A Partial Evaluator for Curry

Michael Hanus and Björn Peemöller

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

Abstract. We present a partial evaluator for functional logic programs written in Curry. In contrast to previous approaches to the partial evaluation of functional logic programs, we take into account the features used in contemporary Curry programs, in particular, non-deterministic operations and recursive `let` expressions. For this purpose, we base our partial evaluator on FlatCurry, an intermediate language for the representation of Curry programs. We sketch our approach and present initial benchmarks of our implementation.

1 Introduction

Partial evaluation of programs is a technique to anticipate the evaluation of computations once at compile time instead of performing them (possibly several times) at run time. This is possible if some part of the input data, also called *static* data, is known at compile time. In this case, some parts of the program are evaluated so that a *residual* program, i.e., a specialized version of the original one, is returned. Since some computations have been performed at compile time, the run time of the specialized program could be considerably decreased. The static data does not need to be some user input, but can also be subexpressions in the original program. *Offline* partial evaluators obtain information about static data from a separate static analysis phase (binding-time analysis), whereas *online* partial evaluators obtain this information on the fly and propagate it during the partial evaluation process. In this work we follow the online partial evaluation approach.

Partial evaluation has already been studied for different kinds of programming languages, like functional languages, logic languages, as well as for combined functional logic languages. An interesting aspect of the partial evaluation of functional logic programs is the fact that the effects of supercompilation [23] can be obtained by applying the operational semantics of the source language (narrowing) at partial evaluation time [5]: if a function is called with unknown arguments, narrowing instantiates these arguments such that the rules defining this function can be applied. Hence, one mainly needs to control the partial evaluator, i.e., avoiding infinite unfoldings and instantiations of logic variables, in order to obtain residual programs.

Thanks to this insight, partial evaluators for functional logic languages can be constructed with techniques similarly to the implementation of these languages. For instance, Albert et al. [3] proposed a partial evaluator for Curry [17] based on the intermediate language FlatCurry. Since FlatCurry makes the evaluation strategy of Curry programs explicit [1], the use of FlatCurry led to a partial evaluator able to optimize practical Curry programs. Unfortunately, when this partial evaluator was constructed, the use of non-deterministic operations, although proposed some years ago [14], was not well established. Therefore, the partial evaluation scheme was based on term rewriting and restricted to confluent programs, i.e., all operations were required to be deterministic, and recursive `let` expressions were also not taken into account. Thus, if this partial evaluator is applied to programs containing non-deterministic operations, which is a useful programming pattern in contemporary functional logic programs [6,8], the resulting programs are not semantically equivalent to the source programs.

In order to deal with realistic Curry programs, it is crucial for a partial evaluator to cover the full source language, including both logic features such as non-determinism and functional features such as recursive `let` expressions. Therefore, we extend in this work the partial evaluator of Albert et al. [3] to cover the full language of FlatCurry. In contrast to [3], we base our partial evaluator on an operational semantics [1] which is adequate for contemporary Curry programs.

We start with an introduction to the functional logic language Curry in Sect. 2 before we sketch the structure of the partial evaluator in Sect. 3. The partial evaluation scheme is presented in Sect. 4, whereas control issues are discussed in Sect. 5. We evaluate our implementation with some benchmarks in Sect. 6 before we conclude in Sect. 7.

2 Curry

We briefly review the basic concepts of the functional logic language Curry. More details can be found in recent surveys on functional logic programming [7,15] and in the language report [17].

The syntax of Curry [17] is close to Haskell [21], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of an operation f to an expression e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. If the data type of Booleans and a negation operation are defined by

```
data Bool = False | True
not True  = False
not False = True
```

the expression "`not x where x free`" non-deterministically reduces to `False` with the binding `x = True`, and to `True` with the binding `x = False`. A further kind of non-determinism is supported in Curry by the *choice* operator "`?`", which can be considered as predefined by the overlapping rules

```
x ? _ = x
_ ? y = y
```

Thus, we can define a *non-deterministic operation* `coin` yielding the values `0` and `1` by

```
coin = 0 ? 1
```

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operation

```
double x = x + x
```

and the expression "`double coin`". If we evaluated this expression by term rewriting, we could have the reduction

```
double coin  →  coin + coin  →  0 + coin  →  0 + 1  →  1
```

leading to the unintended result `1`. Note that this result cannot be obtained with a strict reduction strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, Curry is based on the rewriting logic CRWL, proposed by González-Moreno et al. [14] as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [18] where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `coin` are shared so that all of them consistently reduce to either `0` or `1`.

3 Overview of the Partial Evaluator

Before describing the details of the partial evaluation process, we provide an overview of the partial evaluator and its usage. Since our partial evaluator is an extension of the first partial evaluator for Curry described in [3], our representation is oriented towards the original description.

Our partial evaluator is intended to specialize some parts of a given input program in order to create an optimized, residual program. In order to support the specification of expressions to be optimized, we assume that these expressions are annotated with `PEVAL`. For example, we assume a program which contains the function definition

```
main xs = map (twice square) xs
```

We can then annotate the main expression (or parts of it) as follows:

```
main xs = PEVAL (map (twice square) xs)
```

Actually, `PEVAL` is the identity function, i.e., it has the type `a → a`. As a consequence, annotations with `PEVAL` do not change the semantics of the original program. After annotating the program, the process of partial evaluation is fully automatic. The process itself consists of the following phases (depicted in Fig. 1):

1. The partial evaluator is called for a given program, containing annotated expressions as described above. This source program is converted into the standard intermediate representation for Curry programs, called FlatCurry (see Sect. 4.1).
2. The process continues by extracting the set of annotated expressions and creating a copy of the original program without annotations.
3. Both form the input for the partial evaluation phase, which is later described.
4. The output of the partial evaluation is a set of semantically equivalent, potentially more efficient expressions. These expressions are converted to new function definitions to allow reuse, a process called *renaming*.
5. The evaluation process tends to produce some "intermediate" functions which only pass their parameters to another function. Therefore, the process is finished by a *compression* phase which removes such intermediate functions by inlining and simplifies expressions to produce a more efficient and legible result.
6. Finally, the annotated expressions of the form `(PEVAL` e`)` are replaced with their (hopefully more efficient) equivalents e', where e' is the renaming of e. This optimized program is then stored as a FlatCurry program.

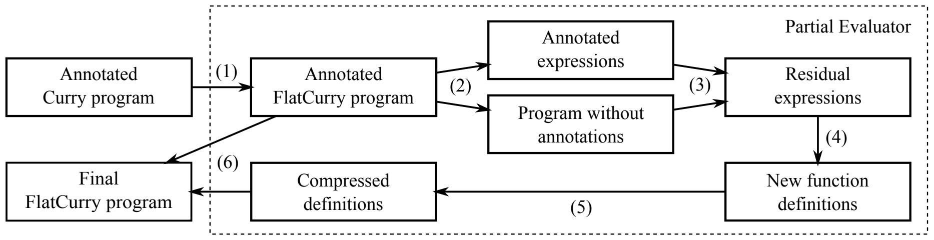

Fig. 1. Overview of the partial evaluation process with phases (1) to (6)

For instance, with the usual definitions of `map`, `twice`, and `square`, the example above is transformed into

```
main xs = map0 xs
map0 xs = case xs of []    → []
                     y:ys  → let z = (y*y) in (z*z) : map0 ys
```

so that the overhead of the higher-order operations `map` and `twice` is eliminated.

The fact that the partial evaluator internally operates on the FlatCurry format is no restriction, since this format is used by current Curry compilers anyway, e.g., PAKCS [16] or KiCS2 [11]. Hence, the partial evaluator can easily be incorporated into a compilation chain.

4 The Partial Evaluation Scheme

As already mentioned, the partial evaluator described in [3] lacks support for two language features, namely non-deterministic operations and `let` expressions. For instance, consider the definition

```
main = PEVAL (double coin)
```

w.r.t. the definitions of `coin` and `double` shown in Sect. 2. The partial evaluator [3] unfolds the call to `double` in the body of `main` to `(0 ? 1) + (0 ? 1)`, so that the residual program yields the values `0`, `1`, `1`, and `2` for `main`. However, according to the call-time choice semantics of Curry [14,18], the correct result would be `0` or `2` but not `1`. This problem arises from the residual semantics of the original partial evaluator, which is based on term-rewriting so that non-determinism in shared subexpressions is duplicated in the residual programs.

The second missing feature are (mutually recursive) `let` expressions, i.e., bindings where the variables to be bound might occur in the right-hand side of the bindings. For example, it is not possible to partially evaluate the program

```
ones = let ones = 1 : ones in ones
main = PEVAL (take 2 ones)
```

One might encounter that this does not impose a real restriction because recursive let-bindings could be interpreted by recursive function definitions (at the cost of some overhead). While this is possible for the example above, it is not whenever a non-deterministic value should be shared. For instance, consider the following program:

```
digits = let digits = (0 ? 1) : digits in digits
main   = PEVAL (take 2 digits)
```

$$
\begin{array}{rcll}
P & ::= & \overline{D_m} & \text{(program)}\\
D & ::= & f(\overline{x_n}) \texttt{ = } e & \text{(defined function)}\\
e & ::= & x & \text{(variable)}\\
 & | & c(\overline{e_k}) & \text{(constructor call)}\\
 & | & f(\overline{e_k}) & \text{(function call)}\\
 & | & \texttt{let \{ } \overline{x_n \texttt{ = } e_n} \texttt{ \} in } e & \text{(recursive let binding)}\\
 & | & \texttt{let } \overline{x_n} \texttt{ free in } e & \text{(free variables)}\\
 & | & e_1 \texttt{ ? } e_2 & \text{(disjunction)}\\
 & | & \texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}} & \text{(case expression, } p_i \text{ pairwise different)}\\
p & ::= & c(\overline{x_n}) & \text{(constructor pattern)}
\end{array}
$$

Fig. 2. The FlatCurry representation of programs

Because of the `let` binding, the decision to bind `digit` to either `0` or `1` is shared, and, in consequence, the expression `main` evaluates to either `[0,0]` or `[1,1]`. If we replaced the definition of `digits` by a top-level operation, as in

```
digits = (0 ? 1) : digits
main   = PEVAL (take 2 digits)
```

the expression `main` would produce the additional results `[0,1]` and `[1,0]`. Thus, recursive `let` expressions cannot be transformed into operations but must be explicitly considered by a partial evaluator.

The usage of both features in contemporary Curry programs is the motivation for us to develop a new partial evaluator. In contrast to [3], we do not use a semantics based on term rewriting. Instead, we base our work on the natural semantics for FlatCurry proposed in [1] which is intended to specify the call-time choice semantics of non-deterministic operations by modeling a heap structure to express sharing. A similar semantics has been used in [13] in a partial evaluator for first-order functional programs. In contrast to our approach, non-determinism, which is essential for Curry, has not been considered there.

4.1 FlatCurry

FlatCurry is a simple intermediate language used by Curry compilers [11,16]. Moreover, it is also the basis of precise descriptions of the semantics of Curry [3] and semantics-based tools for Curry (e.g., [2,3,4]). The syntax of this representation is depicted in Fig. 2, where we denote a sequence of objects $o_1, \dots, o_n$ by $\overline{o_n}$. A FlatCurry program P consists of a sequence of function definitions D such that each function must be defined by a single rule with a linear left-hand side, i.e., the variables $\overline{x_n}$ must be pairwise different. The right-hand side of a function definition is an expression e composed of variables (x, y, z, ...), constructors (A, B, C, ...), and function calls (f, g, h, ...). In the following, we denote by ϕ a constructor c or a function f. For the sake of simplicity, we assume that literals occurring in the source program, like numbers or characters, are represented as nullary constructors. Additionally, we allow local (mutually recursive) bindings of variables, the introduction of free (logic) variables, disjunctions (to represent overlapping left-hand sides in the source language), and pattern matching. The patterns p_i in `case` expressions are required to be pairwise different and only consist of constructors applied to variables. In consequence, nested patterns in the source language are represented by nested `case` expressions. For example, the list concatenation `conc` is represented in FlatCurry as

```
conc(xs,ys) = case xs of { []    →  ys
                         ; z:zs  →  z : conc(zs,ys) }
```

Note that, in contrast to [1], we do not distinguish between *flexible* and *rigid* `case` expressions. Although they behave differently on free (logic) variables [17], this difference is not relevant for partial evaluation [3]. Furthermore, we omit the representation of external functions like arithmetics, which are implemented in the partial evaluator but do not play a significant role in the evaluation scheme. Finally, we do not consider higher-order applications in the syntax of FlatCurry since they can be represented by an operation $apply$ where partial applications are interpreted as constructor calls [3].

4.2 Natural Semantics

We base our partial evaluator on a variant of the operational semantics of FlatCurry [1], also referred to as the *natural semantics* of FlatCurry. The semantics uses a heap structure to specify sharing of expressions and computes the *(flat) value* of an expression which is either a logic variable (w.r.t. the associated heap) or a constructor applied to variables.

$$Heap = \mathcal{V} \to \{\text{free}, \blacksquare\} \uplus Exp \qquad\qquad Value \ ::= \ x \mid c(\overline{x_n})$$

A *heap* is a partial mapping from a set of variables $\mathcal{V}$ to either an expression (Exp is the set of expressions according to the syntax of FlatCurry), a special symbol "free" to represent a free variable,[1] or a symbol "$\blacksquare$" representing a black hole.[2] We denote the empty heap by $[]$, and the value associated to a variable x in a heap Γ by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for all $y \neq x$.

We use judgements of the form $\Gamma : e \Downarrow \Delta : v$ which express the fact that "the expression e under the heap Γ evaluates to the value v and the (possibly modified) heap Δ". The basic inference rules of the natural semantics are depicted in Fig. 3. We briefly describe these rules and explain the differences to the original version of [1].

(Value) Evaluation of a value directly returns the value without modifying the heap.

(VarExp) This rule implements sharing of subexpressions. If a variable to be evaluated is bound to an expression, the expression is evaluated and its value is returned. In addition, the heap is updated with the value. During evaluation of the expression, the binding is replaced by $\blacksquare$, in contrast to [1]. This allows the detection of black holes and is necessary for the correctness of the semantics [10] (see also Appendix A for a detailed explanation).

(Flatten) To correctly implement sharing, arguments of function or constructor calls must be represented in the heap. This is usually achieved by a preprocessing step called *flattening* or *normalization* [1,19], but it can also be performed on demand.

[1] [1] represents free variables by circular `let` bindings of the form `let {`x `=` x`} in` e, but this prohibits the correct representation of such bindings occurring in the source code.

[2] We use a special symbol for black holes instead of simply removing the binding for a variable (as in [19]) in order to distinguish black holes from unbound variables.

(Value) $\Gamma : v \Downarrow \Gamma : v \quad$ where $v = c(\overline{x_n})$ or $v \in \mathcal{V}$ with $\Gamma[v] = \text{free}$

(VarExp) $$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

(Flatten) $$\frac{\Gamma[y \mapsto e_i] : \phi(x_1, \ldots, x_{i-1}, y, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v}{\Gamma : \phi(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v} \quad \text{where } e_i \notin \mathcal{V}, y \text{ fresh}$$

(Fun) $$\frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \quad \text{where } f(\overline{x_n}) = e \in P,\ \sigma = \{\overline{x_n \mapsto y_n}\}$$

(Let) $$\frac{\Gamma[\overline{y_k \mapsto \sigma(e_k)}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let}\ \{\ \overline{x_k = e_k}\ \}\ \texttt{in}\ e \Downarrow \Delta : v} \quad \text{where } \sigma = \{\overline{x_k \mapsto y_k}\}, \overline{y_k} \text{ fresh}$$

(Or) $$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ ?\ e_2 \Downarrow \Delta : v} \quad \text{where } i \in \{1, 2\}$$

(Free) $$\frac{\Gamma[\overline{y_n \mapsto \text{free}}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let}\ \overline{x_n}\ \texttt{free in}\ e \Downarrow \Delta : v} \quad \text{where } \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

(Select) $$\frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v} \quad \text{where } \begin{array}{l} p_i = c(\overline{x_n}), \\ \sigma = \{\overline{x_n \mapsto y_n}\} \end{array}$$

(Guess) $$\frac{\Gamma : e \Downarrow \Delta[x \mapsto \text{free}] : x \qquad \Delta[x \mapsto \sigma(p_i), \overline{y_n \mapsto \text{free}}] : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v}$$
$$\text{where } i \in \{1, \ldots, k\}, p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

Fig. 3. Natural semantics

(Fun) This rule unfolds a function call, where the result is obtained by evaluation of the function's right-hand side. We assume that the program P is a global parameter of the calculus. Generally, whenever new variables are introduced by the program, we apply a renaming substitution σ to prohibit name clashes.

(Let) The bindings of a `let` construct are added to the heap after all variables have been renamed to fresh variable names.

(Or) This rule non-deterministically chooses one of the arguments to be futher evaluated. In consequence, this rule introduces non-determinism into the calculus itself.

(Free) Like variables bound to expressions, logic variables are renamed and afterwards bound in the heap.

(Select) For `case` expressions whose argument evaluates to a constructor-rooted term the right-hand side of the corresponding alternative is selected and evaluated.

(Guess) For `case` expressions whose argument evaluates to a logic variable, one of the alternatives is non-deterministically chosen to be evaluated. The variable is then bound to the corresponding pattern where the variables inside the pattern are also bound as logic variables.

4.3 Ensuring Termination

Following the general idea of partial evaluation of functional logic programs [5] as well as logic programs [20], we evaluate an annotated expression e with a (possibly incomplete) standard derivation $[\,] : e \Downarrow \Delta : e'$. In order to ensure the termination of the partial evaluation process, we defer the evaluation of some expressions. For example, consider the program

```
loop xs = loop xs
main xs = PEVAL (loop xs)
```

The evaluation of the expression `main` does not terminate due to the recursive function call to `loop`. To achieve termination of the partial evaluation process, we modify the natural semantics as follows:

1. The evaluation of an expression can be deferred to avoid non-termination.
2. An operation *proceed* is used to decide whether a function call should be unfolded or deferred.

This *residualizing natural semantics* is similar to [3,4] but more complex due to the use of a heap for sharing instead of term rewriting. Regarding the first modification, we extend the representation of values with a new symbol $\langle\!\langle\cdot\rangle\!\rangle$ which encloses expressions whose evaluation should be deferred.

$$Value \ ::= \ \ldots \mid \langle\!\langle e \rangle\!\rangle \quad (\text{annotated expression})$$

This annotation directly corresponds to the `PEVAL` annotation in source programs. Second, we extend the inference system with the operation *proceed*, deciding whether a function call should be unfolded, and replace the rule Fun with:[3]

$$(\text{FunEval}) \quad \frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \quad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ \mathit{proceed}(\Gamma, f(\overline{y_n})) = \text{true} \end{array}$$

$$(\text{FunDefer}) \quad \Gamma : f(\overline{y_n}) \Downarrow \Gamma : \langle\!\langle f(\overline{y_n}) \rangle\!\rangle \quad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ \mathit{proceed}(\Gamma, f(\overline{y_n})) = \text{false} \end{array}$$

Approaches for the concrete definition of *proceed* will be discussed in Sect. 5.1. Furthermore, we extend the rule Value to also return deferred expressions unchanged and constrain the rule VarExp in that the value must not be a deferred expression. Finally, we add two more rules for deferred expressions where the annotation is lifted upwards:

$$(\text{VarDefer}) \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto e'] : \langle\!\langle x \rangle\!\rangle} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

$$(\text{CaseDefer}) \quad \frac{\Gamma : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \langle\!\langle \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

4.4 Dealing with Partial Information

In contrast to the evaluation performed in a standard interpreter, the partial evaluation process has to deal with partial knowledge in the form of unbound variables. For instance, if the right-hand side of a function declaration like `f x = PEVAL (g x)` should be evaluated, there is no binding information for the parameter variable `x`.

[3] Actually, the operation *proceed* also takes into account the context of reductions already performed, but we omit them here for the sake of simplicity.

A possible solution is to handle such unbound variables as logic variables, as done in [5], so that they are bound to appropriate values by the partial evaluator. Since it has been shown in [2] that the back-propagation of these bindings can lead to incorrect residual programs, [3] uses a residualizing semantics which represents such bindings by `case` expressions in the residual program. However, this is only necessary for unbound variables. Explicitly introduced logic variables are known to be free during the actual evaluation so that they can be bound during partial evaluation time. For instance, consider the expression

```
let x free in case x of { True  →  1 }
```

Here we can bind `x` to `True`, since this binding is not visible outside the scope of this expression, select the (single) branch as the value of the `case` expression, and continue by evaluating its right-hand side. In consequence, our implementation evaluates this expression to `1`, while the partial evaluator described in [3] cannot evaluate the expression any further.

Hence, we distinguish unbound variables from logic variables by *not binding* them in the heap. Furthermore, we assume that rule Value is also applicable to variables not bound in the heap so that unknown variables reduce to themselves. Thus, only the rules for `case` expressions have to be changed, where it is now also possible that the scrutinized value is an unknown variable. Following the idea of [3], we generate *residual case expressions* to defer the inspection of the variable to the run time of the specialized program. Therefore, we extend the definition of values to

$$Value \;::=\; \ldots \mid \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to v_k}\ \} \qquad \text{(residual case expression)}$$

where the variable x inspected in the `case` expression is not bound in the corresponding heap. Because `case` expressions are now contained in the set of values, we also have to consider them as the value of a variable or an expression examined by another `case` expression. Hence, we add the following rules:

$$\text{(CaseUnbound)}\quad \frac{\Gamma : e \Downarrow \Delta : x}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}} \quad \text{where } x \notin dom(\Delta)$$

$$\text{(CaseCase)}\quad \frac{\Gamma : e \Downarrow \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p'_j \to \langle\!\langle e'_j \rangle\!\rangle}\ \}}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p'_j \to \langle\!\langle \begin{array}{c}\mathtt{case}\ e'_j\ \mathtt{of}\\ \{\ \overline{p_k \to e_k}\ \}\end{array} \rangle\!\rangle}\ \}}$$

$$\text{(VarCase)}\quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}] : x} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

$$\text{(CaseVarCase)}\quad \frac{\Gamma : e \Downarrow \Delta : x}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p'_j \to \langle\!\langle \begin{array}{c}\mathtt{case}\ x\ \mathtt{of}\\ \{\ \overline{p_k \to e_k}\ \}\end{array} \rangle\!\rangle}\ \}} \quad \text{where } \Delta[x] = \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p'_j \to e'_j}\ \}$$

The general idea is to lift `case` expressions inspecting an unbound variable upwards and to defer the evaluation of the alternatives. Such deferred expressions are not further evaluated in the residual semantics but later extracted by the global iterative process as the initial expressions of a new specialization run. Because the alternatives are then evaluated independently, it will be possible to take the binding information of the `case` expression into account. For instance, if we consider the expression

```
case x of { True  →  not x }
```

a subsequent evaluation of the right-hand side `not x` may respect the binding of `x` to `True` and, thus, directly evaluate to `False`.

4.5 Dereferencing the Heap

After evaluating an expression to a residual value, this value might contain variables which are either free or bound to expressions in the corresponding heap. To be able to replace parts of the input program with residual values, these bindings have to be added to the values to form valid expressions, a process we call *dereferencing the heap*. Conceptually, for a given configuration $\Gamma : e$, we retrieve the set of variables transitively reachable from e and bound in Γ and add the corresponding bindings to the expression. For residual `case` expressions, we also respect the bindings represented by the `case` expression. The bindings are divided into logic variables (*fv*) and variables bound to expressions (bv) and added to the original expression:

$$drf(\Gamma, e) = \begin{cases} \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to drf(\Gamma[x \mapsto p_k], e_k)}\ \} & \text{if } e = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \\ \langle\!\langle \texttt{let}\ fv(\Gamma, e)\ \texttt{free in let}\ bv(\Gamma, e)\ \texttt{in}\ e \rangle\!\rangle & \text{otherwise} \end{cases}$$

For instance, if we consider the configuration

$$[\mathrm{y} \mapsto \mathrm{free}] : \texttt{case x of \{True -> x; False -> y\}}$$

then dereferencing will produce the expression

$$\texttt{case x of \{ True} \to \langle\!\langle \texttt{let \{ x = True \} in x} \rangle\!\rangle\texttt{; False} \to \langle\!\langle \texttt{let y free in y} \rangle\!\rangle\ \texttt{\}}$$

5 Control

Our partial evaluation algorithm follows the general procedure of Alpuente et. al. [5], which is parametric w.r.t. an *unfolding rule* used to construct a finite derivation for an expression and an *abstraction operator* used to guarantee that only finitely many expressions are evaluated. The basic algorithm is depicted in Fig. 4 and works as follows. Given an input program P and a set of annotated expressions E, the algorithm starts by applying an unfolding rule which evaluates each expression according to the residual semantics presented in the previous section and extracts the results by drf. If there is more than one derivation in the residual semantics due to the non-deterministic inference rules Or and Guess, the different extracted results of the derivation are combined by the choice operator "`?`". If there is no derivation at all, the result is represented by the predefined operation `failed`. In the next step, an abstraction operator is applied to this set, adding the new expressions to the set of already evaluated expressions. This phase yields a new set which may need further evaluation, hence, this process is iteratively repeated until no more expressions are added to the set. This iteration is necessary for the correctness of partial deduction [20] in order to achieve a "closed" set of expressions

```
**Input**: A program $P$ and a set of expressions $E$
**Output**: A set of expressions $S$
$i := 0;\ E_0 := E;$
**repeat**
    $E' := unfold(E_i, P);$
    $E_{i+1} := abstract(E_i, E');$
    $i := i + 1;$
**until** $E_i = E_{i+1}$ (modulo renaming);
**return** $S := E_i$
```

Fig. 4. Basic algorithm for partial evaluation

that covers all expressions possibly occurring in the residual program. To generate the resulting program, the same unfolding rule has to be applied to the resulting set of expressions to generate the corresponding resultants, i.e., the rules of the residual program (in our implementation, this step is integrated into the algorithm). Finally, the set of generated resultants are compressed to eliminate intermediate and redundant functions (see [5] for details).

This procedure distinguishes two levels of control, namely the *local level*, managed by the unfolding rule to avoid infinite evaluations, and the *global level*, managed by the abstraction operator to avoid infinitely repetitions of the partial evaluation algorithm. To ensure termination of the whole process, both *local* and *global* termination is required.

5.1 Local Control

Termination of the unfolding rule directly corresponds to termination of the residual semantics presented in Sect. 4. For this purpose, the semantics has already been extended by an oracle $proceed(\Gamma, e)$ responsible for the decision whether a function call should be unfolded or not. There exist several well-known techniques in the literature to come to this decision, e. g., depth-bounds, loop-checks [9], well-founded orderings [12], or well-quasi orderings [22]. Our implementation currently supports the following simple strategies:

None No unfolding is performed for user-defined functions.
One Only one function call is unfolded for each evaluation.
Each At most one call is unfolded for each user-defined function, subsequent calls are deferred.
All All function calls are unfolded, which corresponds to the original inference system. This does not guarantee termination but may be useful if the user is sure that the process terminates.

Note that, regardless of the chosen strategy, built-in functions (such as arithmetics) are evaluated in any case, since they are known to terminate.

Expressions that have been deferred during evaluation will be extracted and eventually added to the set of expressions to be evaluated, depending on the operation $abstract$ (see Sect. 5.2 for details). Generally, a strategy that allows more evaluation steps in one derivation than another strategy might seem superior. If an evaluation is split into multiple derivations with deferred subexpressions, each of these subexpressions has to be evaluated anew and leads to a new residual function to be generated. In contrast, longer derivations will produce less deferred subexpressions and, hence, less residual functions. Nevertheless, although a simpler strategy may produce more intermediate expressions, there are better chances that some of these expressions have already been encountered before, reducing the overall number of expressions to be evaluated. Furthermore, the final compression phase will eliminate intermediate functions so that even the simple strategies perform very well in practice.

5.2 Global Control

The local control is parametric w.r.t. the decision whether to stop or to proceed with the evaluation, since it is safe to terminate the evaluation at any point. This flexibility does not apply to the global control because we cannot stop the iterative extension of the set of expressions until all function calls in this set are "closed" w.r.t. the set of expressions. An expression e is closed w.r.t. a set of expressions if it is an instance of an expression in the set and all expressions in the matching substitution are recursively closed (see [5] for details). This condition is necessary to ensure the correctness of the partial evaluator so that the specialized program computes the same solutions as the original program. In order to avoid the construction of infinite sets of expressions, expressions in this set are generalized to ensure termination of this process.

Hence, the operation $abstract$ returns a safe approximation of $E_i \cup E'$ so that each expression in the set of $E_i \cup E'$ is closed w.r.t. the result of $abstract(E_i, E')$. More precisely, an expression $e' \in E'$ is added to the set E_i according to the following rules (note that the result of unfolding is either a variable, a deferred expression, a constructor appplication, a `case` expression, or a choice of these results):

1. If e' is a variable, it is discarded.
2. If e' has the form $\langle\!\langle e \rangle\!\rangle$, one of the following options is considered:
   (a) add e to the set E_i,
   (b) discard the expression e, or
   (c) compute the most specific generalization of e and some expression $e' \in E'$, say $\hat{e}$, and try to add both $\hat{e}$ and the expressions in the corresponding substitutions σ and θ, where $e = \sigma(\hat{e})$ and $e' = \theta(\hat{e})$.
3. For all other cases (constructor calls, `case` expressions, choices), the corresponding subexpressions are considered.

Like for the unfolding rule, the abstraction can be parameterized by a criterion to decide the option taken in (2). Our implementation currently supports abstractions using a well-founded ordering or an embedding ordering to distinguish between (2a) and (2c), i.e., smaller expressions are added but larger expressions are generalized.

To achieve a good level of specialization, it is crucial to recognize different variants of one expression as equivalent in order to discard them in (2b). This is more complex in our framework compared to [3], since we take `let` expressions into account. For example, consider the equivalent expressions "`map(square,xs)`" and "`let {f = square} in map(f,xs)`". If we do not recognize them as variants, they might be generalized to `map(f,xs)` which could not further be specialized. Therefore, we *normalize* expressions by applying α-conversion and flattening [19] before computing their abstractions.

Benchmark	Time for PE	Original	Specialized	Speedup
`allOnes`	280	180	140	1.29
`doubleApp`	330	190	160	1.19
`doubleFlip`	330	230	210	1.10
`lengthApp`	320	120	90	1.33
`kmp`	6100	1000	50	20.00
`foldr (+) 0 xs` (sum)	300	600	400	1.50
`foldr (+) 0 (map square xs)`	340	1280	800	1.60
`foldr (++) [] xs` (concat)	400	440	220	2.00
`map (twice square) xs`	420	1600	1410	1.13
`foldr (?) failed ys` (choose)	330	50	40	1.25
`head (perm ys)`	410	2960	40	74.00

Table 1. Benchmarks of selected partial evaluation examples (in msec)

6 Experimental Results

In this section we evaluate the implementation of our partial evaluator by some benchmarks. We compile both the partial evaluator and the benchmarks with the PAKCS Curry compiler (version 1.11.3, based on SICStus Prolog 4.2.3). All benchmarks were executed on a Linux machine (Debian Wheezy) with an Intel Core i5-750 (2.66GHz) processor and 4GiB of memory. The timings were performed using the profiling operation `profileTimeNF` of PAKCS and denote the time required for computing the normal form of the respective result in milliseconds (the arguments passed to the various functions were evaluated before to bring out the speedup obtained by partial evaluation). The benchmark examples have been specialized with one unfolding per evaluation and without any abstraction, since all examples terminated. Experiments with both a well-founded ordering or a well-quasi ordering resulted in the same or worse performance. Table 1 presents the time required for the partial evaluation process itself, for executing the original and the specialized program, and the gained speedup.

In the first group of benchmarks, we consider some typical examples of partial deduction and functional program transformations. These are simple functions working on lists or trees as (intermediate) data structures: `allOnes` computes the length of its input list, represented as Peano numbers, and constructs a new list of the same length with `1` as all elements, `doubleApp` is the concatenation of three lists, `doubleFlip` flips a tree structure twice, returning the same tree, `lengthApp` computes the length of the concatenation of two lists, and `kmp` implements a generic string pattern matcher. The first four functions were specialized without static input data, while the `kmp` example was specialized w.r.t. a fixed pattern of length 4, explaining both the time needed for partial evaluation and the gained speedup.

In the second group, we benchmark some examples with higher-order functions: the computation of the sum of list elements using `foldr`, the sum of squared numbers, the concatenation of a list of lists, and repeatedly applying a function to a list. All functions are applied to an input list `xs` containing 200,000 elements. The speedup is generally achieved because of the removal of intermediate data structures. For instance, the Curry expression “`foldr (+) 0 (map square xs)`” is specialized to the following residual FlatCurry definition:

```
sumSquare(xs) = case xs of { []    →  0
                           ; y:ys  →  (y*y) + sumSquare(ys) }
```

Finally, we evaluate two (complicated) variants of the function `choose`, which non-deterministically chooses one element of a given list `ys` containing 10,000 elements:

```
choose (x:xs) = x ? choose xs
```

Our partial evaluator computes this simple implementation of `choose` for the first example. The result for the second example only differs from `choose` in the order in which the two non-deterministic alternatives are taken, which stems from the implementation of `perm`. The huge speedup is achieved because of the omission of the non-deterministic intermediate list structure.

To summarize, our partial evaluator shows promising results and is capable of performing optimizations such as deforestation [24] and transformation of higher-order functions to first-order ones. In addition, non-deterministic operations are correctly specialized in contrast to [3], and the results for deterministic operations are almost identical.

7 Conclusions and Future Work

We have presented a new partial evaluation scheme for the functional logic language Curry based on its intermediate representation FlatCurry. The partial evaluator is based on an adaptation of the natural semantics of FlatCurry, extending the semantics to deal with the requirements of partial evaluation such as ensuring termination. In contrast to the original partial evaluator [3], which is based on term rewriting without sharing, the new implementation correctly handles both recursive `let` expressions and non-deterministic operations and, thus, supports full (Flat)Curry. As our benchmarks demonstrate, the implementation is capable of powerful optimizations both to deterministic and non-deterministic programs.

For future work, we intend to formally prove the correctness of the partial evaluation scheme, which should be manageable due to the similarity of the original and residual semantics. Another aspect for further investigations is the improvement of the abstraction operator. While the abstraction is necessary to ensure termination, a too general abstraction reduces the quality of the specialization. Thus, more sophisticated abstraction operators might be beneficial.

References

1. 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.
2. E. Albert, M. Hanus, and G. Vidal. Using an abstract representation to specialize functional logic programs. In *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, pages 381–398. Springer LNCS 1955, 2000.
3. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.

4. E. Albert, M. Hanus, and G. Vidal. A residualizing semantics for the partial evaluation of functional logic programs. *Information Processing Letters*, 85(1):19–25, 2003.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
6. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
7. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
8. S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.
9. R.N. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
10. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
11. 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.
12. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding. *New Generation Computing*, 11(1):47–79, 1992.
13. S. Fischer, J. Silva, S. Tamarit, and G. Vidal. Preserving sharing in the partial evaluation of lazy functional programs. In A. King, editor, *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'07)*, pages 74–89. Springer LNCS 4915, 2008.
14. 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.
15. 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.
16. 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/`, 2013.
17. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
18. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
19. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
20. J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
21. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
22. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. of the 1995 International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.
23. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), 1986.
24. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Black Hole Detection

As mentioned in Sect. 4.2, rule VarExp of the natural semantics shown in Fig. 3 replaces the variable binding $x \mapsto e$ by $x \mapsto \blacksquare$ in the heap when evaluating the associated expression e. This allows the detection of black holes (a self-dependent infinite loop) [19], as done in some implementations of functional (logic) languages. For instance, an attempt to evaluate the expression "`let {x = x} in x`" would result in a finite but incomplete derivation tree, whereas it would trigger the construction of an infinite derivation tree if the binding $x \mapsto e$ was kept.

The detection of black holes included in the semantics seems to be an optimization that could be omitted for deterministic programs [19]. However, it is crucial in combination with non-determinism in order to prevent the binding of a variable to *different* values in the *same* derivation, as shown in [10]. For example, consider the expression "`let { x = T ? case x of { T → F }} in x`". If we do not replace the variable binding in rule VarExp, the following derivation would be possible:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma : \mathtt{T} \Downarrow \Gamma : \mathtt{T}}{\Gamma : \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}} \Downarrow \Gamma : \mathtt{T}}}{\Gamma : \mathtt{x} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}} \qquad [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}{\Gamma : \mathtt{case\ x\ of\ \{\ T \to F\ \}} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}}{\Gamma : \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}}{\Gamma : \mathtt{x} \Downarrow [\mathtt{x} \mapsto \mathtt{F}] : \mathtt{F}}}{[\,] : \mathtt{let\ \{\ x = T\ ?\ case\ x\ of\ \{\ T \to F\ \}\ \}\ in\ x} \Downarrow [\mathtt{x} \mapsto \mathtt{F}] : \mathtt{F}}$$

$$\text{where } \Gamma = [\mathtt{x} \mapsto \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}}]$$

In this derivation, the variable `x` is looked up in the heap twice, where at first the right (non-deterministic) branch is chosen and afterwards the left branch. Hence, `x` is bound to `T` as well as `F`, which violates the single assignment property of call-time choice.

With our semantics, there is one successful and one failing derivation but no derivation where `x` is bound to `T` as well as `F`:

$$\dfrac{\dfrac{\dfrac{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T} \Downarrow [\mathtt{x} \mapsto \blacksquare] : \mathtt{T}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}} \Downarrow [\mathtt{x} \mapsto \blacksquare] : \mathtt{T}}}{[\mathtt{x} \mapsto \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}}] : \mathtt{x} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}}}{[\,] : \mathtt{let\ \{\ x = T\ ?\ case\ x\ of\ \{\ T \to F\ \}\ \}\ in\ x} \Downarrow [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\mathtt{x} \mapsto \blacksquare] : \mathtt{x} \Downarrow \text{failure}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{case\ x\ of\ \{\ T \to F\ \}} \Downarrow}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}} \Downarrow}}{[\mathtt{x} \mapsto \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}}] : \mathtt{x} \Downarrow}}{[\,] : \mathtt{let\ \{\ x = T\ ?\ case\ x\ of\ \{\ T \to F\ \}\ \}\ in\ x} \Downarrow}$$

B Residualizing Semantics

Since the various rules of the residualizing semantics used in our partial evaluator are distributed over the paper and some of them were only informally sketched, we summarize in the following the complete set of rules of our residualizing semantics.

(Value)
$$\Gamma : v \Downarrow \Gamma : v \qquad \text{where } \begin{array}{l} v = c(\overline{x_n}) \text{ or } v = \langle\!\langle e' \rangle\!\rangle \text{ or } v \in \mathcal{V} \\ \text{with } (v \notin dom(\Gamma) \text{ or } \Gamma[v] = \text{free}) \end{array}$$

(VarExp)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \qquad \text{where } \begin{array}{l} e \notin \{\text{free}, \blacksquare\} \\ \text{and } (v \in \mathcal{V} \text{ or } v = c(\overline{x_n})) \end{array}$$

(VarDefer)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto e'] : \langle\!\langle x \rangle\!\rangle} \qquad \text{where } e \notin \{\text{free}, \blacksquare\}$$

(VarCase)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}] : x} \qquad \text{where } e \notin \{\text{free}, \blacksquare\}$$

(Flatten)
$$\frac{\Gamma[y \mapsto e_i] : \phi(x_1, \ldots, x_{i-1}, y, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v}{\Gamma : \phi(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v} \qquad \text{where } e_i \notin \mathcal{V}, y \text{ fresh}$$

(FunEval)
$$\frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \qquad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ proceed(\Gamma, f(\overline{y_n})) = \text{true} \end{array}$$

(FunDefer)
$$\Gamma : f(\overline{y_n}) \Downarrow \Gamma : \langle\!\langle f(\overline{y_n}) \rangle\!\rangle \qquad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ proceed(\Gamma, f(\overline{y_n})) = \text{false} \end{array}$$

(Let)
$$\frac{\Gamma[\overline{y_k \mapsto \sigma(e_k)}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \mathtt{let}\ \{\ \overline{x_k = e_k}\ \}\ \mathtt{in}\ e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_k \mapsto y_k}\}, \overline{y_k} \text{ fresh}$$

(Or)
$$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ ?\ e_2 \Downarrow \Delta : v} \qquad \text{where } i \in \{1, 2\}$$

(Free)
$$\frac{\Gamma[\overline{y_n \mapsto \text{free}}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \mathtt{let}\ \overline{x_n}\ \mathtt{free}\ \mathtt{in}\ e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

(Select)
$$\frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v} \qquad \text{where } \begin{array}{l} p_i = c(\overline{x_n}), \\ \sigma = \{\overline{x_n \mapsto y_n}\} \end{array}$$

(Guess)
$$\frac{\Gamma : e \Downarrow \Delta[x \mapsto \text{free}] : x \qquad \Delta[x \mapsto \sigma(p_i), \overline{y_n \mapsto \text{free}}] : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v}$$
$$\text{where } i \in \{1, \ldots, k\}, p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

(CaseDefer)
$$\frac{\Gamma : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \langle\!\langle \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

(CaseUnbound)
$$\frac{\Gamma : e \Downarrow \Delta : x}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}$$
$$\text{where } x \notin dom(\Delta)$$

(CaseCase)
$$\frac{\Gamma : e \Downarrow \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p'_j \to \langle\!\langle e'_j \rangle\!\rangle}\ \}}{\Gamma : \begin{array}{l} \mathtt{case}\ e\ \mathtt{of} \\ \quad \{\ \overline{p_k \to e_k}\ \} \end{array} \Downarrow \Delta : \begin{array}{l} \mathtt{case}\ x\ \mathtt{of} \\ \quad \{\ \overline{p'_j \to \langle\!\langle \mathtt{case}\ e'_j\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \} \end{array}}$$

(CaseVarCase)
$$\frac{\Gamma : e \Downarrow \Delta : x}{\Gamma : \begin{array}{l} \mathtt{case}\ e\ \mathtt{of} \\ \quad \{\ \overline{p_k \to e_k}\ \} \end{array} \Downarrow \Delta : \begin{array}{l} \mathtt{case}\ y\ \mathtt{of} \\ \quad \{\ \overline{p'_j \to \langle\!\langle \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \} \end{array}}$$
$$\text{where } \Delta[x] = \mathtt{case}\ y\ \mathtt{of}\ \{\ \overline{p'_j \to e'_j}\ \}$$