": ...  
  }  
}
```

4.2.4 Tupel

Tupel haben einen Namen und *unbenannte* Datenfelder. Beispiel:

```
data ... = Record  $\alpha \beta$ 
```

Sie werden als `struct` kodiert, die ein `array` enthält:

```
{  
  "Record": [  
    ...,  
    ...  
  ]  
}
```

4.2.5 Either

```
data Either  $\alpha \beta$  = Left  $\alpha$  | Right  $\beta$ 
```

Abweichend von der normalen Kodierung für Tupel werden die Werte von `Either` nicht extra in array verpackt. So wird `Left ...` als

```
{  
  "Left": ...  
}
```

kodiert. Die Kodierung von `Right ...` ist analog.