1 CASS: A Generic Curry Analysis Server System

CASS (Curry Analysis Server System) is a tool for the analysis of Curry programs. CASS is generic so that various kinds of analyses (e.g., groundness, non-determinism, demanded arguments) can be easily integrated into CASS. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. Moreover, it can be used by different programming tools, like documentation generators, analysis environments, program optimizers, as well as Eclipse-based development environments. For this purpose, CASS can also be invoked as a server system to get a language-independent access to its functionality. CASS is completely implemented Curry as a master/worker architecture to exploit parallel or distributed execution environments. The general design and architecture of CASS is described in [1]. In the following, CASS is presented from a perspective of a programmer who is interested to analyze Curry programs.

1.1 Installation

The current implementation of CASS is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of CASS, use the following commands:

```
> cypm update
> cypm install cass
```

This downloads the newest package, compiles it, and places the executable `cass` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CASS as described below.

1.2 Using CASS to Analyze Programs

CASS is intended to analyze various operational properties of Curry programs. Currently, it contains more than a dozen program analyses for various properties. Since most of these analyses are based on abstract interpretations, they usually approximate program properties. To see the list of all available analyses, use the help option of CASS:

```
> cass -h
Usage: ...
⋮
Registered analyses names:
...
Demand          : Demanded arguments
Deterministic   : Deterministic operations
⋮
```

More information about the meaning of the various analyses can be obtained by adding the short name of the analysis:

```
> cass -h Deterministic
...
```

For instance, consider the following Curry module `Rev.curry`:

```
append :: [a] → [a] → [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys

rev :: [a] → [a]
rev []     = []
rev (x:xs) = append (rev xs) [x]

nth :: [a] → Int → a
nth (x:xs) n | n == 0 = x
             | n > 0  = nth xs (n - 1)
```

CASS supports three different usage modes to analyze this program.

1.2.1 Batch Mode

In the batch mode, CASS is started as a separate application via the shell command `cass`, where the analysis name and the name of the module to be analyzed must be provided:[1]

```
> cass Demand Rev
append : demanded arguments: 1
nth : demanded arguments: 1,2
rev : demanded arguments: 1
```

The `Demand` analysis shows the list of argument positions (e.g., 1 for the first argument) which are demanded in order to reduce an application of the operation to some constructor-rooted value. Here we can see that both arguments of `nth` are demanded whereas only the first argument of `append` is demanded. This information could be used in a Curry compiler to produce more efficient target code.

The batch mode is useful to test a new analysis and get the information in human-readable form so that one can experiment with different abstractions or analysis methods.

1.2.2 API Mode

The API mode is intended to use analysis information in some application implemented in Curry. Since CASS is implemented in Curry, one can import the modules of the CASS implementation and use the CASS interface operations to start an analysis and use the computed results. For instance, CASS provides an operation (defined in module `CASS.Server`)

```
analyzeGeneric :: Analysis a → String → IO (Either (ProgInfo a) String)
```

to apply an analysis (first argument) to some module (whose name is given in the second argument). The result is either the analysis information computed for this module or an error message in case of some execution error.

In order to use CASS via the API mode in a Curry program, one has to use the package `cass` by the Curry package manager CPM (the subsequent explanation assumes familiarity with the basic features of CPM):

[1] More output is generated when the property `debugLevel` is changed in the configuration file `.curryanalysisrc` which is installed in the user's home directory when CASS is started for the first time.

1. Add the dependency on package `cass` and also on package `cass-analysis`, which contains some base definitions, in the package specification file `package.json`.

2. Install these dependencies by "`cypm install`".

Then you can import in your application the modules provided by CASS.

The module `Analysis.ProgInfo` (from package `cass-analysis`) contains operations to access the analysis information computed by CASS. For instance, the operation

```
lookupProgInfo:: QName → ProgInfo a → Maybe a
```

returns the information about a given qualified name in the analysis information, if it exists. As a simple example, consider the demand analysis which is implemented in the module `Analysis.Demandedness` by the following operation:

```
demandAnalysis :: Analysis DemandedArgs
```

`DemendedArgs` is just a type synonym for `[Int]`. We can use this analysis in the following simple program:

```
import CASS.Server           ( analyzeGeneric )
import Analysis.ProgInfo     ( lookupProgInfo )
import Analysis.Demandedness ( demandAnalysis )

demandedArgumentsOf :: String → String → IO [Int]
demandedArgumentsOf modname fname = do
  deminfo <- analyzeGeneric demandAnalysis modname >>= return . either id error
  return $ maybe [] id (lookupProgInfo (modname,fname) deminfo)
```

Of course, in a realistic program, the program analysis is performed only once and the computed information `deminfo` is passed around to access it several times. Nevertheless, we can use this simple program to compute the demanded arguments of `Rev.nth`:

```
...> demandedArgumentsOf "Rev" "nth"
[1,2]
```

1.2.3 Server Mode

The server mode of CASS can be used in an application implemented in some language that does not have a direct interface to Curry. In this case, one can connect to CASS via some socket using a simple communication protocol that is specified in the file `Protocol.txt` (in package `cass`) and sketched below.

To start CASS in the server mode, one has to execute the command

```
> cass --server [ -p <port> ]
```

where an optional port number for the communication can be provided. Otherwise, a free port number is chosen and shown. In the server mode, CASS understands the following commands:

```
GetAnalysis
SetCurryPath <dir1>:<dir2>:...
AnalyzeModule          <analysis name> <output type> <module name>
```

```
AnalyzeInterface       <analysis name> <output type> <module name>
AnalyzeFunction        <analysis name> <output type> <module name> <function name>
AnalyzeDataConstructor <analysis name> <output type> <module name> <constructor name>
AnalyzeTypeConstructor <analysis name> <output type> <module name> <type name>
StopServer
```

The output type can be `Text`, `CurryTerm`, or `XML`. The answer to each request can have two formats:

```
error <error message>
```

if an execution error occured, or

```
ok <n>
<result text>
```

where `<n>` is the number of lines of the result text. For instance, the answer to the command `GetAnalysis` is a list of all available analyses. The list has the form

```
<analysis name> <output type>
```

For instance, a communication could be:

```
> GetAnalysis
< ok 5
< Deterministic curryterm
< Deterministic text
< Deterministic json
< HigherOrder   curryterm
< DependsOn     curryterm
```

The command `SetCurryPath` instructs CASS to use the given directories to search for modules to be analyzed. This is necessary since the CASS server might be started in a different location than its client.

Complete modules are analyzed by `AnalyzeModule`, whereas `AnalyzeInterface` returns only the analysis information of exported entities. Furthermore, the analysis results of individual functions, data or type constructors are returned with the remaining analysis commands. Finally, `StopServer` terminates the CASS server.

For instance, if we start CASS by

```
> cass --server -p 12345
```

we can communicate with CASS as follows (user inputs are prefixed by "`>`");

```
> telnet localhost 12345
Connected to localhost.
> GetAnalysis
ok 198
Functional text
Functional short
Functional curryterm
Functional json
Functional jsonterm
Functional xml
```

```
Overlapping text
...
> AnalyzeModule Demand text Rev
ok 3
append : demanded arguments: 1
nth : demanded arguments: 1,2
rev : demanded arguments: 1
> AnalyzeModule Demand curryterm Rev
ok 1
[(("Rev","append"),"[1]"),(("Rev","nth"),"[1,2]"),(("Rev","rev"),"[1]")]
> AnalyzeModule Demand json Rev
ok 15
[ {
  "module": "Rev",
  "name": "append",
  "result": "demanded arguments: 1"
}
, {
  "module": "Rev",
  "name": "nth",
  "result": "demanded arguments: 1,2"
}
, {
  "module": "Rev",
  "name": "rev",
  "result": "demanded arguments: 1"
} ]
> AnalyzeModule Demand xml Rev
ok 19
<?xml version="1.0" standalone="yes"?>

<results>
  <operation>
    <module>Rev</module>
    <name>append</name>
    <result>demanded arguments: 1</result>
  </operation>
  <operation>
    <module>Rev</module>
    <name>nth</name>
    <result>demanded arguments: 1,2</result>
  </operation>
  <operation>
    <module>Rev</module>
    <name>rev</name>
    <result>demanded arguments: 1</result>
  </operation>
</results>
> StopServer
```

```
ok 0
Connection closed by foreign host.
```

1.3 Implementing Program Analyses

This section explains the implementation of program analyses available in CASS. Since CASS is implemented in Curry, a program analysis must also be implemented in Curry and added to the source code of CASS. Therefore, one has to download the source code which is easily done by the command

```
> cypm checkout cass
```

This downloads the most recent version of CASS as a Curry package into the directory `cass`.

Each program analysis accessible by CASS must be registered in the CASS module `CASS.Registry`. Such an analysis must contain an operation of type

```
Analysis a
```

where "`a`" denotes the type of analysis results. Furthermore, the analysis must also contain a "show" operation of type

```
AOutFormat → a → String
```

intended to show the analysis results in various formats. The type `AOutFormat` is defined in module `Analysis.Types` of package `cass-analysis` as

```
data AOutFormat = AText | ANote
```

It is intended to specify the desired kind of output, e.g., `AText` for a longer standard textual representation or `ANote` for a short note (e.g., in the Curry Browser).

Thus, in order to add a new analysis to CASS, one has to do the following steps:

1. Implement a corresponding analysis operation and show operation.
2. Registering it in the module `CASS.Registry` (in the constant `registeredAnalysis`).
3. Compile/install the modified CASS implementation.

In the following, we explain these steps by some examples. For instance, the `Overlapping` analysis should indicate whether a Curry operation is defined by overlapping rules. This analysis can be implemented as a function

```
overlapAnalysis :: Analysis Bool
```

so that the analysis result is `False` if the analyzed operation is not defined by overlapping rules.

In general, an analysis is implemented as a mapping from Curry operations, represented in FlatCurry, into the analysis result. Hence, to implement the `Overlapping` analysis, we define the following operation on function declarations in FlatCurry format:

```
import FlatCurry.Types
...
isOverlappingFunction :: FuncDecl → Bool
isOverlappingFunction (Func _ _ _ _ (Rule _ e))   = orInExpr e
```

```
isOverlappingFunction (Func f _ _ _ (External _)) = f == ("Prelude","?")

-- Check an expression for occurrences of Or:
orInExpr :: Expr  → Bool
orInExpr (Var _)       = False
orInExpr (Lit _)       = False
orInExpr (Comb _ f es) = f == ("Prelude","?") || any orInExpr es
orInExpr (Free _ e)    = orInExpr e
orInExpr (Let bs e)    = any orInExpr (map snd bs) || orInExpr e
orInExpr (Or _ _)      = True
orInExpr (Case _ e bs) = orInExpr e || any orInBranch bs
 where orInBranch (Branch _ be) = orInExpr be
orInExpr (Typed e _)   = orInExpr e
```

In order to support the inclusion of different kinds of analyses in CASS, CASS offers several constructor operations for the abstract type "`Analysis a`" (which is defined in module `Analysis.Types`). Each analysis has a name provided as a first argument to these constructors. The name is used to store the analysis information persistently and to pass specific analysis tasks to analysis workers. For instance, a simple function analysis which depends only on a given function definition can be defined by the analysis constructor

```
simpleFuncAnalysis :: String  → (FuncDecl  → a)  → Analysis a
```

The arguments are the analysis name and the actual analysis function. Hence, the "overlapping rules" analysis can be specified as

```
import Analysis.Types
...
overlapAnalysis :: Analysis Bool
overlapAnalysis = simpleFuncAnalysis "Overlapping" isOverlappingFunction
```

In order to integrate this analysis into CASS, we also have to define an operation to show the analysis results in a human-readable form:

```
showOverlap :: AOutFormat  → Bool  → String
showOverlap _     True  = "overlapping"
showOverlap AText False = "non-overlapping"
showOverlap ANote False = ""
```

Here, the typical case of non-overlapping rules is not printed in case of short notes.

Now we have all elements available in order to add this analysis to CASS. To support this easily, there is an operation

```
cassAnalysis :: (Read a, Show a, Eq a)
             => String  → Analysis a  → (AOutFormat  → a  → String)
              → RegisteredAnalysis
```

to transform an analysis with some title, an analysis operation, and a "show" operation into an analysis ready to be registered in CASS. The actually registered analyses are specified by the constant

```
registeredAnalysis :: [RegisteredAnalysis]
```

defined in module `CASS.Registry`. Hence, the `Overlapping` can be integrated into CASS by adding it to the definition of `registeredAnalysis`, e.g.,

```
registeredAnalysis :: [RegisteredAnalysis]
registeredAnalysis =
  [
  ⋮
  cassAnalysis "Overlapping rules" overlapAnalysis showOverlap
  ⋮
  ]
```

As a final step, we have to compile and install this extended version of CASS by executing

```
> cypm install
```

in the downloaded package. After this step, one can executed

```
> cass --help
```

to check whether the `Overlapping` analysis occurs in the list of registered analyses names.

To show an example of a more complex kind of analysis, we consider a determinism analysis. Such an analysis could be based on an abstract domain described by the data type

```
data Deterministic = NDet | Det
  deriving (Eq, Read, Show)
```

Here, `Det` is interpreted as "the operation always evaluates in a deterministic manner on ground constructor terms." However, `NDet` is interpreted as "the operation *might* evaluate in different ways for given ground constructor terms." The apparent imprecision is due to the approximation of the analysis. For instance, if the function `f` is defined by overlapping rules and the function `g` *might* call `f`, then `g` is judged as non-deterministic (since it is generally undecidable whether `f` is actually called by `g` in some run of the program).

The determinism analysis requires to examine the current function as well as all directly or indirectly called functions for overlapping rules. Due to recursive function definitions, this analysis cannot be done in one shot for a given function—it requires a fixpoint computation. CASS provides such fixpoint computations and simplifies its implementation by requiring only the implementation of an operation of type

```
FuncDecl → [(QName,a)] → a
```

where "`a`" denotes the type of abstract values. The second argument of type `[(QName,a)]` represents the currently known analysis values for the functions *directly* used in this function declaration. Hence, in the implementation one can assume that the analysis results of all functions occurring in the definition of the function to be analyzed are already known, although they will be approximated by a fixpoint computation performed by CASS. Technically, the abstract values must be a domain with some bottom element and the analysis operation must be monotone. Since this is not checked by CASS, we omit these details.

In our example, the determinism analysis can be implemented by the following operation:

```
detFunc :: FuncDecl → [(QName,Deterministic)] → Deterministic
```

```
detFunc (Func f _ _ _ (External _)) _           = f == ("Prelude","?")
detFunc (Func f _ _ _ (Rule _ e))   calledFuncs =
  if orInExpr e || freeVarInExpr e || any (==NDet) (map snd calledFuncs)
    then NDet
    else Det
```

Thus, it computes the abstract value `NDet` if the function itself is defined by overlapping rules or contains free variables that might cause non-deterministic guessing (we omit the definition of `freeVarInExpr` since it is quite similar to `orInExpr`), or if it depends on some non-deterministic function.

To support the integration of such fixpoint analyses in CASS, there exists the following analysis constructor:

```
dependencyFuncAnalysis :: String → a → (FuncDecl → [(QName,a)] → a)
                       → Analysis a
```

Here, the second argument specifies the start value of the fixpoint analysis, i.e., the bottom element of the abstract domain. Hence, the complete determinism analysis can be specified as

```
detAnalysis :: Analysis Deterministic
detAnalysis = dependencyFuncAnalysis "Deterministic" Det detFunc
```

In order to register this analysis, we define a show function

```
showDet :: AOutFormat → Deterministic → String
showDet _     NDet = "non-deterministic"
showDet AText Det  = "deterministic"
showDet ANote Det  = ""
```

extend the definiton of `registeredAnalysis` by the line

```
cassAnalysis "Deterministic operations" detAnalysis showDet
```

and compile and install the package.

This simple definition is sufficient to execute this analysis with CASS, since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, caching analysis results, etc. The actual analysis time depends on the size of modules and their imports, the size of the dependencies, and the number of fixpoint iterations (which depends also on the depth of the abstract domain).[2] Beyond the analysis time, it is also important that the analysis terminates, which is not ensured in general fixpoint computations. Termination can be achieved by using an abstract domain with finitely many values and defining the analysis function so that it is monotone w.r.t. some ordering on the abstract values.

Required class instances. Note that the type of an abstract domain are required to have instances of the type classes `Eq`, `Read`, `Show`, and `ReadWrite`, since abstract values need to be compared (e.g., to check whether a fixpoint has been reached) and persistently stored (to support an incremental modular analysis). Whereas instances of `Eq`, `Read`, and `Show` can be automatically derived

[2]CASS supports different methods to compute fixpoints, see the property `fixpoint` in the configuration file `.curryanalysisrc` which is installed in the user's home directory when CASS is started for the first time. This property can also be set in the command to invoke CASS.

(via a `deriving` annotation as shown above), instances of `ReadWrite` (which support a compact data representation) can be generated by the tool `curry-rw-data`. This tool is available as a Curry package and can be installed by

```
> cypm install rw-data-generator
```

Then, `ReadWrite` instances of all data types defined in a module `AnaMod` can be generated by the command

```
> curry-rw-data AnaMod
```

This command generates a new module `AnaModRW` containing the instance definitions which might be inserted into the analysis implementation module `AnaMod`.

References

[1] M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.