Improving Residuation in Declarative Programs*

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

November 2, 2018

Abstract. Residuation is an operational principle to evaluate functions in logic-oriented languages. Residuation delays function calls until the arguments are sufficiently instantiated in order to evaluate the function deterministically. It has been proposed as an alternative to the non-deterministic narrowing principle and is useful to connect externally defined operations. Residuation can be implemented in Prolog systems supporting coroutining, but this comes with a price: the coroutining mechanism causes a considerable overhead even if it is not used. To overcome this dilemma, we propose a compile-time analysis which approximates the run-time residuation behavior. Based on the results of this analysis, we improve an existing implementation of residuation and evaluate the potential efficiency gains by a number of benchmarks.

1 Introduction

Declarative programming is an attempt to build reliable software systems in a high-level manner on sound theoretical principles. Functional languages support functions as programming entities and use reduction for evaluation. Logic languages support relations as main entities and use unification-based resolution for evaluation. When combining both kinds of languages in order to provide a single declarative language, there are two principle choices for evaluation. *Narrowing* extends reduction by unification so that functions can also be invoked with partially known arguments. Thus, functions might be evaluated non-deterministically like relations in logic programming. *Residuation* restricts non-deterministic evaluation to predicates only so that functions are suspended if their arguments are not sufficiently instantiated for deterministic reduction. Both operational mechanisms are applied in functional logic languages that combine the most important features of functional and logic programming in a single language (see [8,18] for recent surveys).

Both narrowing and residuation have their justifications. Optimal evaluation strategies are known for narrowing [4] whereas residuation supports concurrent computations and allows to connect externally defined operations in a declarative manner [10]. This motivated the development of the functional logic language

* The research described in this paper has been partially supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH15006B.

Curry [22] as a unified language for functional logic programming which combines narrowing and residuation in a single evaluation principle [17]. A functional logic language can be implemented with limited efforts by compiling it into Prolog (e.g., [5,13,24]). To implement residuation, one can exploit coroutining facilities supported by many Prolog systems: if some argument of a residuating operation is not sufficiently instantiated, the evaluation of the corresponding Prolog predicate is suspended so that another predicate can be activated [5]. Although this implementation is quite simple, it causes additional costs if residuation is not used, i.e., all function calls are sufficiently instantiated, since one has to check each function call before activating them. To avoid these costs, we develop in this paper a compile-time analysis which approximates operations w.r.t. their runtime residuation behavior. Based on this analysis, we improve a Curry compiler and evaluate the efficiency gains for some benchmarks.

This paper is structured as follows. After a short introduction to functional logic programming and Curry, we sketch in Sect. 3 an existing implementation of residuation in Prolog. Our compile-time analysis of residuation is described in Sect. 4. The improved implementation of residuation w.r.t. analysis information is sketched in Sect. 5 and evaluated in Sect. 6. Section 7 discusses related work before we conclude in Sect. 8.

2 Declarative Programming with Curry

The declarative programming language Curry [22] amalgamates the most important features of functional and logic programming as well as operational principles of combined functional logic languages [8,18], such as narrowing and residuation, in a single language. Conceptually, Curry extends Haskell [28] with nondeterminism, free variables, and constraint solving. Thus, the syntax of Curry is close to Haskell but Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner and allows *free* (*logic*) *variables* in conditions and right-hand sides of rules. These variables must be explicitly declared unless they are anonymous. Similarly to Haskell, functions are evaluated lazily to support modular programming, infinite data structures, and optimal evaluation [4]. Unlike Haskell, function calls might contain free (unbound) variables, i.e., without a value at call time. If the value of such an argument is demanded, the function call is either suspended (which corresponds to residuation) or the variable is non-deterministically instantiated (which corresponds to narrowing [4]).

Example 1. Consider the following definition of natural numbers in Peano representation, the addition on natural numbers, and a predicate which is true on natural numbers:

```
data Nat = Z | S Nat

add :: Nat → Nat → Nat          nat :: Nat → Bool
add Z     y = y                 nat Z     = True
add (S x) y = S (add x y)       nat (S x) = nat x
```

If `add` is evaluated by narrowing (such functions are also called *flexible*), the equation

```
let x free in add x (S Z) == S (S Z)
```

is solved by instantiating the free variable `x` to `(S Z)`. However, if `add` is evaluated by residuation (in this case `add` is called *rigid*), the equation solving suspends. To proceed with suspended computations, Curry has a *concurrent conjunction* operator "`&`" which evaluates both arguments concurrently, i.e., if the evaluation of one argument suspends, the other is evaluated. Thus, if the function `add` is rigid and the predicate `nat` is flexible (as in languages like Le Fun [2] or Oz [30]), the conjunction

```
let x free in add x (S Z) == S (S Z) & nat x
```

is successfully evaluated by interleaving the evaluation of `add` and `nat` (which instantiates `x` to `(S Z)`). This kind of concurrent computation is also called *declarative concurrency* [31].

In the first version of Curry, functions were rigid and predicates flexible by default, similarly to residuation-based languages [1,2,26,30]. Later, narrowing became the default for all defined operations so that only externally defined operations and conditionals, like "`if-then-else`" or "`case-of`", are evaluated by residuation. There is also an explicit "suspension" combinator for concurrent programming: `ensureNotFree` returns its argument evaluated to head normal form but suspends as long as the result is a free variable.

Curry has many additional features not described here, like monadic I/O [32] for declarative input/output, set functions [7] to encapsulate non-deterministic search, functional patterns [6] and default rules [9] to specify complex transformations in a high-level manner, and a hierarchical module system together with a package manager[1] that provides access to currently more than 80 packages with several hundred modules.

3 Implementing Residuation in Prolog

A scheme to compile functional logic languages with residuation, such as Curry, into Prolog is proposed in [5] and used in the Curry implementation PAKCS [20] which is part of recent Debian und Ubuntu Linux distributions. Coroutining features of contemporary Prolog systems can be exploited to implement residuating operations. SICStus-Prolog[2] offers `block` declarations to enforce the suspension of predicate calls under particular conditions. For instance, the declaration "`:- block p(?,-,?).`" specifies that a call to `p` is delayed if the second argument is a free variable. Thus, the following code defines the multiplication function on integers as a predicate which suspends if one of the arguments is a free variable:

```
:- block mult(-,?,?), mult(?,-,?).
```

[1] http://curry-language.org/tools/cpm
[2] http://sicstus.sics.se/

```
mult(X,Y,R) :- R is X*Y.
```

An alternative to block declarations is the predicate `freeze(X,G)` which suspends the evaluation of the goal `G` if the first argument `X` is a free variable. Since `freeze` is less efficient than `block` [5], PAKCS uses `block` declarations when it compiles to SICStus-Prolog and `freeze` declarations when it compiles to SWI-Prolog (since SWI-Prolog does not offer `block` declarations).

Unfortunately, such simple `block` declarations are not sufficient when compiling functional logic programs into Prolog due to nested function calls. Since functions are compiled into Prolog predicates by adding a result argument and evaluating demanded inner arguments before outer function calls [5,13], it must be ensured that all predicates involved in a function call are suspended when some argument suspends. For instance, consider the Curry program

```
g x = ensureNotFree x

h []     = []
h (y:ys) = h ys

main x = h (g x)
```

If we evaluate `main x` where `x` is a free variable, the evaluation of `(g x)` suspends due to the call of `ensureNotFree`. Hence, the calls to `h` and, thus, `main` also suspend. The Prolog code obtained by translating functions into predicates is[3]

```
:- block g(-,?).
g(X,R) :- R=X.

h(A,R) :- hnf(A,B), h_1(B,R).
h_1([],R) :- R=[].
h_1([Y|Ys],R) :- h(Ys,R).

main(A,R) :- h(g(A),R).
```

The evaluation of `main(A,R)` leads to the evaluation of `hnf(g(A),B)` and `g(A,B)`, which suspends. However, the subsequent literal `h_1(B,R)` can still be evaluated which results in an infinite search space by applying the second rule of `h_1` forever.

In order to avoid such problems, more control is needed so that the call to `h_1` is activated only if the evaluation of the call to `g` is not suspended. For this purpose, [5] proposes to add specific input and output arguments to each predicate. These arguments are either uninstantiated or bound to the constant `eval` (the actual value is irrelevant). A computation of a predicate (implementing some Curry function) is activated only if the input control argument is instantiated. If this computation is complete, i.e., without suspension, the output control argument is bound to `eval`. Thus, one can implement the required control by chaining these control arguments through the program. As a concrete example, our program above is implemented as follows:

[3] The predicate `hnf` computes the head normal form of its first argument. It can be defined by a case distinction on all function and constructor symbols in the program. The use of `hnf` instead of simply flattening nested function calls is essential to implement lazy evaluation.

```
:- block g(-,?,?,?), g(?,?,-,?).
g(X,R,E0,E1) :- R=X, E1=E0.

:- block h(?,?,-,?).
h(A,R,E0,E2) :- hnf(A,B,E0,E1), h_1(B,R,E1,E2).

:- block h_1(?,?,-,?).
h_1([],R,E0,E1) :- R=[], E1=E0.
h_1([Y|Ys],R,E0,E1) :- h(Ys,R,E0,E1).

:- block main(?,?,-,?).
main(A,R,E0,E1) :- h(g(A),R,E0,E1).
```

Now, the evaluation of the goal `main(A,R,eval,E)` suspends where the call to `h_1` is also suspended.

This scheme together with a sophisticated implementation of sharing (see [5] for details) is the basis of the Curry implementation PAKCS [20]. However, this implementation has some cost since *every* predicate is annotated with a `block` declaration. The costs are even higher when `block` declarations are replaced by `freeze` declarations (as shown later by our benchmarks). On the other hand, residuation is not a dominating principle in actual programs. Originally, residuation has been proposed as an alternative to narrowing in order to avoid evaluating functions in a non-deterministic manner, see, for instance, the languages Escher [23], Le Fun [2], Life [1], NUE-Prolog [26], or Oz [30]. Since the language Curry is an attempt to unify the different approaches to combine functional and logic programming, it supports residuation and narrowing in a unified way [17]. As time has passed, residuation became less important so that functions are now non-residuating by default [18]. Nevertheless, residuation is still interesting to support concurrent computations and to connect externally defined operations in a declarative way [10]. This demands for an implementation where the overhead of residuation is accepted only if it is actually used in the program. For this purpose, we develop in the next section a program analysis to approximate the actual usage of residuation during run time in a Curry program.

4 Approximating Residuation Behavior

In order to improve the implementation of potentially residuating programs sketched above, it is important to characterize programs or part of programs where residuation is not used. This is the case if residuating functions are not invoked at run time or they are invoked with sufficiently instantiated arguments. Since such properties are obviously undecidable, we develop a compile-time technique to approximate them.

4.1 CASS: An Analysis Framework for Curry

CASS [21] is an incremental and modular analysis system for Curry programs. Since CASS provides a good infrastructure to implement new program analyses, we will use it for our purpose. A new program analysis can be added to CASS if it is defined in a bottom-up manner, i.e., the analysis computes some abstract

$$
\begin{array}{lll}
P & ::= & D_1 \ldots D_m & \text{(program)} \\
D & ::= & f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & | & f(e_1, \ldots, e_n) & \text{(function call)} \\
 & | & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
 & | & \mathit{fcase}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)} \\
 & | & e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
 & | & \mathit{let}\ x\ \mathit{free}\ \mathit{in}\ e & \text{(free variable)} \\
 & | & \mathit{let}\ x = e\ \mathit{in}\ e' & \text{(let binding)} \\
p & ::= & c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Fig. 1. Syntax of the intermediate language FlatCurry [18]

information about a given operation from the definition of this operation together with abstract information about the operations used in this definition. Then CASS performs the necessary fixpoint computations, incremental analysis of imported modules, etc, to analyze a given module.

To be more precise, an analysis added to CASS must be defined on an intermediate language, called FlatCurry, which is used in compilers, optimization, and verification tools, and to specify the operational semantics of Curry programs [3]. In FlatCurry, the syntactic sugar of the source language is eliminated and the pattern matching strategy is explicit. The abstract syntax of FlatCurry is summarized in Fig. 1. A FlatCurry program consists of a sequence of function definitions, where each function is defined by a single rule. We assume that all variables introduced in the left-hand side, patterns, and let expressions are disjoint. Patterns in source programs are compiled into flexible case expressions and overlapping rules are joined by explicit disjunctions. The difference between *case* and *fcase* corresponds to residuation and narrowing: when the argument e evaluates to a free variable, *case* suspends whereas *fcase* nondeterministically binds this variable to a pattern in a branch of the case expression.

Any Curry program can be translated into a FlatCurry program by making the pattern matching strategy explicit. For instance, the operation `h` defined in Section 3 has the following FlatCurry representation:

$$\texttt{h(xs)} \;=\; \mathit{fcase}\ \texttt{xs}\ \mathit{of}\ \{\ \texttt{[]} \to \texttt{[]};\ \ \texttt{y:ys} \to \texttt{h(ys)}\ \}$$

In principle, let bindings as shown in Fig. 1 are not required to translate standard Curry programs. However, they can be used to translate circular data structures and are convenient to express sharing without the use of complex graph structures [14,15]. Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential for lazy computations [3].

4.2 A Domain for Residuation Analysis

We use FlatCurry to specify our analysis of the residuation behavior of Curry programs. Since residuation, i.e., the suspension of function calls, might occur if

variables are unbound during run time, we have to approximate which arguments are ground at run time and under which conditions functions do not residuate. For instance, the addition operation `(+)` does not residuate if both arguments are ground. However, the constant function

```
const :: a → _ → a
const x y = x
```

does not residuate if the first argument is ground and the second argument is arbitrary, since the latter is not evaluated by `const` due to lazy evaluation. Therefore, our analysis associates to each n-ary operation f a set $f^\alpha \subseteq \{1, \ldots, n\}$ with the following interpretation: if $e = f\ t_1 \ldots t_n$ and t_i is a ground constructor term for each $i \in f^\alpha$, then the evaluation of e does not suspend and each value of e is ground. For instance, $\texttt{+}^\alpha = \{1, 2\}$ and $\texttt{const}^\alpha = \{1\}$. Since there are also operations without such a strong property or where our analysis is not precise enough, we also add a top element $\top$. $f^\alpha = \top$ means that a call to f might residuate or does not yield a ground term.[4] Finally, there is also an abstract bottom element $\bot$, representing no information, which is used to start the fixpoint analysis. Thus, our analysis associates to an n-ary operation f an element of the abstract domain

$$\{\bot, \top\} \cup \{s \mid s \subseteq \{1, \ldots, n\}\}$$

Note that $\bot$ has a different meaning than $\varnothing$. $\bot$ means "unknown" or "no analysis result" (e.g., $\texttt{loop}^\alpha = \bot$ for the definition `loop = loop`), whereas $\varnothing$ means "no suspension." For instance, if $\texttt{main}^\alpha = \varnothing$, then the evaluation of `main` will never residuate. Similarly, if f is n-ary, the abstract value $\{1, \ldots, n\}$ is different from $\top$. If $f^\alpha = \{1, \ldots, n\}$, then $f\ t_1 \ldots t_n$ does not suspend and has a ground result value if each $t_1, \ldots, t_n$ are ground constructor terms, whereas $f^\alpha = \top$ means that any call to f might residuate or does not yield a ground term.

Abstract elements are ordered as usual, i.e., $\bot \sqsubseteq x$, $x \sqsubseteq \top$, and, for $\bot \neq s_i \neq \top$ $(i = 1, 2)$, $s_1 \sqsubseteq s_2$ iff $s_1 \subseteq s_2$. Consequently, the least upper bound $s_1 \sqcup s_2$ of two abstract elements s_1, s_2 is defined as follows:

$$s_1 \sqcup s_2 = \begin{cases} s_1 & \text{if } s_2 = \bot \\ s_2 & \text{if } s_1 = \bot \\ \top & \text{if } s_1 = \top \text{ or } s_2 = \top \\ s_1 \cup s_2 & \text{otherwise} \end{cases}$$

4.3 Residuation Analysis

The analysis of a single operation f uses an assumption $\mathcal{A}$ which maps operations and variables into abstract elements. $\mathcal{A}[x \mapsto \alpha]$ denotes the extended assumption $\mathcal{A}'$ which is defined by

$$\mathcal{A}'(y) = \begin{cases} \alpha & \text{if } y = x \\ \mathcal{A}(y) & \text{if } y \neq x \end{cases}$$

[4] One could also refine the abstract domain in order to distinguish between residuation and non-ground results, but this does not seem to provide better results in practice.

$$\textit{FDecl}\quad \frac{\mathcal{A}[x_1 \mapsto \{1\}, \ldots, x_n \mapsto \{n\}] \vdash e : \alpha}{\mathcal{A} \vdash f(x_1, \ldots, x_n) = e : \alpha}$$

$$\textit{Var}\quad \mathcal{A} \vdash x : \mathcal{A}(x)$$

$$\textit{Cons}\quad \frac{\mathcal{A} \vdash e_1 : \alpha_1 \ \ldots \ \mathcal{A} \vdash e_n : \alpha_n}{\mathcal{A} \vdash c(e_1, \ldots, e_n) : \alpha_1 \sqcup \ldots \sqcup \alpha_n}$$

$$\textit{Fun}\quad \frac{\mathcal{A} \vdash e_1 : \alpha_1 \ \ldots \ \mathcal{A} \vdash e_n : \alpha_n}{\mathcal{A} \vdash f(e_1, \ldots, e_n) : \alpha}$$

where $\alpha = \alpha_{i_1} \sqcup \ldots \sqcup \alpha_{i_k}$ if $\mathcal{A}(f) = \{i_1, \ldots, i_k\}$, otherwise $\alpha = \mathcal{A}(f)$

$$\textit{Case}\quad \frac{\mathcal{A} \vdash e : \alpha \qquad \mathcal{A}_1 \vdash e_1 : \alpha_1 \quad \ldots \quad \mathcal{A}_n \vdash e_n : \alpha_n}{\mathcal{A} \vdash \textit{case } e \textit{ of } \{p_1 \to e_1; \ldots; p_n \to e_n\} : \alpha \sqcup \alpha_1 \sqcup \ldots \sqcup \alpha_n}$$

where $\mathcal{A}_i = \mathcal{A}[x_{i_1} \mapsto \alpha, \ldots, x_{i_{k_i}} \mapsto \alpha]$ if $p_i = c_i(x_{i_1}, \ldots, x_{i_{k_i}})$

$$\textit{FCase}\quad \frac{\mathcal{A} \vdash e : \alpha \qquad \mathcal{A}_1 \vdash e_1 : \alpha_1 \quad \ldots \quad \mathcal{A}_n \vdash e_n : \alpha_n}{\mathcal{A} \vdash \textit{fcase } e \textit{ of } \{p_1 \to e_1; \ldots; p_n \to e_n\} : \alpha \sqcup \alpha_1 \sqcup \ldots \sqcup \alpha_n}$$

where $\mathcal{A}_i = \mathcal{A}[x_{i_1} \mapsto \alpha, \ldots, x_{i_{k_i}} \mapsto \alpha]$ if $p_i = c_i(x_{i_1}, \ldots, x_{i_{k_i}})$

$$\textit{Or}\quad \frac{\mathcal{A} \vdash e_1 : \alpha_1 \ \ldots \ \mathcal{A} \vdash e_2 : \alpha_2}{\mathcal{A} \vdash e_1 \textit{ or } e_2 : \alpha_1 \sqcup \alpha_2}$$

$$\textit{Free}\quad \frac{\mathcal{A}[x \mapsto \top] \vdash e : \alpha}{\mathcal{A} \vdash \textit{let } x \textit{ free in } e : \alpha}$$

$$\textit{Let}\quad \frac{\mathcal{A}[x \mapsto \bot] \vdash e : \alpha \qquad \mathcal{A}[x \mapsto \alpha] \vdash e' : \alpha'}{\mathcal{A} \vdash \textit{let } x = e \textit{ in } e' : \alpha'}$$

Fig. 2. Abstract semantics for residuation analysis

To analyze a function f w.r.t. an assumption $\mathcal{A}$ about operations defined in the program, we apply the inference rules shown in Fig. 2. Rule *FDecl* is the main rule to analyze a function defined by $f(x_1, \ldots, x_n) = e$. For this purpose, the right-hand side e is analyzed with the assumption extended by information about the position of the argument variables. Rule *Var* simply returns the abstract element associated to the variable so that we obtain the information of arguments passed as results. For instance, rules *FDecl* and *Var* are sufficient to derive the judgement $\mathcal{A} \vdash \texttt{const}(x, y) = x : \{1\}$. Rule *Cons* combines the information of all arguments by returning their least upper bound. Hence, one can derive the judgement $\mathcal{A} \vdash \texttt{pair}(x, y) = (x, y) : \{1, 2\}$ showing that `pair` does not residuate and returns a ground value if both arguments are ground. In case of function applications (rule *Fun*), the computation of the least upper bound can be restricted to the arguments required by the assumption about the function. Rules *Case* and *FCase* simply combine the information of the discriminating expression and all branches since all of these expressions might contribute to

the overall result. Note that the operational difference between *case* and *fcase* is not considered here since our abstract domain does not distinguish between non-ground and possibly residuating expressions. Such a distinction could be introduced in principle, but practical evaluations showed that such a refined domain does not yield more useful results for our intended application. Rule *Or* combines both branches since both will be executed at run time. Rule *Free* assigns the top element to the introduced variable so that the overall result will be "possibly non-ground/suspending" if this variable is used. Finally, rule *Let* analyzes the bound expression with no information about the bound variable and, then, analyzes the main expression with the information computed for the bound variable.

External operations, like "`+`", are not explicitly mentioned in the analysis rules. They can be simply covered by defining $\mathcal{A}(f) = \{1, \ldots, n\}$ for each external operation f of arity n. This is a correct approximation for all currently supported external operations, since they do not residuate and yield a ground value if all arguments are ground values.

One might wonder why higher-order functions are not explicitly mentioned in the analysis rules. This is because they can be transformed into first-order ones by providing an "apply" operation between two expressions (this technique is known as "defunctionalization" [29] and also used to extend logic programs with higher-order features [33]). In this implementation, partially applied function calls are considered as constructor applications where the operation `apply` adds an argument and, if all arguments are provided, calls the actual function. Thus, partial applications are analyzed by *Cons* and `apply` is considered as a predefined operation with $\mathcal{A}(\texttt{apply}) = \{1, 2\}$. The only disadvantage of this simple approach is a possible over-approximation since *all* arguments of a partial application are assumed to be evaluated. For instance, consider the definitions

```
f x y = y+1
main = map (f x) [1,2,3]  where x free
```

Then the analysis yields the result that `main` might residuate although `main` evaluates to a ground value. This over-approximation can be avoided by the following specialized rule for partial applications which takes into account the abstract information about functions even for partial applications:

$$PFun \quad \frac{\mathcal{A} \vdash e_1 : \alpha_1 \ \ldots \ \mathcal{A} \vdash e_m : \alpha_m}{\mathcal{A} \vdash f(e_1, \ldots, e_m) : \alpha} \qquad f \text{ } n\text{-ary function and } m < n$$

$$\text{where } \alpha = \bigsqcup\{\alpha_i \mid i \in \mathcal{A}(f) \text{ and } i \leq m\}, \text{ otherwise } \alpha = \mathcal{A}(f)$$

An assumption $\mathcal{A}$ is correct if, for all operations f defined by $f(x_1, \ldots, x_n) = e$, $\mathcal{A}(f) = \alpha$ and $\mathcal{A} \vdash f(x_1, \ldots, x_n) = e : \alpha$ is derivable. Since all abstract operations used in Fig. 2 are monotone and the abstract domain is finite, we can compute a correct assumption as a least fixpoint by starting with the initial assumption $\mathcal{A}_0(f) = \bot$ for all operations f. Since such fixpoint computations are supported by CASS, the residuation analysis can be implemented by encoding the rules of Fig. 2 in a straightforward way and adding this code to CASS. The

analysis is available in the current implementation of CASS (install package `cass` with the Curry package manager [19]) or via the online version of CASS.[5]

The soundness of the residuation analysis can be proved by induction on the evaluation steps of the concrete semantics. For this purpose, one has to extend the operational semantics of Curry programs presented in [3] to cover suspended computations by returning a specific SUSPEND result when the discriminating expression of a *case* expression is a free variable. Then one can show that an expression e will not be evaluated to SUSPEND if all variables required to be ground by the analysis of e are actually bound to ground expressions, where the latter means that these expressions evaluate to ground values (and not to SUSPEND). Thus, if $\mathcal{A} \vdash \mathit{main} : \varnothing$ is a correct judgement, then the evaluation of *main* never suspends. The detailed definitions and proofs can be found in the appendix.

In order to evaluate the precision of the presented analysis, we analyzed the system libraries distributed with PAKCS. For instance, the library `Prelude`, which is the largest one and contains the predefined standard operations of Curry, contains 867 operations (including auxiliary operations that are not exported) but only one operation has the analysis result $\top$: the operation `unknown` which yields a fresh variable:

```
unknown = let x free in x
```

For all other operations, our analysis yields an argument index set, i.e., these operations do not residuate if they are called with ground values. When analyzing all system libraries, only 25 of 2616 operations might residuate or yield non-ground values. These are mainly operations in logic-oriented libraries for combinatorial programming or encapsulated search.

5 Implementing Residuation with Analysis Information

In this section we sketch how one can use the results of the residuation analysis to improve the implementation of residuation shown in Sect. 3.

As discussed in Sect. 3, suspension declarations, like `block` or `freeze`, are necessary to suspend the evaluation of an expression if some of its demanded subexpressions are suspended. However, if it is ensured that these subexpressions do not suspend, the run-time checking of suspension declarations become superfluous since the conditions under which they fire (i.e., suspend a goal) are never satisfied. Unfortunately, these conditions might not be the same in all calls to an operation. For instance, the factorial function can be evaluated without residuation if its argument is a number, but it is suspended when it is called with an unbound variable until this variable is instantiated by some other thread. Thus, if we want to keep the overall functionality of a program but improve it on calls with sufficiently instantiated arguments, we have to duplicate the code: in addition to the original Prolog code, we add code without suspension declarations which is activated only on sufficiently instantiated arguments. If the compiler

[5] `https://www-ps.informatik.uni-kiel.de/~mh/webcass/`

uses the information of the residuation analysis, calls to the appropriate version of the code can be generated.

For instance, consider the translation of a function `f(x) = g(x,h(0))`. We add the suffix `_NR` to Prolog predicates implementing non-residuating code. If $\mathtt{f}^\alpha \neq \top$, there are sufficient conditions to evaluate `f` without residuation so that we generate the non-residuating version of the code:[6]

```
f_NR(X,R,E0,E1) :- g_NR(X,h_NR(0),R,E0,E1).
```

Thus, non-residuating code always calls other non-residuating code. If $\mathtt{f}^\alpha = \{1\}$, the predicate `f_NR` is invoked when `f` is called with an expression that evaluates to a ground value and does not suspend.

However, `f` might also be called with non-ground arguments or expressions which residuate. For this purpose, we also need the standard code for `f` but we can improve the translation of some subexpressions if they are non-residuating. For instance, if $\mathtt{h}^\alpha = \{1\}$, the following code is generated:

```
:- block f(?,?,-,?).
f(X,R,E0,E1) :- g(X,h_NR(0),R,E0,E1).
```

If $\mathtt{g}^\alpha = \{2\}$, i.e., `g` does not use its first argument, the code can be improved even more:

```
:- block f(?,?,-,?).
f(X,R,E0,E1) :- g_NR(X,h_NR(0),R,E0,E1).
```

Thus, standard code can call non-residuating code but not vice versa.

Although the code duplication is a slight a drawback of our approach, it is acceptable in practice since Prolog compilers often generate compact executables, e.g., by using virtual machine (WAM) instructions. This is shown in the next section where we evaluate our approach in a concrete compiler.

6 Benchmarks

In order to evaluate our approach, we added to PAKCS [20], which compiles Curry programs into Prolog programs based on the scheme sketched in Sect. 3 and described in detail in [5], a compilation flag to select one of the following three *residuation compilation modes*:

Full residuation: This is the existing compilation scheme sketched in Sect. 3 where `block` declarations (if SICStus-Prolog is used as the back end) or `freeze` goals (if SWI-Prolog is used as the back end) are used in the translation of all Curry operations.

No residuation: In this compilation mode, `block` and `freeze` are completely omitted. Instead, run-time errors are emitted in cases where residuation should occur according to the definition of Curry. Although this mode

[6] Although the control arguments `E0` and `E1` are superfluous in non-residuating computations, we leave them in the code in order to simplify the interaction of residuating and non-residuating code. Improving this scheme is a topic for future work.

	Full Res.	No Residuation		Optimized Resid.	
Program	Time	Time	Speedup	Time	Speedup
ReverseUser	14.72	9.07	1.62	9.10	1.62
Reverse	13.06	7.75	1.69	7.77	1.68
TakPeano	6.09	2.89	2.11	4.00	1.52
Tak	4.68	3.34	1.40	4.00	1.17
ReverseHO	3.55	3.03	1.17	3.32	1.07
Primes	11.42	7.90	1.45	9.82	1.16
PrimesPeano	7.89	3.82	2.07	4.14	1.91
Queens	9.74	6.36	1.53	7.84	1.24
QueensUser	10.61	6.63	1.60	8.13	1.31
PermSort	9.57	6.50	1.47	7.81	1.23
PermSortPeano	6.20	3.18	1.95	4.84	1.28
RegExp	5.18	4.06	1.28	4.39	1.18

Fig. 3. Run times (in seconds) and speedups with SICStus-Prolog

changes the semantics of Curry, it shows the best efficiency gain which can be obtained by removing coroutining annotations.

Optimized residuation: This is the compilation mode described in the previous section, i.e., the code generated for each operation is duplicated and the non-residuating code is invoked depending on the results of the program analysis described in Sect. 4.

All benchmarks were executed on a Linux machine (Debian 9.4) with an Intel Core i7-7700K (4.20Ghz) processor and 32GiB of memory. The Curry implementation PAKCS (Version 2.0.2) uses SICStus-Prolog (Version 4.3.5) or SWI-Prolog (Version 7.6.4) as back ends. Timings were performed with the Unix `time` command measuring the execution time to compute all solutions (in seconds) of a compiled executable for each benchmark as a mean of three runs.

The concrete benchmarks are Curry programs that were already used to compare different Curry implementations [12]. "ReverseUser" is the naive list reverse program applied to a list of 16384 elements, where all data (lists, numbers) are user-defined. "Reverse" is the same but with built-in lists. "Tak" is a highly recursive function on naturals [27] applied to arguments (24,16,8) and "TakPeano" is the same but with user-defined natural numbers in Peano representation (see Example 1) so that no built-in arithmetic operations are used. "ReverseHO" reverses a list with one million elements in linear time using higher-order functions like `foldl` and `flip`. "Primes" computes the 2000th prime number via the sieve of Eratosthenes using higher-order functions, and "PrimesPeano" computes the 256th prime number but with Peano numbers and user-defined lists. "Queens" (and "QueensUser" with user-defined lists) computes the number of safe positions of 10 queens on a 10×10 chess board. Finally, "PermSort" sorts a list containing 15 elements by enumerating all permutations and selecting the sorted ones ("PermSortPeano" does the same for Peano numbers and 14 elements), and "RegExp" matches a regular expression in a string of length 400,000 following

	Full Res.	No Residuation		Optimized Resid.	
Program	Time	Time	Speedup	Time	Speedup
ReverseUser	136.47	29.65	4.60	29.85	4.57
Reverse	133.62	29.01	4.61	28.29	4.72
TakPeano	53.76	16.91	3.18	23.00	2.34
Tak	42.97	24.90	1.73	32.52	1.32
ReverseHO	17.97	8.16	2.20	9.83	1.83
Primes	140.56	75.46	1.86	97.70	1.44
PrimesPeano	91.34	22.38	4.08	22.50	4.06
Queens	124.75	63.78	1.96	87.15	1.43
QueensUser	190.68	90.94	2.10	122.73	1.55
PermSort	82.61	52.74	1.57	64.20	1.29
PermSortPeano	49.09	20.80	2.36	30.52	1.61
RegExp	34.36	15.93	2.16	21.82	1.57

Fig. 4. Run times (in seconds) and speedups with SWI-Prolog

the non-deterministic specification of `grep` shown in [8]. In all these examples, residuation is not used so that, in principle, our optimization is applicable.

Figures 3 and 4 show the execution times and speedups for these programs with the SICStus-Prolog and SWI-Prolog back end, respectively. The speedups are computed relative to the "full residuation" compilation mode. The timings and speedups show that our proposed improvement is effective, in particular, if `freeze` is used for coroutining, as in SWI-Prolog. This is of practical relevance, since the SWI-Prolog implementation of PAKCS is used in the Debian package "`pakcs`" which is part of recent distributions of the Ubuntu Linux system. Our analysis can detect, for all benchmark programs, that the main expression is non-residuating. The difference in the timings between "no residuation" and "optimized residuation" can be explained by the fact that the run-time system of PAKCS (i.e., the implementation of predefined operations) is not optimized in the optimized residuation mode, since it might also be used by operations requiring residuation.

Another interesting question is the increase of the program size due to the code duplication in the optimized residuation mode. Figures 5 and 6 show the sizes (in bytes) of the executables ("saved state") of all benchmark programs with the SICStus-Prolog and SWI-Prolog back end, respectively. Note that also all standard operations defined in the prelude are duplicated in the optimized residuation mode. Since the run-time system is the largest part of the executable, the difference in program size is not really relevant for such small programs. In order to get some idea of the different sizes for realistic applications, we compiled the Curry package manager [19], a non-trivial Curry application consisting of 116 modules, with different residuation modes. The following table contains the sizes of the executables (in bytes):

Back end	Full Residuation	No Residuation	Optimized Res.
SICStus-Prolog	3,230,549	3,153,585	5,644,240
SWI-Prolog	7,720,682	5,490,839	11,641,481

Program	Full Residuation	No Residuation	Optimized Res.
ReverseUser	612,257	612,126	857,588
Reverse	612,227	612,085	856,735
TakPeano	611,661	611,030	855,907
Tak	608,459	608,586	851,502
ReverseHO	612,979	611,896	859,200
Primes	610,701	610,482	855,398
PrimesPeano	614,630	614,056	864,129
Queens	610,413	609,681	852,238
QueensUser	612,797	612,715	858,428
PermSort	610,339	609,244	853,135
PermSortPeano	613,505	613,778	862,806
RegExp	614,466	613,096	859,765

Fig. 5. Program sizes (in bytes) with SICStus-Prolog

Program	Full Residuation	No Residuation	Optimized Res.
ReverseUser	1,433,607	1,179,270	1,800,747
Reverse	1,432,692	1,178,537	1,799,469
TakPeano	1,431,921	1,178,003	1,798,219
Tak	1,427,842	1,175,038	1,791,529
ReverseHO	1,434,424	1,179,887	1,802,239
Primes	1,431,504	1,177,585	1,797,703
PrimesPeano	1,438,153	1,182,305	1,808,340
Queens	1,429,760	1,176,368	1,794,638
QueensUser	1,434,593	1,179,992	1,802,729
PermSort	1,429,425	1,176,095	1,794,054
PermSortPeano	1,436,714	1,181,373	1,805,870
RegExp	1,437,173	1,181,854	1,806,103

Fig. 6. Program sizes (in bytes) with SWI-Prolog

Although the increase in the program size is considerable, it is not relevant for the practical execution if we take into account the memory sizes of contemporary computer hardware.

7 Related Work

The integration of functions into logic-oriented languages by suspending function calls with free variables has been proposed for various languages, e.g., Escher [23], Le Fun [2], Life [1], NUE-Prolog [26], or Oz [30]. The main motivation for this alternative to narrowing is to evaluate functions in a deterministic manner and to delegate all non-determinism to relations, as in logic programming. Although this principle sounds reasonable at a first glance, there are no strong results about completeness and optimality, as for narrowing [4]. Actually, there are examples where residuation has an infinite derivation space whereas the search space of narrowing is finite [16]. Abandoning residuation completely is also not desirable,

since it is a good principle to connect external operations [10] and to support concurrent computations [31].

The potential incompleteness of residuation is investigated in [16] where a program analysis to approximate the groundness of variables for residuating logic program is proposed. Although this has some similarities with our approach, the analysis is different due to the different underlying languages (e.g., functions in [16] are always strict).

Coroutining is also used in logic programming to delay insufficiently instantiated negated subgoals to avoid logically incorrect answers. This delay might cause "floundering" if only delayed negated subgoals remain. A program analysis to analyze such situations is presented in [25]. Although the overall objective of this work is similar to our work, the underlying operational semantics is quite different to the work presented in this paper.

There are many approaches to implement functional features in logic languages (see [13] and the survey in [18]). Some of them support residuation and use `block`/`freeze` [5] or `when` [26] declarations. As shown by our benchmarks, such declarations have considerable costs which can be reduced by the techniques developed in this paper.

8 Conclusions

We have presented a method to improve the implementation of declarative programs with residuation. Since residuation is implemented in Prolog by coroutining annotations and these annotations have run-time costs even if they are not activated, we developed a compile-time analysis to approximate classes of programs or parts of programs where residuation is not used. For these parts, specific code without residuation annotations is generated. Our benchmarks show that the code optimized in this way can be more than four times faster than the original code if `freeze` is used to implement residuation. This also shows that `freeze` is a costly operation for coroutining (in SWI-Prolog). The use of `block` declarations (in SICStus-Prolog) is less expensive but, even in this case, we could measure a significant speedup by our optimization.

References

1. H. Aït-Kaci. An overview of life. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
3. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

5. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
6. 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.
7. 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.
8. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
9. S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017.
10. S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.
11. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
12. 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.
13. A. Casas, D. Cabeza, and M.V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 146–162. Springer LNCS 3945, 2006.
14. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340, 1998.
15. A. Habel and D. Plump. Term graph narrowing. *Mathematical Structures in Computer Science*, 6(6):649–676, 1996.
16. M. Hanus. On the completeness of residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 192–206. MIT Press, 1992.
17. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
18. 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.
19. 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.
20. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2017.
21. 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.

22. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.
23. J.W. Lloyd. Combining functional and logic programming languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.
24. R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.
25. K. Marriott, H. Søndergaard, and P. Dart. A characterization of non-floundering logic programs. In *Proc. of the 1990 North American Conference on Logic Programming*, pages 661–680. MIT Press, 1990.
26. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
27. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1993.
28. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
29. J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.
30. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
31. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
32. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
33. D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

A Soundness of the Residuation Analysis

In this section we show the soundness of the residuation analysis. For this purpose, we have to define an operational semantics of FlatCurry programs which includes suspended computations. For this purpose, we use the operational semantics presented in [3]. According to this semantics, we consider only *normalized* FlatCurry programs, i.e., programs where the arguments of constructor and function calls are always variables. Any FlatCurry program can be normalized by introducing new variables by let expressions [3]. For instance, the expression "$h(g(x)$" is normalized into "*let* $z = g(x)$ *in* $h(z)$." In the following, we assume that all FlatCurry programs are normalized.

In order to model sharing, which is important for lazy evaluation, variables are interpreted as references into a heap where new let bindings are stored and function calls are updated with their evaluated results. To be more precise, a *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to (normalized FlatCurry) expressions. The *empty heap* is denoted by []. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$. We represent a free variable x in a heap Γ as a circular binding $\Gamma(x) = x$.

Using heap structures, one can provide a high-level description of the operational behavior of FlatCurry programs in natural semantics style. The semantics uses judgements of the form "$\Gamma : e \Downarrow \Delta : v$" with the meaning that in the context of heap Γ the expression e evaluates to a modified heap Δ and a value (head normal form) v. In addition to [3], we also allow that v is the specific constant SUSPEND which indicates a suspended evaluation. Figure 7 shows the rules defining this semantics w.r.t. a given normalized FlatCurry program P ($\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$).

Constructor-rooted expressions (i.e., head normal forms), free variables, or suspended computations are just returned by rule Val. Rule VarExp retrieves a binding for a variable from the heap and evaluates it. In order to avoid the re-evaluation of the same expression, VarExp updates the heap with the computed value, which models sharing. In contrast to the original rules [3], VarExp removes the binding from the heap. On the one hand, this allows the detection of simple loops ("black holes") as in functional programming. On the other hand, it is crucial in combination with non-determinism to avoid the binding of a variable to different values in the same derivation (see [11] for a detailed discussion on this issue). Rule Fun unfolds function calls by evaluating the right-hand side after binding the formal parameters to the actual ones. Let introduces a new binding in the heap and renames the new variable in the expressions with the fresh name introduced in the heap. Similarly, Free introduces a new logic variable in the heap represented by a circular binding. Or non-deterministically evaluates one of its arguments. Rule Select deals with *case* and *fcase* expressions where the discriminating argument evaluates to a constructor-rooted term. In this case Select evaluates the corresponding branch of the (f)*case* expression. If the discriminating argument of an *fcase* expression evaluates to a logic variable, rule Guess non-deterministically binds this variable to some pattern and evaluates the corresponding branch. If the discriminating argument of an *fcase* expression

$$\textsf{Val} \qquad \Gamma : v \;\Downarrow\; \Gamma : v \qquad \text{where } v \text{ is constructor-rooted, } v = \textsc{Suspend} \text{ or a variable with } \Gamma(v) = v$$

$$\textsf{VarExp} \qquad \frac{\Gamma : e \;\Downarrow\; \Delta : v}{\Gamma[x \mapsto e] : x \;\Downarrow\; \Delta[x \mapsto v] : v} \qquad \text{where } x \neq e$$

$$\textsf{Fun} \qquad \frac{\Gamma : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v} \qquad \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

$$\textsf{Let} \qquad \frac{\Gamma[y \mapsto \rho(e)] : \rho(e') \;\Downarrow\; \Delta : v}{\Gamma : \mathit{let}\ x = e\ \mathit{in}\ e' \;\Downarrow\; \Delta : v} \qquad \text{where } y \text{ fresh and } \rho = \{x \mapsto y\}$$

$$\textsf{Free} \qquad \frac{\Gamma[y \mapsto y] : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : \mathit{let}\ x\ \mathit{free}\ \mathit{in}\ e \;\Downarrow\; \Delta : v} \qquad \text{where } y \text{ fresh and } \rho = \{x \mapsto y\}$$

$$\textsf{Or} \qquad \frac{\Gamma : e_i \;\Downarrow\; \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \;\Downarrow\; \Delta : v} \qquad \text{where } i \in \{1, 2\}$$

$$\textsf{Select} \qquad \frac{\Gamma : e \;\Downarrow\; \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : (f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v} \quad \text{where } p_i = c(\overline{x_n}) \text{ and } \rho = \{\overline{x_n \mapsto y_n}\}$$

$$\textsf{Guess} \qquad \frac{\Gamma : e \;\Downarrow\; \Delta : x \qquad \Delta : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$$
where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ fresh variables

$$\textsf{Suspend} \qquad \frac{\Gamma : e \;\Downarrow\; \Delta : x}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Delta : \textsc{Suspend}}$$

$$\textsf{CaseSusp} \qquad \frac{\Gamma : e \;\Downarrow\; \Delta : \textsc{Suspend}}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Delta : \textsc{Suspend}}$$

Fig. 7. Natural semantics of a normalized FlatCurry program P

evaluates to a logic variable, rule Suspend indicates the suspension of this computation by returning the specific constant SUSPEND. Finally, CaseSusp handles a suspended computation in the discriminating argument of a *case* and *fcase* expression.

Our goal is to show that suspended computations cannot occur if the residuation analysis approximates this behavior. A difficulty is the fact that the residuation analysis returns only the positions of arguments which must be ground for non-residuation, whereas we have to deal with the concrete state of variables in the natural semantics. Therefore, we change the abstract domain to contain variables instead of positions. Thus, rule *FDecl* is modified to

$$\mathit{FDecl} \qquad \frac{\mathcal{A}[x_1 \mapsto \{x_1\}, \ldots, x_n \mapsto \{x_n\}] \vdash e : \alpha}{\mathcal{A} \vdash f(x_1, \ldots, x_n) = e : \alpha}$$

Another difference between the residuation analysis and the concrete semantics is the fact that the latter is defined on normalized expressions where the bindings of

the arguments are stored in the heap. To transform such normalized expressions into their standard form without heap information, we dereference normalized expressions w.r.t. a heap Γ by putting all bindings stored in the heap into the expression. This dereference operation is denoted by Γ^* and defined by (for the sake of simplicity, we assume that all pattern and let-bound variables are fresh, otherwise one has to rename them):

$$\Gamma^*(e) = \begin{cases} x & \text{if } e = x \text{ and } \Gamma(x) \text{ undefined} \\ x & \text{if } e = x \text{ and } \Gamma(x) = x \\ \Gamma^*(\Gamma(x)) & \text{if } e = x \text{ and } \Gamma(x) \neq x \\ c(\overline{\Gamma^*(x_n)}) & \text{if } e = c(\overline{x_n}) \\ f(\overline{\Gamma^*(x_n)}) & \text{if } e = f(\overline{x_n}) \\ \mathit{case}\ \Gamma^*(e')\ \mathit{of}\ \{\overline{p_k \to \Gamma^*(e_k)}\} & \text{if } e = \mathit{case}\ e'\ \mathit{of}\ \{\overline{p_k \to e_k}\} \\ \mathit{fcase}\ \Gamma^*(e')\ \mathit{of}\ \{\overline{p_k \to \Gamma^*(e_k)}\} & \text{if } e = \mathit{fcase}\ e'\ \mathit{of}\ \{\overline{p_k \to e_k}\} \\ \Gamma^*(e_1)\ \mathit{or}\ \Gamma^*(e_2) & \text{if } e = e_1\ \mathit{or}\ e_2 \\ \mathit{let}\ y\ \mathit{free}\ \mathit{in}\ \Gamma^*(e') & \text{if } e = \mathit{let}\ y\ \mathit{free}\ \mathit{in}\ e' \\ \mathit{let}\ y = \Gamma^*(e_1)\ \mathit{in}\ \Gamma^*(e_2) & \text{if } e = \mathit{let}\ y = e_1\ \mathit{in}\ e_2 \end{cases}$$

For instance, if $e = h(x)$, $\Gamma(x) = g(y)$, and $\Gamma(y) = 0$, then $\Gamma^*(e) = h(g(0))$.

The soundness of the residuation analysis can be stated as follows:

Theorem 1. *Let $\mathcal{A}$ be correct assumption for a given program, $\Gamma : e \Downarrow \Delta : v$ be a valid judgement of the natural semantics, and z be ground w.r.t. Γ for all $z \in \alpha$ if $\mathcal{A} \vdash \Gamma^*(e) : \alpha$ (i.e., $\bot \neq \alpha \neq \top$). Then $v = c(\overline{y_n})$ and y_i is ground w.r.t. Δ $(i = 1, \ldots, n)$.*

Here, an expression e is called *ground* w.r.t. a heap Γ if, whenever the judgement $\Gamma : e \Downarrow \Delta : v$ holds, $v = c(\overline{y_n})$ for some constructor c and y_i is ground w.r.t. Δ (for $i = 1, \ldots, n$). In other words, if e evaluates to some result, it is a ground constructor term and this evaluation does not suspend. Thus, the theorem also implies that $v \neq$ SUSPEND, i.e., the computation does not suspend.

Since SUSPEND might occur in expressions in the concrete semantics, we add the abstract judgement $\mathcal{A} \vdash$ SUSPEND $: \top$ to express the fact that a suspended computation trivially residuates. To deal with abstraction of free variables occurring in the concrete semantics, we extend rule *Var* with a case for variables not defined in the assumption $\mathcal{A}$:

$$\mathcal{A} \vdash x : \top \qquad \text{if } \mathcal{A}(x) \text{ undefined}$$

Proof (of Theorem 1). Let $\mathcal{A}$ be a correct assumption for a given program, $\Gamma : e \Downarrow \Delta : v$ be a valid judgement, and $\mathcal{A} \vdash \Gamma^*(e) : \alpha$ with $\alpha = \{x_1, \ldots, x_k\}$ and x is ground w.r.t. Γ for all $x \in \alpha$. We have to show that $v = c(\overline{y_n})$ and y_i is ground w.r.t. Δ $(i = 1, \ldots, n)$.

We prove this claim by induction on the height of the proof tree of $\Gamma : e \Downarrow \Delta : v$ and the height of the proof trees to evaluate the arguments of v (if v is not a constant and these proof trees exist).

Base case: Rule Val is applied, i.e., $e = v$ with

$$\Gamma : v \;\Downarrow\; \Gamma : v$$

and v is constructor-rooted, $v = \textsc{Suspend}$, or a variable with $\Gamma(v) = v$. If $v = \textsc{Suspend}$ or v is a variable with $\Gamma(v) = v$, the $\mathcal{A} \vdash \Gamma^*(v) : \top$, which is a contradiction to our assumption. Hence, v is constructor-rooted, i.e., $e = c(\overline{y_n})$ for some constructor c. If $n = 0$, v is a constant and the base case is proven.

Otherwise, we are in the inductive case where we consider the proof tree to evaluate y_i $(i = 1, \ldots, n)$. By rule Cons, $\mathcal{A} \vdash \Gamma^*(y_i) : \alpha_i$ with $\alpha_i \sqsubseteq \alpha$. Hence, by our assumption on α, z is ground w.r.t. Γ for all $z \in \alpha_i$ so that we can apply the induction hypothesis to the evaluation of y_i which shows that y_i is ground w.r.t. Γ.

For the inductive case, we consider the different rules applied to prove the judgement $\Gamma : e \;\Downarrow\; \Delta : v$:

VarExp: Then $e = x$, $\Gamma = \Gamma'[x \mapsto e']$, $\Delta = \Delta'[x \mapsto v]$, and

$$\frac{\Gamma' : e' \;\Downarrow\; \Delta' : v}{\Gamma : x \;\Downarrow\; \Delta : v}$$

By definition of dereferencing, $\Gamma^*(e) = \Gamma^*(x) = \Gamma'^*(e')$ (note that x cannot occur in e', otherwise Γ^* would be undefined). Hence, $\mathcal{A} \vdash \Gamma'^*(e') : \alpha$ so that we can apply the induction hypothesis to $\Gamma' : e' \;\Downarrow\; \Delta' : v$. Thus, the claim holds for this case.

Fun: Then $e = f(\overline{x_n})$. W.l.o.g. we assume that $f(\overline{x_n}) = e'$ is a program rule, i.e., the renaming ρ in rule Fun is the identity (this can always be obtained by renaming program rules). Hence, the inference rule

$$\frac{\Gamma : e' \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v}$$

is applied. Note that the left- and right-hand side of a program rule have the same abstraction: if $\bot \neq \mathcal{A}(f) \neq \top$, then rules *Fun* and *Var* imply

$$\mathcal{A}[x_1 \mapsto \{x_1\}, \ldots, x_n \mapsto \{x_n\}] \vdash f(x_1, \ldots, x_n) : \mathcal{A}(f)$$

and rule *FDecl* yields

$$\mathcal{A}[x_1 \mapsto \{x_1\}, \ldots, x_n \mapsto \{x_n\}] \vdash e' : \mathcal{A}(f)$$

The same holds when the arguments x_i are instantiated. Hence, if $z \in \alpha$ is ground w.r.t. Γ for $\mathcal{A} \vdash \Gamma^*(f(\overline{x_n})) : \alpha$, then $z \in \alpha$ is ground w.r.t. Γ for $\mathcal{A} \vdash \Gamma^*(e') : \alpha$. Thus, we can apply the induction hypothesis to $\Gamma : e' \;\Downarrow\; \Delta : v$ which implies the claim for this case.

Let: Then $e = \mathit{let}\ x = e_1\ \mathit{in}\ e_2$ and rule

$$\frac{\Gamma[x \mapsto e_1] : e_2 \;\Downarrow\; \Delta : v}{\Gamma : \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \;\Downarrow\; \Delta : v}$$

is applied (for the sake of simplicity, we assume that x is already fresh so that we ignore the renaming). By rule *Let*, $\mathcal{A}[x \mapsto \bot] \vdash \Gamma^*(e_1) : \alpha_1$ and $\mathcal{A}[x \mapsto \alpha_1] \vdash \Gamma^*(e_2) : \alpha$. If x is used in expression e_2 (otherwise, we can ignore its binding), $\alpha_1 \sqsubseteq \alpha$. Hence, $\mathcal{A} \vdash \Gamma[x \mapsto e_1]^*(e_2) : \alpha$ so that we can apply the induction hypothesis to $\Gamma[x \mapsto e_1] : e_2 \Downarrow \Delta : v$ which implies the claim for this case.

Free: Then $e = \mathit{let}\ x\ \mathit{free}\ \mathit{in}\ e'$ and rule

$$\frac{\Gamma[x \mapsto x] : e' \Downarrow \Delta : v}{\Gamma : \mathit{let}\ x\ \mathit{free}\ \mathit{in}\ e' \Downarrow \Delta : v}$$

is applied (for the sake of simplicity, we assume that x is already fresh so that we ignore the renaming). Since x is abstracted to $\top$ by rule *Free*, x is not an element of α if $\mathcal{A} \vdash \Gamma^*(e') : \alpha$. Hence, by our assumption on α, all $z \in \alpha$ are ground w.r.t. $\Gamma[x \mapsto x]$. Thus, we can apply the induction hypothesis to $\Gamma[x \mapsto x] : e' \Downarrow \Delta : v$ which implies the claim for this case.

Or: Then $e = e_1\ \mathit{or}\ e_2$ and rule

$$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \Downarrow \Delta : v}$$

is applied for some $i \in \{1, 2\}$. If $\mathcal{A} \vdash \Gamma^*(e_1\ \mathit{or}\ e_2) : \alpha$, then $\mathcal{A} \vdash \Gamma^*(e_i) : \alpha_i$ and $\alpha_i \sqsubseteq \alpha$ by rule *Or*. Hence, if $z \in \alpha_i$, then z is ground w.r.t. Γ by our assumption on α. Thus, we can apply the induction hypothesis to $\Gamma : e_i \Downarrow \Delta : v$ which implies the claim for this case.

Select/Guess/Suspend/CaseSusp: Then $e = (f)\mathit{case}\ e'\ \mathit{of}\ \{\overline{p_k \to e_k}\}$. By our assumption on α, we have

$$\mathcal{A} \vdash \Gamma^*((f)\mathit{case}\ e'\ \mathit{of}\ \{\overline{p_k \to e_k}\}) : \alpha$$

By rule *Case* and *FCase*, $\mathcal{A} \vdash \Gamma^*(e') : \alpha_0$ and $\alpha_0 \sqsubseteq \alpha$. If $z \in \alpha_0$, then z is ground w.r.t. Γ by our assumption on α. Thus, we can apply the induction hypothesis to $\Gamma : e \Downarrow \Delta : w$. Hence, the claim states that $w = c(\overline{y_n})$, i.e., rules Guess, Suspend, and CaseSusp have not been applied here so that only Select is applicable:

$$\frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : e_i \Downarrow \Theta : v}{\Gamma : (f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v}$$

where $p_i = c(\overline{y_n})$ (for the sake of simplicity, we assume that the branch variables are already fresh so that we ignore their renaming). Again by rule *Case* and *FCase*, $\mathcal{A}[y_1 \mapsto \alpha_0, \ldots, y_k \mapsto \alpha_0] \vdash \Gamma^*(e_i) : \alpha_i$ and $\alpha_i \sqsubseteq \alpha$. Thus, we can apply the induction hypothesis to $\Delta : e_i \Downarrow \Theta : v$ which implies the claim.

□