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]

main :: Int → Int → [Int]
main x y = rev [x .. y]
```

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
main : 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 `main` 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 the module `AnalysisServer`)

```
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.

The modules of the CASS implementation are stored in the directory *curryhome*`/currytools/CASS` and the modules implementing the various program analyses are stored in *curryhome*`/currytools/analysis`. Hence, one should add these directories to the Curry load path when using CASS in API mode.

[1] More output is generated when the parameter `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.

The CASS module `GenericProgInfo` 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 `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 AnalysisServer   (analyzeGeneric)
import GenericProgInfo  (lookupProgInfo)
import 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.main`:

```
...> demandedArgumentsOf "Rev" "main"
[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 *curryhome*`/currytools/CASS/Protocol.txt` 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 XML
< 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 57
Overlapping XML
Overlapping CurryTerm
Overlapping Text
Deterministic XML
...
> AnalyzeModule Demand Text Rev
ok 3
append : demanded arguments: 1
main : demanded arguments: 1,2
rev : demanded arguments: 1
> AnalyzeModule Demand CurryTerm Rev
ok 1
```

```
[(("Rev","append"),"demanded arguments: 1"),(("Rev","main"),"demanded arguments: 1,2"),(("Rev","re
> 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>main</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

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

```
Analysis a
```

where `a` denotes the type of analysis results. For instance, the `Overlapping` analysis is implemented as a function

```
overlapAnalysis :: Analysis Bool
```

where the Boolean analysis result indicates whether a Curry operation is defined by overlapping rules.

In order to add a new analysis to CASS, one has to implement a corresponding analysis operation, registering it in the module `Registry` (in the constant `registeredAnalysis`) and compile the modified CASS implementation.

An analysis is implemented as a mapping from Curry programs 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==(pre "?") || 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 enable the inclusion of different analyses in CASS, CASS offers several constructor operations for the abstract type "`Analysis a`" (defined in the CASS module `Analysis`). 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
...
overlapAnalysis :: Analysis Bool
overlapAnalysis = simpleFuncAnalysis "Overlapping" isOverlappingFunction
```

Another analysis constructor supports the definition of a function analysis with dependencies (which is implemented via a fixpoint computation):

```
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.

For instance, a determinism analysis could be based on an abstract domain described by the data type

```
data Deterministic = NDet | Det
```

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—it requires a fixpoint computation. CASS provides such fixpoint computations and requires 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.

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

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

The complete determinism analysis can be specified as

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

This definition is sufficient to execute the analysis with CASS since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, etc. Nevertheless, the analysis must be defined so that the fixpoint computation always terminates. This can be achieved by using an abstract domain with finitely many values and ensuring that the analysis function is monotone w.r.t. some ordering on the values.

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.