CurryInfo: Managing Analysis and Verification Information about Curry Packages

Michael Hanus[0000−0002−4953−8202]

Institut für Informatik, Kiel University, Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. CurryInfo is a tool to manage information about program entities defined in modules of Curry packages. CurryInfo is designed as a generic and extensible system so that the kind of information ranges from syntactic (e.g., comments, types, source code) to semantic aspects (e.g., determinism, termination, total definition, non-failing). CurryInfo collects such information and provides various methods to access them. For instance, one can show this information in a REPL or IDE to help the programmer to select or use operations in an appropriate manner. Another application is to include this information in other tools to analyze or verify programs. Since CurryInfo manages a cache containing information about all Curry packages, this can speed up such tools on larger applications.

1 Introduction

Larger software system are usually not developed from scratch but by re-using many existing pieces of software—typically organized in packages. Although such packages provide an API describing its use, the knowledge of the API information does often not suffice for its correct use. For instance, in a strongly typed programming language, the API is a collection of signatures of operations exported by the package so that the type-correct use of these operations can be checked by the compiler. However, the knowledge about the signature is sometimes not sufficient. For instance, in imperative languages, operations might have side effects which are not explicitly mentioned in their interfaces. Although side effects cannot occur in declarative languages, there are other aspects relevant to the programmer, like determinism or termination behavior, or conditions for the non-failing execution of an operation. Since declarative languages are a good basis to infer or approximate such semantic aspects, one can show them to the programmer instead of manually browsing through the source code.

This is one of the motivations to develop the system described in this paper. As the name indicates, CurryInfo is a system to support the development and analysis of programs written in the multi-paradigm declarative language Curry[1] which amalgamates features of functional and logic programming. Due to this combination, operations have various semantic properties which are not immediately visible from their definition. For instance,

[1] https://www.curry-lang.org

- Operations might be non-deterministic [9], i.e., might yield more than one value for a given argument: although this is an important concept of contemporary functional logic languages (see [3,9] for more details about the advantages of non-deterministic operations), such operations must be used with care in top-level computations involving I/O, since non-deterministic computations need to be encapsulated inside deterministic I/O operations.
- Operations might be non-terminating: in lazy functional languages, computing with non-terminating operations is a feature supporting modularity [20] but they must be used with care. Hence, it is useful to see the termination status of operations during program development.
- Operations might fail on some arguments: when partially defined operations are used, one should either check before the call the admissibility of the arguments or check after the call whether it has failed (e.g., by encapsulated search or exception handlers). Thus, it is important to know whether an operation is totally defined or has a specific non-fail condition [13].

In the language Mercury [23], some of these properties, like non-determinism and possible failures, are part of the source programs and used to generate efficient target code. As a consequence, explicitly defined properties restrict the set of admissible Mercury programs whereas we are interested to keep the flexibility of Curry but approximate semantic properties at compile time [15,16,18]. Since such approximations require non-trivial program analyses based on fixpoint computations, their computation needs some time for larger applications with dozens or hundreds of modules. To avoid a time-consuming computation in an interactive programming environment (e.g., REPL or IDE), CurryInfo provides an infrastructure to compute and collect such information when packages are uploaded, stores this information in a central cache, and provides methods to deliver this information in various formats so that it can be used by different tools. This has the following advantages:

- The use of CurryInfo during program development (e.g., in a REPL or IDE) supports the programmer to consider non-trivial semantic properties of operations when they are applied.
- The use of CurryInfo in other analysis or verification tools can speed up their computations since they can fetch information about imported packages from CurryInfo instead of locally (re)computing it.

After a short introduction to Curry and existing analysis and verification tools in Sect. 2, we survey the information managed by CurryInfo in Sect. 3 and show the basic usage of CurryInfo in Sect. 4. Section 5 describes the structure and implementation of CurryInfo and the methods provided to extend CurryInfo in a modular manner. Section 6 contains our conclusions with a short evaluation.

2 Analysis and Verification of Curry Programs

The declarative language Curry [19] amalgamates features from functional programming (demand-driven evaluation, strong typing, higher-order functions)

and logic programming (computing with partial information, unification, constraints), see [3,10] for surveys. The syntax of Curry is close to Haskell [22]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (where Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. For instance, the following operation inserts an element at an unspecified position into a list:

```
insert :: a → [a] → [a]
insert x []     = [x]
insert x (y:ys) = x : y : ys
insert x (y:ys) = y : insert x ys
```

Note that both of the last two rules can be applied when the second argument of `insert` is a non-empty list. Thus, the expression `insert 0 [1,2]` non-deterministically evaluates to any of the values `[0,1,2]`, `[1,0,2]`, or `[1,2,0]`. `insert` can be used to define permutations of lists in a simple manner:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Note that `perm` is defined by non-overlapping rules, but `perm [1,2,3,4]` non-deterministically evaluates to any of the 24 permutations of the input list. This demonstrates that the (non-)determinism status of `perm` is not directly visible from its signature or definition.

In order to help the programmer to recognize semantic properties of operations which might depend on the behavior of directly or indirectly used operations, there exist various tools to approximate such properties. For instance, the generic analysis system CASS [18] provides an infrastructure for fixpoint computations which is incremental w.r.t. a given modular structure of programs to enable the analysis of larger applications. CASS is based on the idea of abstract interpretation [7] so that abstract domains and the corresponding abstract operations can be defined and plugged into the infrastructure of CASS. Currently, CASS supports more than 30 different analyses[2] on Curry programs to approximate properties like determinism, totally definedness, termination, demanded arguments, groundness, etc. It has been used in various applications, e.g., to transform Boolean equalities into unification to reduce search spaces [5], to detect non-deterministic operations relevant to top-level computations [4], or to optimize programs [6].

Another recent tool targets the safe execution of Curry programs by verifying the absence of failures at compile time [16]. For this purpose, non-fail conditions [13], which approximate arguments that ensure the fail-free execution of operations, are automatically inferred . For instance, consider the operations

```
last :: [a] → a
last [x]      = x
last (_:x:xs) = last (x:xs)
```

```
fac :: Int → Int
fac n | n == 0 = 1
      | n > 0  = n * fac (n-1)
```

[2] See https://cpm.curry-lang.org/webapps/cass/main.cgi?DOC_Analyses for more details.

The tool described in [16] infers that, if the argument of `last` is a non-empty list, it does not fail. This is done by an inference with a domain of abstract types and using properties approximated by CASS. Furthermore, the non-fail condition "`n==0 || n>0`" is inferred for `fac` by combining this approach with solving arithmetic constraints [15], for which an SMT solver [8] is used. If `last` or `fac` are used in other operations, either the non-fail conditions are satisfied in these uses or they imply new non-fail conditions of operations using them. Altogether, the inference and verification of non-fail conditions can be time-consuming on larger applications.

Since Curry is a universal programming language intended to implement larger applications, it has many additional features not described here, like monadic I/O [24] for declarative input/output, set functions [2] to encapsulate non-deterministic search, or functional patterns [1] to specify complex transformations in a high-level manner. Curry programs are structured in modules[3] and modules can be organized in packages The Curry package manager[4] provides access to currently more than 140 packages with several hundred modules.[5] Since non-trivial applications written in Curry are based on dozens of packages (e.g., the Curry package manager is written in Curry and uses 40 packages and more than 130 modules), it is relevant to have information about imported program entities during the development of such applications. This is one of the main motivations for the development of CurryInfo.

3 Information Managed by CurryInfo

CurryInfo is intended to provide an overview of entities defined in Curry packages. This section surveys the entities and the information managed by CurryInfo.

A *package* contains the implementation of some functionality in a particular application domain. Since the implementation and functionality evolves over time, packages come in different *versions* with unique identifiers. The semantic versioning standard[6] is a recommendation to associate meaningful identifiers to different versions of a package. The Curry package manager supports checking these recommendations [12].

A package contains, apart from management information, documentation, or test suites, a set of *modules*. Each module defines *operations* to implement the required functionality. Since Curry is strongly typed with a Haskell-like type system, a module might also contain definitions of *types* (data types or type synonyms) and *classes* (type classes [25] or type constructor classes [21]).

To reflect this structure, CurryInfo manages various information about packages, versions, modules, operations, types, and classes, which are also called *entities* in CurryInfo. For each entity, one can ask CurryInfo for different pieces of

[3] The module system is almost identical to Haskell's module system.
[4] https://curry-lang.org/tools/cpm
[5] https://cpm.curry-lang.org/
[6] http://www.semver.org/

information which are called *requests* of CurryInfo. Some of the requests for the different entities are:[7]

- *package requests*: the name and the identifiers of the versions of a package
- *version requests*: the version identifier, documentation, modules, and package dependencies of this version
- *module requests*: name, documentation, source code, lists of exported operations, types, and classes
- *operation requests*: name, documentation, definition, signature, fixity, and various semantic properties, like determinism, demanded arguments, solution completeness, termination, totally definedness, result values, non-fail conditions
- *type requests*: name, documentation, definition, and list of constructors
- *class requests*: name, documentation, definition, and list of methods

Since CurryInfo is designed to be extensible, new requests can easily be added provided that a tool is available to compute them (see Sect. 5).

4 Usage

Before we show details of the implementation of CurryInfo, we survey various use cases of CurryInfo in this section.

Basically, the executable of CurryInfo can be invoked with options to specify an entity and the requests to be shown for this entity. For instance, requests about the operation `length` defined in the module `Prelude` of package `base` with version `3.3.0` can be shown as follows:

```
curry-info --package=base --version=3.3.0 --module=Prelude
  --operation=length documentation deterministic totally-defined
  terminating demand failfree
```

Requests are computed on-demand or taken from a cache (see Sect. 5) if they have already been computed.

Since this raw access to requests is cumbersome, there is also the tiny wrapper tool `cpm-query`[8] so that one can compute the same information as above by the command

```
cpm-query Prelude length
```

`cpm-query` searches for a package version containing `length` in module `Prelude` according to the current load path. Moreover, the default requests shown for a given kind of entity can be specified in a configuration file. Thus, `cpm-query` provides a simple method to access analysis and verification information by other tools, as shown next.

Curry systems come with an interactive REPL (Read-Eval-Print-Loop) which has commands to load modules or to evaluate expressions. The recent release of

[7] The page https://cpm.curry-lang.org/curry-info/run.cgi?--requests shows the actual list of all requests supported by CurryInfo.

[8] https://cpm.curry-lang.org/pkgs/cpm-query.html

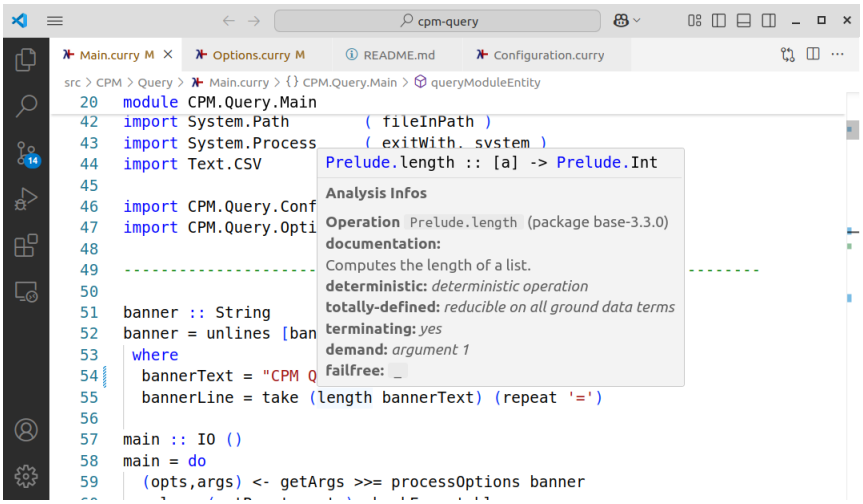

Fig. 1. Showing analysis information of `length` by CurryInfo in Visual Studio Code

PAKCS [17] has a command "`:info`" which invokes `cpm-query` to show requests of operations, types, or classes. For instance, the command

```
> :info length
```

returns the same information as the call to `cpm-query` shown above provided that the `Prelude` operation `length` is in the scope of the REPL expression (otherwise, one has to qualify the operation with its module name).

The most convenient use of CurryInfo is via the Curry Language Server[9] since it supports an extension to invoke any command when hovering over particular program entities, like operations, types, or classes, in a program editor supporting the language server protocol. If this extension is configured to invoke `cpm-query`,[10] hovering over an occurrence of `length` in the editor pops up a window with analysis and verification information provided by CurryInfo. Figure 1 shows a screenshot when hovering over `length` using this extension in Visual Studio Code.

A further use of CurryInfo is to speed up the computation of other analysis or verification tools. For instance, in global program analyses, such as mentioned in Sect. 2, CASS analyzes all imported modules before analyzing the current module [18]. This means that all imported packages are analyzed before the main modules of an application can be analyzed which might be time-consuming for larger applications. A concrete implementation of this usage is evaluated in Appendix A.

[9] https://github.com/fwcd/curry-language-server
[10] See https://github.com/curry-packages/cpm-query/blob/main/README.md

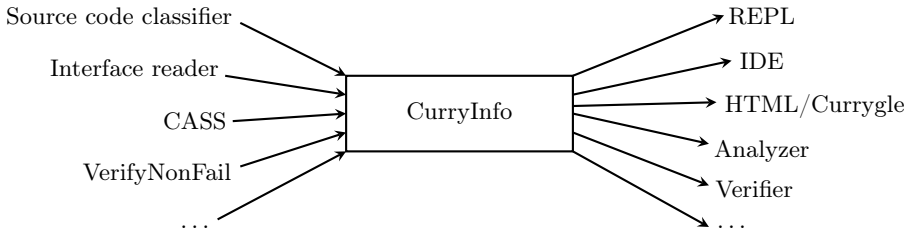

Fig. 2. Structure of the CurryInfo system

5 System Structure and Implementation

CurryInfo provides a generic infrastructure to collect information about Curry packages from different sources and delivers the collected information in various forms, as shown in Fig. 2. Currently, the following tools contribute information:

- Information about documentation comments and definitions of operations, types, and classes in the source text is extracted by a specific source code classifier[11] which performs a lexical and syntactical analysis of Curry programs to group the source text.
- The front end of Curry implementations generates interface files to organize the independent compilation of modules. These interface files are processed by a specific interface reader[12] in order to extract the signature and fixity information of operations defined in modules.
- Analysis and verification information is obtained from external tools, like [15,16,18]. Since these tools can deliver their results also in a standard JSON format, it is easier to parse these results instead of integrating these complex tools in CurryInfo.

In order to get requests from CurryInfo for various purposes, CurryInfo can deliver results in different output formats, like human-readable text, optional with markdown syntax, JSON, or as Curry data terms. Moreover, CurryInfo can be invoked in different modes:

- *command mode* (as shown in Sect. 4): parameters are passed as options and the results are printed on the output stream.
- *server mode*: communication with CurryInfo is done via a socket connection and a specified protocol so that CurryInfo is kept active until the communication is closed.
- *web service mode*: this is similar to the command mode but parameters are passed by the environment variable `QUERY_STRING` so that the executable `curry-info` can be used as a CGI script on a web server. The default installation of this web service is at https://cpm.curry-lang.org/curry-info/. When a new package version is uploaded, this web service will be informed so that it

[11] https://cpm.curry-lang.org/pkgs/curry-source.html
[12] https://cpm.curry-lang.org/pkgs/curry-interface.html

automatically starts to compute all requests for this package and stores this information in its cache.

In its default mode, `cpm-query` contacts this web service to request information. Thus, when using `cpm-query` in a REPL or IDE, as discussed in Sect. 4, it is not necessary to have a local installation of CurryInfo.

CurryInfo also stores all information in human-readable HTML format in static web pages. The Curry API search engine Curr(y)gle[13] delivers references to the information stored in the CurryInfo web pages so that Curry users have immediate access to analysis and verification information of operations defined in published packages.

Since the actual computation of requests can be costly in the case of complex analysis or verification tasks, CurryInfo caches already computed results for program entities. For this purpose, CurryInfo associates a file to each entity which contains a JSON object with request names as keys.[14] For requests dealing with source texts (documentation comments, definitions), only references to the original source files are stored in the JSON objects to limit their sizes.

As already mentioned, CurryInfo is designed to be extensible, i.e., new requests can easily be added to the implementation of CurryInfo. For this purpose, CurryInfo has a configuration module where the requests for the different entities are defined by calls to the generic operation

```
registerRequest :: ConvertJSON b => String  →  String
  →  Generator a b  →  Printer b  →  RegisteredRequest a
```

Here, `a` is the type of entities (package, module, operation,...) and `b` is the type of values of the request. Since these values are stored in the cache in JSON format, the class constraint "`ConvertJSON b`" requires conversion operations to and from JSON. The first argument is an identifier of the request and the second argument a short description (to be shown in the help menu). The third argument is an operation which generates, for an entity of type `a`, the requested value of type `b`. The fourth argument prints a value of the request as a string (according to output format options). Hence, in order to add a new kind of request to CurryInfo, one has to implement a generator for this request, e.g., by using an external analysis tool, and a pretty printer for request values, and add them to the implementation of CurryInfo[15] by using `registerRequest`.

6 Conclusions

We presented CurryInfo, an infrastructure to collect information about entities defined in Curry packages and to distribute them to different applications. CurryInfo is generic in the kind of information so that it can cover syntactic as well

[13] https://cpm.curry-lang.org/currygle/

[14] In principle, one can also use a data base for all entities. Since there is a direct mapping from entities to files, the use of JSON files was advantageous for debugging during the development of CurryInfo.

[15] Available at https://github.com/curry-language/curry-info-system.

as semantic properties of program entities, like information inferred by program analysis or verification systems. Providing information about semantic aspects is quite relevant since computing them is time consuming for larger packages. Therefore, CurryInfo caches this information so that it is immediately available when required by a REPL or IDE.

To get an impression of the usability of CurryInfo in an IDE, we measured the time[16] to execute `cpm-query` to compute the results as shown in Fig. 1. The elapsed time of this execution is 0.3 seconds when using the remote CurryInfo web service (which runs on a different server outside the local network), which is an acceptable time in an IDE. This time reduces to 0.1 seconds when a local installation of CurryInfo is used (for this purpose, the CurryInfo web server allows to download its cache so that it can be locally installed). Although this difference seems relevant, it is hardly recognizable when hovering in an IDE. On the other hand, the on-demand analysis of the same information without CurryInfo requires more than a minute. This clearly demonstrates the advantage of CurryInfo and its information caching. The use of CurryInfo to speed up other analysis tools is discussed and evaluated in Appendix A.

For future work it is interesting to evaluate whether the use of a database for the cache might provide an efficiency improvement, to add further requests that could be helpful to the programmer, like verified contracts [14], or to use CurryInfo in other analysis or verification systems.

References

1. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
2. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
3. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
4. S. Antoy and M. Hanus. Eliminating irrelevant non-determinism in functional logic programs. In *Proc. of the 19th International Symposium on Practical Aspects of Declarative Languages (PADL 2017)*, pages 1–18. Springer LNCS 10137, 2017.
5. S. Antoy and M. Hanus. Transforming boolean equalities into constraints. *Formal Aspects of Computing*, 29(3):475–494, 2017.
6. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[16] We used a Linux machine running Ubuntu 22.04 with an Intel Core i7-1165G7 (2.80GHz) processor with eight cores. The tools were executed with KiCS2 [6].

8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer LNCS 4963, 2008.
9. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
10. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
11. M. Hanus. CurryCheck: Checking properties of Curry programs. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, pages 222–239. Springer LNCS 10184, 2017.
12. M. Hanus. Semantic versioning checking in a declarative package manager. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, OpenAccess Series in Informatics (OASIcs), pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
13. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
14. M. Hanus. Combining static and dynamic contract checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.
15. M. Hanus. Hybrid verification of declarative programs with arithmetic non-fail conditions. In *Proc. of the 22nd Asian Symposium on Programming Languages and Systems (APLAS 2024)*, pages 109–129. Springer LNCS 15194, 2024.
16. M. Hanus. Inferring non-failure conditions for declarative programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*, pages 167–187. Springer LNCS 14659, 2024.
17. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. PAKCS: The Portland Aachen Kiel Curry System. Available at https://www.curry-lang.org/pakcs/, 2025.
18. 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.
19. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-lang.org, 2016.
20. J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
21. M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.
22. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
23. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
24. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
25. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, 1989.

A Using CurryInfo in Analysis Tools

As mentioned in Sect. 4, the information managed by CurryInfo can be useful to speed up the computation of other analysis or verification tools. To evaluate this potential usage, we modified the Curry analysis system CASS [18] by adding an option to force CASS to query CurryInfo for required analysis information: if CurryInfo has this information available, it will be copied from CurryInfo instead of computing it locally by CASS. One can also choose to contact the CurryInfo web service or a local installation of CurryInfo.

The subsequent table shows the timings of these different usages (on the same machine as described in Sect. 6). We compared the elapsed times (in seconds) of two implementations of CASS: one compiled with PAKCS [17] (which compiles to Prolog and uses SICStus-Prolog 4.9.0 as back end) and one compiled with KiCS2 [6] (which compiles to Haskell and uses GHC 9.4.5 as back end). For each Curry application shown in the first column, we show the timings for the analysis without CurryInfo (*no CI*), with the CurryInfo web service (*web*), and a local installation of CurryInfo (*local*). Moreover, the "Modules" column shows the total number of modules of the application (*all*), the number of modules from imported packages (*imp*) where the results can be copied from CurryInfo, and the number of local modules (*loc*) which must always be analyzed by CASS. The applications used in these tests are the CurryInfo system, CASS [18], the inference and verification system for non-fail conditions [15,16], the property-based testing tool CurryCheck [11], the Curry Package Manager CPM [12], and cpm-manage, a tool to generate the web pages of the CPM repository. For all these applications, we analyzed the determinism behavior of the operations in the main module.

	Modules			CASS (PAKCS)			CASS (KiCS2)		
	all	*imp*	*loc*	*no CI*	*web*	*local*	*no CI*	*web*	*local*
CurryInfo	83	58	25	6:48.81	3:14.27	3:01.10	19.85	23.01	11.79
CASS	89	74	15	7:47.93	3:17.15	4:47.05	17.70	23.77	9.43
VerifyNonFail	122	103	19	14:52.85	8:07.68	4:42.85	29.98	34.80	15.16
CurryCheck	122	111	11	9:57.19	4:51.37	3:41.69	25.33	34.57	13.25
CPM	134	105	29	8:36.62	6:42.24	4:56.29	32.45	37.67	17.25
cpm-manage	64	59	5	6:01.36	3:41.91	2:27.37	14.03	19.51	8.09

As one can see, the speed-up obtained by copying analysis information from CurryInfo depends on the implementation of CASS, the network connection or time to copy and parse the analysis results, and the complexity of the analysis process. For instance, KiCS2 is quite efficient for purely functional programs. Hence, the network connection to the CurryInfo web service causes a substantial delay so that the use of the local installation of CurryInfo is much faster.[17]

One can expect that for more complex systems, like the inference and verification of non-fail conditions, where also external SMT-solvers are used [15], the efficiency improvements are higher than for the CASS analyses tested above.

[17] Due to this reason, the CurryInfo web service provides a copy of its cache which can easily be downloaded and locally installed.