Calculation Carrying Programs

Zhenjiang Hu Masato Takeichi

Technical Report METR 99-07
Department of Mathematical Engineering
University of Tokyo

Summary.
In this paper, we propose a new mechanism called *calculation carrying programs* that can relax the tension between efficiency and clarity in programming. The idea is to accompany clear programs with some calculation specifying our intention of how to manipulate programs to be efficient. And this calculation specification can be executed automatically by our compiler to derive efficient programs. As a result, each calculation carrying program makes itself be a complete document including a concise specification of given problem as well as an effective way to derive both efficient and correct code.

Keywords: Program Calculation, Functional Programming, Transformational Programming, Meta Programming, Haskell

1 Introduction

Consider to write a program to check whether a list of numbers is *steep*. A list is said to be steep if each element of the list is greater than the average of the elements that follow it; a similar problem was discussed in [dMS98]. A straightforward program to solve the problem is

$$\begin{array}{lll}
steep & :: & [Int] \to Bool \\
steep\ [\,] & = & True \\
steep\ (a : x) & = & (a > average\ x)\ \wedge\ steep\ x \\
\\
average & :: & [Int] \to Int \\
average\ x & = & sum\ x\ /\ length\ x.
\end{array}$$

This program, though being clear, is terribly inefficient (a quadratic algorithm) due to repeated applications of *average* to the sublists. In fact, a linear efficient program does exist.

$$\begin{array}{lcl}
\mathit{steepOpt} & :: & [\mathit{Int}] \to \mathit{Bool} \\
\mathit{steepOpt}\ x & = & \mathbf{let}\ (st, s, l) = \mathit{steep}'\ x \\
 & & \mathbf{in}\ st \\
\mathit{steep}'\ [\,] & = & (\mathit{True}, 0, 0) \\
\mathit{steep}'\ (a : x) & = & \mathbf{let}\ (st, s, l) = \mathit{steep}'\ x \\
 & & \mathbf{in}\ ((a > (s/l)) \wedge st,\ a + s,\ l + 1)
\end{array}$$

Programmers are now forced to select one from the two programs by most practical programming systems, but this selection is essentially difficult.

- The straightforward one is of high readability and good modularity. It, however, comes at the cost of inefficiency, which may probably be intolerable. One may hope that a language compiler could automatically improve the program, but this is practically difficult. As far as we know, no popular Haskell compilers can automatically generate linear code from the straightforward program of *steep*.
- The efficient one is rather appealing, but it is far from being obvious why the program does correctly solve the problem without enough comment. Unfortunately, comment to the program is usually several lines in practice, which is too informal and too simple for program readers to understand algorithm completely. This makes the program difficult to be maintained, and even harder to be adapted to solve similar problems.

To remedy this situation, we shall propose a new mechanism called *calculation carrying programs* that can relax the tension between clarity and efficiency in programming. The idea is to accompany straightforward programs with some calculation specifying the intention of how to manipulate programs to be efficient. Thus, a calculation carrying program is not just means to show how to solve a problem, but also to show how to achieve improvement.

Program Calculation

Program calculation is a kind of program transformation based on the theory of *Constructive Algorithmics* (also known as *Bird-Meertens Formalisms*) [Bir87, Mal90, MFP91, Fok92, Bac95], which is a program calculus for program derivation. In Constructive Algorithmics, calculation is a series of applications of calculational laws (i.e. rules) that describe some properties of programs. Theorems may be used to capture larger steps in calculation in which there is ample opportunity for machine assistance. Theoretically, data types in constructive algorithmics are categorically defined as initial algebras of functors, and functions from one data type to another are represented as structure-preserving maps between algebras. By doing so, an orderly structure can be imposed on the program and such structure can be exploited to facilitate program transformation. This is in sharp contrast to the popular *fold/unfold* program transformation technique [BD77] whose emphasis is on the generality of transformation process instead of the structure of programs.

Our work on calculation carrying programs was highly motivated by the success of the application of program calculation both to derivation of various kinds of efficient programs [Gib92, dM92, Jeu93], and to construction of some optimization passes of compilers [GLJ93, OHIT97]. Particularly, it has been shown that many important program transformations such as *deforestation* (or *fusion*), *tupling transformation*, *parallelization* and *accumulation* can effectively and elegantly be formalized in calculational form [TM95, HIT96, HITT97, HTC98, HIT99].

Why to Code Calculation

We believe that it is both worthwhile and challenging to provide a flexible mechanism to code program calculations, and to make such calculations be part of programs. There are two main reasons.

- *Coding calculation can help programmers to document and reuse their program development process.*

  As argued in [dMS98], a typical functional programmer usually develops his program using calculation on the back of an envelope, and only records the final result in his codes, like the above program for solving the steep problem. Of course, he could document his ideas in comments, but as we all know, this is rarely done. Furthermore, when the programmer finds himself in a similar situation using the same technique to develop a new piece of code, there is no way he can reuse the development recorded as a comment.

- *Coding calculation can help to mechanize derivation of efficient programs.*

  Many calculation laws and theorems such as fusion, tupling, parallelization have been developed, but few of them have been fully implemented in practical compilers. There are two major difficulties. First, even for a simple calculation law like the cheap fusion in [GLJ93, TM95], one cannot code it as *naturally* as expressed in the paper. Rather one has to take pain to *design* an algorithm to implement the law by induction on the syntax tree. Second, the *creative steps*, which are often required during calculation, are hard to be mechanized in general. By coding calculation, we can program these creative steps by ourselves and only leave those parts that can be mechanized for compiler.

Our Work

This paper makes the first attempt at the design and implementation of a (functional) language that can support calculation carrying programs. In this language, programmers can write both a straightforward solution to a problem as they usually do, and a calculation declaring their intention for transforming the solution better. As a result, each calculation carrying program makes itself be a complete document including a concise specification of given problem as well as an effective way to derive both efficient and correct code.

To realize calculation carrying programs, we are confronted with several design issues; how to code calculation in a natural and declarative way, how to ensure the correctness of generated programs by calculation, how to make object programs and meta programs coexist well in a single framework.

The two important technical contributions of this paper are as follows.

- We design a core language (Section 3) for the purpose of writing the calculation carrying programs, which has the following two distinguishing features.
  - First, the language is very similar to existing functional languages like Haskell, but it is unique in that the patterns that are used in λ abstraction, **let** binding and **case** matching are generalized to be higher order ones [HL78, Hec88, dMS98, dMS99]. With this explicit higher order matchings, we are able to code calculation in a very natural way, where termination and correctness can be guaranteed.
  - Second, the typing system of the language can guarantee the correctness (both in type and in meaning) of generated programs after calculation. This correctness issue has been considered a big difficulty in meta programming [TS97, TBS98]. We tackle this problem by not allowing *open* code (code containing free variables) to be generated. This leads to a integrated framework in which object programs and meta programs can co-exist well.
- We have implemented an experimental programming environment to support developing calculation carrying programs. The main point is that the system can automatically derive efficient and correct programs by executing calculations. As shown in Section 4, we have successfully applied it to many interesting practical examples, demonstrating how one can concisely code calculation rules, calculation strategies, and even algorithmic development process. To the best of our knowledge, we have not seen any other similar systems which can do as concisely and powerfully as ours.

The rest of this paper is organized as follows. We illustrate informally the basic idea of calculation carrying programs by two simple examples in Section 2. Then in Section 3, we give the formal definition of the core language for writing the calculation carrying programs, including its syntax, semantics and typing rules. More application examples for coding calculation rules, calculation strategies, and development processes are given in Section 4. Finally, we discuss the related work and make a concluding remarks in Section 5 and Section 6 respectively.

2 Basic Idea: Calculation Carrying Programs

Each calculation carrying program consists of two parts: an object program that describes concise solution to given problem, and a meta program that describes how to improve the object program to be more efficient one. In this section, we will illustrate informally the basic idea by two simple examples. More practical examples can be found in Section 4.

2.1 Coding Calculation Rules by Matching

Before giving a whole calculation carrying program, we start with coding the following famous *fusion* calculation rule:

$$\frac{\begin{array}{ll} f\ e & = e' \\ f\ (a \oplus r) & = a \otimes f\ r \end{array}}{f \circ foldr\ (\oplus)\ e = foldr\ (\otimes)\ e'}$$

reading that one can fuse the composition of a function f and a $foldr$ structure into a single $foldr$, provided that the two promotable conditions are satisfied. This rule plays an important role not only in calculating efficient functional programs [Bir89, MFP91], but also in compiler construction like the warm fusion [LS95] in the Glasgow Haskell Compiler (GHC).

Even for such a simple calculation rule just in three lines, it requires a tedious work to implement it. The reason is that one cannot code the rule as *naturally* as expressed as above. Rather one has to *reinvent* an algorithm to implement the law by induction on the syntax tree of programs. To remedy this situation, we introduce explicit *matching* to our language.

Recall that pattern matching is a well-appreciated concept of functional programming [Pey88]. It contributes to concise function definitions by implicitly decomposing data type values. For example, the following defines *length*, a function to compute the length of a list, according to two list patterns; an empty list $[\,]$, and a list whose head element is a and the rest list is x.

$$\begin{array}{ll} length\ [\,] & = 0 \\ length\ (a : x) & = 1 + length\ x \end{array}$$

Pattern matching indeed provides a good way for describing manipulation of data, but it does not fulfill our needs to specify manipulation of programs. If we would insist on using usual pattern matching of program syntax trees to code the fusion calculation rule, we would need a second invention to express the procedure showing how to derive $\otimes$ from $\oplus$ and f by traversing the syntax trees explicitly, which would lead to a rather long program.

Our idea is to relieve pattern matching from the restriction that pattern to be matched must be constructed by data constructors and pattern variables, allowing any *expression* (higher order patterns) to be a target for matching. With this general matching, we can now code fusion simply as follows.

$$\begin{array}{l} fusion\ ::\ <[a] \to b> \ \to\ <[a] \to b> \\ fusion\ <f \circ foldr\ (\oplus)\ e> \\ \quad = \mathbf{letm} \\ \qquad\quad e' \quad\ = f\ e \\ \qquad\quad a \otimes r = f\ (a \oplus x) \Leftarrow\ r == f\ x \\ \qquad \mathbf{in} \\ \qquad\qquad <foldr\ (\otimes)\ e'> \end{array}$$

The *fusion* defines a calculation transforming the code of an expression with type $[a] \to b$ to another. We use brackets $<>$ to surround expressions (or types) to denote expression codes (or code types). To see the meaning of the definition of $fusion$, we demonstrate how

$$fusion \;\; < length \circ foldr\ (:)\ [\,] >$$

works. Upon receiving the expression code of $< length \circ foldr\ (:)\ [\,] >$, $fusion$ matches it with $< f \circ foldr\ (\oplus)\ e >$ to bind f, $\oplus$ and e, and gets

$$\begin{array}{ll} f & \mapsto length \\ \oplus & \mapsto : \\ e & \mapsto [\,]. \end{array}$$

Then with these bindings, we reduce $f\ e$ to a form that does not contain any β and η redex, and then match it with e' to bind e', and gets

$$e' \mapsto 0$$

since the reduction of $length\ [\,]$ gives 0. Similarly, matching the reduction result of $f\ (a \oplus x)$ with $a \otimes r$ gives several bindings for $\otimes$ and r, from which we only choose one such that r is syntactically equivalent to $length\ x$:

$$\otimes \mapsto \lambda a.\lambda x.\,(1 + x)$$

Finally, we build the code by replacing $\otimes$ and e' by their bindings in $< foldr\ (\otimes)\ e' >$, and get

$$< foldr\ (\lambda a.\lambda x.\,(1 + x))\ 0 > .$$

Formal account of the meaning of the language can be found in Section 3.

It is worth noting that the important role of (second order) matching to express powerful transformation has been recognized by Huet and Lang [HL78], but the matching there are only used for implementation transformation systems without being embedded into a functional language. Heckmann [Hec88] combined features of functional language with special means of pattern specification language to describe tree transformation on abstract syntax tree. The language he proposed is rather complicated. In contrast, we take advantage of higher order matching to describe transformation in a declarative way rather than using lower-level traversal of abstract syntax trees. Most recently, de Moor and Sittampalam [dMS99] present a simple but practical algorithm for higher-order matching in the context of automatic program transformation.

2.2 A Complete Calculation Carrying Program

Figure 1 gives the example of a complete calculation carrying program for the steep problem. It contains three parts: a straightforward program, a calculation, and a relationship between them. The straightforward program has been given in the introduction, we will concentrate on the calculation part, showing how to code our calculation to make it be efficient.

As explained in the introduction, the straightforward program is inefficient because of redundant repeated computation of *sum* and *length*. This inefficiency can be handled by tupling *steep*, *sum* and *length*, using the tupling transformation [Chi93, HITT97]. The specific tupling transformation for *steep* is coded in the

– **Straightforward solution:**

$$\begin{array}{lcl}
\mathit{steep} & :: & [\mathit{Int}] \to \mathit{Bool} \\
\mathit{steep}\ [\,] & = & \mathit{True} \\
\mathit{steep}\ (a : x) & = & (a > \mathit{average}\ x)\ \wedge\ \mathit{steep}\ x \\
 & & \\
\mathit{average} & :: & [\mathit{Int}] \to \mathit{Int} \\
\mathit{averag}\ x & = & \mathit{sum}\ x\ /\ \mathit{lengh}\ x
\end{array}$$

– **Calculation:**

```
steepOpt :: < [Int] → Bool >
steepOpt = letm
               e₁ = steep [ ]
               e₂ = sum [ ]
               e₃ = length [ ]
               a ⊕₁ (average x, steep x) = steep (a : x)
               a ⊕₂ (sum x) = sum (a : x)
               a ⊕₃ (length x) = length (a : x)
               g (sum x) (length x) = average x
             in
               < λx. let (st, s, l) = steep' x in st
                 steep' [ ] = (e₁, e₂, e₃)
                 steep' (a : x)
                      = let (st, s, l) = steep' x
                        in (a ⊕₁ (g s l, st), a ⊕₂ s, a ⊕₃ l)
               >
```

– **Relation between** *steep* **and** *steepOpt***:**

$$\mathit{steep}\ \Rightarrow\ \mathit{steepOpt}$$

Fig. 1 A Calculation Carrying Program for the Steep Problem.

calculation part. We use matching to extract body structures from *steep*, *sum* and *length* respectively, and then glue them together inside code-generation brackets <>.

Expanding the relation between *steep* and *steepOpt*, our system will automatically give the following efficient program:

```
steep          = λx. let (st, s, l) = steep' x in st
steep' []      = (True, 0, 0)
steep' (a : x) = let (st, s, l) = steep' x
                 in
                   let
                     x ⊕₁ (y, z) = (x > y) ∧ z
                     x ⊕₂ y = x + y
                     x ⊕₃ y = 1 + y
                     g s l = s/l
                   in
                     (a ⊕₁ (g s l, st), a ⊕₂ s, a ⊕₃ l)
```

which is essentially the same as the efficient one given in the introduction. Note an alternative way to use **let** instead of explicit substitution when instantiating bound variables of $\oplus_i$'s and g in code generation.

3 Formal Development

In this section, we give the formal definition of the core language for writing the calculation carrying programs, including its syntax, semantics and typing rules. Rather than inventing a completely new language, we tried our best to extend the existing functional languages as little as possible, so that those who are familiar with a functional language should have no difficulty in using and understanding our language. The features of our language, as summarized in Figure 2, are two-fold.

- First, it is similar to existing functional languages like Haskell. The crucial difference is that patterns that are used in λ abstraction, **let** binding and **case** matching are extended to be higher order ones which may contain function variables. To make this more explicit, we introduce three new constructs, namely λ^m, **letm**, and **casem**.
- Second, it is a kind of meta language inspecting and generating codes, but with the requirement that *open* code containing free variables not be allowed to be generated. The advantage of doing so is that we can guarantee the type correctness of generated programs, which have been considered a big difficulty in meta programming [TS97, TBS98].

3.1 Object Language: Describing Naive Solutions

The object language, as defined by the upper part in Figure 2, is nothing special but a subset of Haskell, a polymorphically typed pure functional language. We do not discuss their details. Rather we give more examples for the definitions of functions used so far or to be used later.

$$\begin{array}{lll} sum & :: & [Int] \to Int \\ sum & = & \lambda x.\,\mathbf{case}\ x\ \mathbf{of} \\ & & \quad [\,] \to 0 \\ & & \quad (a:x) \to a + sum\ x \\ \\ length & :: & [Int] \to Int \\ length & = & \lambda x.\,\mathbf{case}\ x\ \mathbf{of} \\ & & \quad [\,] \to 0 \\ & & \quad (a:x) \to 1 + length\ x \\ \\ foldr & :: & (a \to b \to b) \to b \to [a] \to b \\ foldr & = & \lambda(\oplus).\lambda e.\lambda x.\mathbf{case}\ x\ \mathbf{of} \\ & & \qquad [\,] \to e \\ & & \qquad (a:x) \to a \oplus foldr\ (\oplus)\ e\ x \end{array}$$

For the sake of readability, we sometimes take liberty to use some familiar syntactic sugars like infix notations, pattern matching equations instead of **case** constructs, **let** binding instead of λ abstraction, and etc.

3.2 Meta Language: Coding Calculation

The meta language basically consists of two parts. One is for encoding and decoding expression, namely $< e >$ and $\$e^{\$}$. And the other is for inspecting expressions by higher order matching as used in λ^m, **letm** and **casem**. We explain each construct informally below, before giving the formal definition later.

- $< e >$

  It is used to build expression code. For instance,

  $$< 2 + 3 >$$

  builds the code of expression $2 + 3$. If e has some free variable like

  $$< \lambda x.x + y >$$

  we require the free variables (y for this example) be bound. For instance, if y is bound as

  $$y \mapsto 2 + 3$$

  then the e is equal to the code

  $$< \mathbf{let}\ y = 2 + 3\ \mathbf{in}\ \lambda x.x + y > .$$

- $e^{\$}$

  It is used for decoding, being analogous to the *unquote* in Lisp. Basically, we have

  $$< e >^{\$} \text{ evaluates to } e.$$

- $\lambda^m e_p.\, e_b$

  It is used to define a meta function, which matches the input with e_p to bind the free variables in e_p, and then computes e_b. For example, the fusion calculation rule in Section 2 can be specified by

  $$\begin{aligned} fusion = \lambda^m &< f \circ foldr\ (\oplus)\ e > . \\ &(\mathbf{letm} \ldots \mathbf{in} \ldots). \end{aligned}$$

  For readability, we sometimes write $f = \lambda^m e_p.\, e_b$ as

  $$f\ e_p = e_b.$$

  Note that we require the pattern expression e_p (also that in the later letm and casm) be tight (i.e., has neither β nor η redex) to take advantage of the higher-order matching algorithm in [dMS99].

- **letm** $e_p = e_b \Leftarrow e_c$ **in** e

  It is used to match expression e_b with expression pattern e_p to bind free variables in e_p while satisfying the condition e_c, and then to compute e as its result. This e_c is useful to reduce the number of solutions[1] of bindings when matching e_b with e_p. With this construct, we can code a simple one-step hoisting transformation by

  ```
  $hoist1\ ::\ < a >\ \rightarrow\ < a >$
  $hoist1\ < e >\ =$
    $\mathbf{letm}$
      $c\ (\mathbf{let}\ x\ =\ (\mathbf{let}\ y = e_0\ \mathbf{in}\ e_1)\ \mathbf{in}\ e_2)\ =\ e$
        $\Leftarrow not(y\ `isElem`\ fv\ e_2)$
    $\mathbf{in}$
      $< c\ (\mathbf{let}\ y = e_0\ \mathbf{in}\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)) >$
  ```

  If y is not free in e_2 under a context c, we can hoist y up. Here, $fv\ e_2$ computes all free variables in e_2.

- **casem** e **of** $e_{p_1} \rightarrow e_1; \ldots; e_{p_n} \rightarrow e_n$

  This construct provides a convenient way to specify manipulation of expression case by case. It will match e with $e_{p_1}, \ldots, e_{p_n}$ one by one till it succeeds, say at e_{p_i}, and it will then compute e_i to give the result. It should be noted that a meta expressions does not need to return an expression code as result. As an example, the following definition of *isContext*, determining whether an expression represents a context, gives a boolean result.

  ```
  $isContext ::\ < a >\ \rightarrow\ Bool$
  $isContext = \lambda^m < e > .\ \mathbf{casem}\ e\ \mathbf{of}$
                          $\lambda x.e' \rightarrow x\ `isElem`\ fv\ e'$
                          $\_\quad\ \rightarrow False$
  ```

  Here, an expression is said to be a context if it is a function with its bound variable appearing in the body as holes.

[1] Note that matching two expression terms may give many solutions, which is different from the pattern matching in functional languages [Pey88].

3.3 Semantics

The semantics for the core language is quite similar to that of general functional languages except for the following two points.

- Our expression evaluates to a *list* of values rather than a single one, because our core language allows higher order pattern matching which may compute *many* solutions. This is opposed to data constructor pattern matching [Pey88] whose solution is unique.
- We add expression codes to the resulting value domain to treat them as first-class values, so that we can manipulate expression codes in a similar way as we manipulate data like lists.

We shall take a closer look at the higher-order matching we use, before giving the semantics of the language.

Higher-order Matching

Higher-order matching plays an important role in our meta language. Given two expressions e_p (the pattern) and e_t (the term) in the object language, matching is to find all possible substitutions ϕ such that

$$\phi\ e_p = e_t$$

Here equality is taken modulo renaming (α-conversion), elimination of redundant abstraction (η-conversion), and substitution of arguments for parameters (β-conversion). We call such ϕ a *match*, which is a map from variables to expressions:

$$\phi :: \mathbf{Var} \rightarrow \mathbf{Exp}$$

where **Exp** denotes all object expressions.

Clearly it is undesirable that matching gives a potentially infinite set of matches in the context of automatic program transformation. In [HL78], Huet and Lang suggested restricting attention to matching of *second-order* terms, and gave a matching algorithm which is both *decidable* and *complete* to compute a finite number of incomparable matches. Recently, de Moor and Sittampalam [dMS99] extended the second-order matching algorithm, and present a new one to suit transformation of Haskell programs. The new algorithm can accept higher-order and polymorphically typed terms, sharing the property that it returns a well-defined, finite set of matches. We will not be involved in more detailed discussion on higher-order matching. Rather we use these results in this paper through the function *matching*:

$$matching :: \mathbf{Exp} \rightarrow \mathbf{Exp} \rightarrow [\mathbf{Var} \rightarrow \mathbf{Exp}]$$

which accepts two expressions e_p and e_t, and returns a list of matches. For instance,

$$matching\ (\lambda x.\, v\ x\ c)\ (\lambda x.\, c'\ x\ c)$$

gives two matches, namely

$$\{v \mapsto c'\} \quad \text{and} \quad \{v \mapsto \lambda x.\lambda y.\, c'\ x\ c\}.$$

Here c and c' represent some constants.

As it is required in [dMS99] that the pattern e_p be free of η-redex, and that the term e_t be free of β and η redex, we use the function

$$reduce :: \mathbf{Exp} \to \mathbf{Exp}$$

to reduce β and η redex in an expression.

Interpretation

To interpret meta expressions to values, we extend our ordinary values with expression codes such that expressions can be manipulated in a similar way as other values like integers. We use **Env** for the environment, and **Val** for the values. The environment maps a variable to a value:

$$\mathbf{Env} \to \mathbf{Val}.$$

Figure 3 gives a formal definition of the semantics of the core language. For simplicity, we have assumed that each bound variable has been renamed with a unique name, and hence we do not consider name conflicts in definition of semantics.

$\mathcal{E}$ evaluates a meta expression, under an environment, to a *list* of values instead of a single one. This is due to many possible matches for a single matching. Using list here resembles the technique used to deal with nondeterminism in construction of the monadic parser [HM98]. $\sqcup$ flattens a list of lists. $\rho_1 \oplus \rho_2$, for extending environment ρ_1 with ρ_2, is defined by

$$\begin{aligned}(\rho_1 \oplus \rho_2)\ x &= \rho_2\ x,\ x \text{ is defined in } \rho_2\\ &= \rho_1\ x,\ \text{otherwise.}\end{aligned}$$

The main characteristics of $\mathcal{E}$ is that it binds local variables using matching. With this in mind, it should not be difficult to understand its definition. Note that we use the list comprehension notation in definition of $\mathcal{E}$.

Now, the meaning of $f \Rightarrow f'$ is simple, just to associate f with one of the codes from computation of $\mathcal{E}[\![f']\!]\rho$.

3.4 Type System

In typing, there is no big difference between the higher-order matching in pure language and the ordinary matching in traditional functional languages. To handle meta codes, we just extend the traditional type system to include the type to represent codes. Thus the syntax for our types are

$$\begin{array}{rlll}\text{Polytype}: & \sigma ::= & \forall\alpha.\,\sigma & \\ & \mid & \tau & \text{monotype}\\ & \mid & <\tau> & \text{code type}\\ \text{Monotype}: & \tau ::= & \alpha & \text{type variable}\\ & \mid & b & \text{base type like } Int\\ & \mid & \tau_1 \to \tau_2 & \text{function type}\end{array}$$

Figure 4 summarizes the typing rules, each of which is defined by a judgment $\Gamma \vdash e :: \sigma$, where e is our well-typed expression, σ is the type of the expression, and Γ is the environment assigning types to term variables. Since we do not allow to generate open code, we can thus treat variables in meta codes in the same way as in other places, and easily guarantee the correctness of generated codes.

Proposition 1 (Type Soundness) For all well-typed expression, $\Gamma \vdash e :: \sigma$, then $\mathcal{E}[\![e]\!] :: [[\![\sigma]\!]]$, where $[\![\cdot]\!]$ is a map from type attribute to its set of value meanings. $\Box$

4 More Application Examples

In Section 2, we have illustrated the idea of calculation carrying programs by two simple examples. We will give more practical application examples in this section. Specifically, we will demonstrate in general how to declare calculational rules, calculational strategies, as well as program development process.

4.1 Coding Calculational Rules

Rules (laws) are most fundamental to transform expressions. In our language they are naturally described by using higher-order matching.

Simple Rules

Consider to define the rule transforming $0 + x$ to x. We can simply code it as

$$\begin{array}{l} removeZero :: \; < Int > \;\; \rightarrow \;\; < Int > \\ removeZero = \lambda^m \;\; < 0 + x > . < x > . \end{array}$$

For readability, we often write the above as:

$$removeZero \;\; < 0 + x > = < x > .$$

Two remarks are worth making. First, the expressions to be matched can have multiple occurrences of the same variable [HL78], i.e., the rules are not necessary to be left linear. So,

$$\begin{array}{l} sum2Double :: \;\; Num\ a \Rightarrow < a > \rightarrow < a > \\ sum2Double = \lambda^m < x + x > . < 2 * x > \end{array}$$

is a valid rule.

Second, like programming in ordinary pattern matching, we can define calculation in a case-by-case way. For example, the following *concatRm* are used to remove the concatenation operator "$+\!\!+$" by matching for two cases:

$$\begin{array}{l} concatRm \;\; :: \;\; < [a] > \rightarrow < [a] > \\ concatRm \;\; < [\,] +\!\!+ x > \\ \quad\quad = \;\; < x > \\ concatRm \;\; < (a : x) +\!\!+ y > \\ \quad\quad = \;\; < a : (concatRm \;\; < x +\!\!+ y >)^{\$} > \end{array}$$

This is equivalent to

$$\begin{array}{l} concatRm = \\ \quad \lambda^m < e > . \textbf{casem}\ e\ \textbf{of} \\ \quad [\,] +\!\!+ x \qquad \rightarrow < x > \\ \quad (a : x) +\!\!+ y \rightarrow < a : (concatRm < x +\!\!+ y >)^{\$} > . \end{array}$$

Tuping Calculation Rule

In Section 2, we have shown how to code a fusion calculation rule. We will show how to code another very important one called *tupling* [Fok92, HITT97]:

$$\frac{\begin{array}{lll} h\ x & = & (f\ x, g\ x) \\ f\ [\,] & = & e_1 \\ f\ (a:x) & = & a \oplus (f\ x, g\ x) \\ g\ [\,] & = & e_2 \\ g\ (a:x) & = & a \otimes (f\ x, g\ x) \end{array}}{\begin{array}{l} h = foldr\ (\odot)\ (e_1, e_2) \\ \quad \text{where} \\ \qquad a \odot (x,y) = (a \oplus (x,y), a \otimes (x,y)) \end{array}}$$

which says that tupling of mutumorphisms (regular mutual recursively defined functions) yields a catamorphism (fold).

This rule can be straightforwardly programmed by

$$\begin{array}{lll} tupling & :: & < [a] \to (b,c) > \to < [a] \to (b,c) > \\ tupling & = & \lambda^m < \lambda x.(f\ x, g\ x) > . \\ \multicolumn{3}{l}{\quad \textbf{letm}} \\ & & \begin{array}{lll} e_1 & = & f\ [\,] \\ a \oplus (f\ x, g\ x) & = & f\ (a:x) \\ e_2 & = & g\ [\,] \\ a \otimes (f\ x, g\ x) & = & g\ (a:x) \end{array} \\ \multicolumn{3}{l}{\quad \textbf{in}\ < \textbf{let}\ a \odot (x,y) = (a \oplus (x,y), a \otimes (x,y))} \\ \multicolumn{3}{l}{\qquad\quad \textbf{in}\ foldr\ (\odot)\ (e_1, e_2) > .} \end{array}$$

Following this way, it should not be difficult to code other interesting calculation laws such as the parallelizing theorem [HTC98], the accumulation calculation theorem [HIT99], and the diffusion calculation law [HTI99].

4.2 Coding Calculation Strategies

Valid transformations on program can be described by a set of calculation rules; while calculation strategies are applied to obtain the desired optimization effects [VBT98]. In this section, we would like to demonstrate that those important strategies in [VBT98] can be programmed here in a more concise and direct way.

Basically, the sequential application of two rules r_1 and r_2 to a term $< t >$ can be coded by

$$r_2\ (r_1\ < t >)$$

and the choice application of rules $r_1, \ldots, r_n$ to term $< t >$ can be coded something like

$$\begin{array}{l} \textbf{casem}\ \mathrm{t}\ \textbf{of} \\ \quad case_1 \to r_1\ t \\ \qquad \cdots \\ \quad case_n \to r_n\ t \end{array}$$

as we have shown in the definition of *concatRm*.

To appreciate its use, suppose we want to apply the fusion calculation *fusion* to an expression to remove as many function compositions as possible.

$$applyFusion \ :: \ \ < a > \to < a >$$

To this end, we repeatedly select the fusible subexpressions and apply the fusion calculation to them, until we have no fusible subexpressions any more. This is coded by

```
$applyFusion\ < e >=$
  **casem** $e$ **of**
    $c\ (\lambda x. f\ (g\ x)) \to$
      **let** $h = fuse2\ < f >\ \ < g >$
      **in if** $h\ != \ Fail$
        **then** $applyFusion\ < c\ h >$
        **else let** $c' = (applyFusion\ < c >)^{\$}$
                   $f' = (applyFusion\ < f >)^{\$}$
                   $g' = (applyFusion\ < g >)^{\$}$
              **in** $< c'\ (\lambda x.(f'\ (g'\ x))) >$
    $\_ \to < e >$
```

where *fuse*2 is a meta function to fuse two functions into a single one, defined by

```
$fuse2 :: \ < b \to c > \to < a \to b > \to < a \to c >$
$fuse2 = \lambda^m < f > .\lambda^m < g > .$
           **letm** $e = g\ [\,]$
                $(a \oplus g\ x) = g\ (a : x)$
           **in** $fusion\ < f \circ foldr\ (\oplus)\ e >\ .$
```

4.3 Coding Programming Development

The creative steps in transformational programming are very difficult to be implemented in a fully automatic way. The calculation carrying code provides us with a flexible means to code these creative steps.

Consider we want to develop an efficient program to reverse a list. A naive definition is

$$\begin{array}{ll} rev & ::\ [a] \to [a] \\ rev\ [\,] & = [\,] \\ rev\ (a : x) & = rev\ x +\!\!+ [a] \end{array}$$

which is a quadratic program. We would like to accompany it with a calculation to turn it to be a linear one. It has been shown in [Hug86] that we need two creative steps for this improvement. First, we need to introduce an accumulation parameter starting from $[\,]$, as define by

$$\begin{array}{ll} rev\ x & = fastrev\ x\ [\,] \\ fastrev\ x\ y & = rev\ x +\!\!+ y. \end{array}$$

Second, to derive efficient definition for *fastrev*, we need to apply the fusion calculation to $(+\!\!+\ y) \circ reverse$, where we are required to use the associative property of $+\!\!+$:

$$(x +\!\!+ y) +\!\!+ z \ = \ x +\!\!+ (y +\!\!+ z).$$

Although these two steps are difficult to be made automatic in general, we are able to code them as

$$\begin{array}{ll} \mathit{revOpt} :: & < [a] \to [a] > \to < [a] \to [a] > \\ \mathit{revOpt} = & \mathbf{letm}\ e = [\,] +\!\!+\, y \\ & \qquad a \oplus (x +\!\!+\, y) = (\mathit{assoc}\ < \mathit{rev}\ (a : x) +\!\!+\, y >)^{\$} \\ & \mathbf{in}\ < (\lambda y.\, \mathit{foldr}\ (\oplus)\ e)\ [\,] > \end{array}$$

where *assoc* specifies the association rule,as defined by:

$$\begin{array}{ll} \mathit{assoc} & :: \ < a > \ \to \ < a > \\ \mathit{assoc}\ < (x +\!\!+\, y) +\!\!+\, z > & = \ < x +\!\!+\, (y +\!\!+\, z) > . \end{array}$$

To complete the calculation carrying program for *rev*, we associate the naive definition of *rev* with the the above calculation by

$$\mathit{rev} \Rightarrow \mathit{revOpt}.$$

Now our compiler can automatically derive *fastrev* after executing the above program.

This example shows that our framework is helpful to support transformational programming approach to develop efficient programs.

5 Related Work

Incorporating the use of higher order matching (or unification) to concisely express program transformations have been seen in many systems, such as Ergo [PE88], KORSO [BLSW95] and MAG [dMS98]. Our work has received much influence from the MAG system which is designed to support generic transformational programming. Like our system, MAG could associate the original clear program with a theory consisting of the optimization rules needed to obtain the second efficient program. However, MAG uses higher order matching only *implicit* by the system. In contrast, we design a language with general matching mechanism so that programmers can express explicitly what exactly they want to match. This provides a flexible way to code transformation (calculation) strategies and program development process as seen in Section 4, which would be very difficult in MAG.

There are a number of specialized *pattern languages* for the purpose of program inspection and transformation [Hec88, AFFW93, Mal93, Cre97, Big98]. Often, these do not include higher order patterns. Instead, they provide a set of primitives for matching and building subtrees, but for the most part require that tree traversal be programmed explicitly in *imperative* style, node by node. In particular, we should compare our work with that in [Hec88]. In [Hec88], a language was proposed to combine features of general purpose functional language with special means to specify tree transformations, and much effort was denoted to design a pattern specific language allowing for partitioning syntax trees by arbitrary vertical and horizontal cuts. This work has the same purpose as ours to embed powerful pattern matching into a functional language, but it uses many lower-level matching constructs. In contrast, our use of the higher order matching makes our programs more concise and more readable than those in [Hec88]. It would be interesting to see what primitive patterns suggested in [Hec88] can be profitably combined with higher order matching.

Our work is also related to the work on meta programming. MetaML [TS97] is a statically-typed multi-stage programming language, allowing the programmer to construct, combine, and execute code fragments. But using MetaML to code calculation would be difficult and complicated, because it does not have powerful matching mechanism. PATH [TH98] is a system in which an intermediate meta language was designed for transforming functional programs. Different from our system where we describe calculation independently, PATH uses annotations to describe application of transformation rules to programs, and programmers need to add suitable annotations during transformation. There are other languages for scripting transformation like [Pul99], which are not so abstract enough for programmer to specify their intention for manipulating programs as ours.

The system RML optimizer [VBT98] is close in spirit to our system. It studies how to transform the source code of a program into another one in the compiler for optimization, based on ideas from term rewriting. It argues that a good way is to use explicit specification of rewriting strategies, and it shows that it is possible to break down rewrite rules into two primitives: *matching* against term patterns and *building* terms. Unlike our system, all these are done inside compiler rather than being open to programmers. So, their implementation of the transformation strategies in terms of the matching/building primitives is much more complicated and difficult to understand than our coding of calculation strategies as shown in Section 4.

There are a lot of work on developing fast implementation of matching algorithm [CPT92, CQS96, dMS99]. Particularly the Oxford Programming Tools Group (`http://www.comlab.ox.ac.uk/oucl/groups/progtools/`) are now actively researching on this topic.

Finally, this work is a continuation of our effort in application of program calculation technique to derivation of efficient programs and to construction of optimizing passes in compilers. We have developed many cheap but powerful calculational rules such as fusion [HIT96, OHIT97], tupling [HITT97], accumulation [HIT99], parallelization [HTC98], and diffusion [HTI99]. Although all these can be implemented in an automatic way in theory, it is quite time-consuming to implement them in practice; implementation of the HYLO fusion calculator [OHIT97] in the Glasgow Haskell Compiler has actually taken us about two years. We believe that we are now in the position to construct an environment for coding and implementing calculation rules.

6 Concluding Remarks

We have proposed a new framework for writing calculation carrying programs, for the purpose of relaxing the tension between clarity and efficiency in programming. As demonstrated by several convincing examples, this framework has shown its significance and promise both in support of transformational/calculational programming and in compiler construction based on program transformation.

It should be noted that we do not need folding in our transformation at all. This is in sharp contrast to many existing transformation systems that are based on fold/unfold transformation technique. We believe that the partial correctness problem and high implementation cost of fold / unfold transformation prevent it from transforming large scale programs [GLJ93, TM95].

Although our framework works on functional languages, we could extend it to be applicable for logic or imperative languages. As to our future work, we should study more on higher-order meta functions and the modularity of calculations, both of which have not been fully addressed in this paper. Furthermore, we are planning to port the current experimental system to Gofer to make it be really **Go**od **f**or **e**xpressing **r**easoning.

References

[AFFW93] M. Alt, C. Fecht, C. Ferdinand, and R. Wilhelm. The Trafola-H system. In B. Krieg-Bruckner and B. Hoffmann, editors, *PROgram Development by SPECification and TRAnsformation*, volume 680 of *LNCS*, pages 539–570, 1993.

[Bac95] R. Backhouse. The calculational method. *Special Issue on the Calculational Method, Information Processing Letters*, 53:121, 1995.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Big98] T. Biggerstaff. Pattern matching for program generation: A user manual. Technical Report Technical Report MSR TR-98-55, Microsoft Research, 1998.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[BLSW95] B.K. Bruckner, J. Liu, H. Shi, and B. Wolff. Towards correct, efficient and reusable transformational developments. In *KORSO: Methods, Languages, and Tools for Construction of Correct Software,* LNCS 1009, pages 270–184. Springer-Verlag, 1995.

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[CPT92] J. Cai, R. Page, and R.E. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, 1992.

[CQS96] R. Curien, Z. Qian, and H. Shi. Efficient second-order matching. In *7th International Conference on Rewriting Techniques and Applications*, pages 317–331. Springer Verlag (LNCS 1103), 1996.

[Cre97] R.F. Crew. A language for examining abstract syntax trees. In *Proc. of the Conf. on Domain-Specific Languages*. Usenix, 1997.

[dM92] O. de Moor. *Categories, relations and dynamic programming.* Ph.D thesis, Programming research group, Oxford Univ., 1992. Technical Monograph PRG-98.

[dMS98] O. de Moor and G. Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[dMS99] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. Draft avaliable at `http://www.comlab.ox.ac.uk/oucl/peple/oege.demoor.html`, April 1999.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[Gib92] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction* (LNCS 669), pages 122–138. Springer-Verlag, 1992.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[Hec88] R. Heckmann. A functional langauge for the specification of complex tree transformation. In *Proc. ESOP (LNCS 300)*, pages 175–190, 1988.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order pattersn. *Acta Informatica*, 11:31–55, 1978.

[HM98] G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[HTI99] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[Hug86] R. J. M. Hughes. A novel representation of lists and its application to the function reverse. *Information Processing Letters*, 22(3):141–144, March 1986.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, (14):255–279, August 1990.

[Mal93] A. Malton. The denotational semantics of a functional tree-manipulation language. *Computer Languages*, 19(3):157–168, 1993.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM PLDI*, pages 199–208. ACM Press, 1988.

[Pey88] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1988.

[Pul99] H. Pull. The partial evaluator script language definition – version 2.0. Technical Report Internal Report IC/FPR/LANG/2.3/1, Department of Computing, Imperial College, July 1999.

[TBS98] W. Taha, Z. Bnaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, Aalborg, Denmark, July 1998.

[TH98] M. Tullsen and P. Hudak. An intermediate meta-language for program transformation. Research report yaleu/dcs/rr-1154, Department of Computer Science, Yale University, June 1998.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 203–217, Amsterdam, June 1997. ACM Press.

[VBT98] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizer with rewriting strategies. In *ACM SIGPLAN International Conference on Functional Programming*, pages 13–26, Baltimore, MD USA, September 1998. ACM Press.

Definition:

def	$::= f = e$	function definition

Expression:

		— Object Part —
$e ::=$	v	expression variable
$\mid$	n	constant
$\mid$	$\lambda v.\, e$	lambda abstraction
$\mid$	$e_1\ e_2$	function application
$\mid$	$\textbf{case}\ e\ \textbf{of}\ p_1 \to e_1; \ldots; p_n \to e_n$	case expression
		— Meta Part —
$\mid$	$< e >$	code of expression
$\mid$	$e^{\$}$	expression from code
$\mid$	em	expression with higher-order matching

Expression with Higher-order Matching:

$em ::=$	$\lambda^m e_p.\, e_b$	generalized lambda expression
$\mid$	$\textbf{letm}\ e_p = e_b \Leftarrow e_c\ \textbf{in}\ e$	generalized let expression
$\mid$	$\textbf{casem}\ e\ \textbf{of}\ e_{p_1} \to e_1; \ldots; e_{p_n} \to e_n$	generalized case expression

Simple Pattern:

$p ::=$	v	pattern variable
$\mid$	$C\ p_1\ \cdots\ p_n$	constructor pattern

Relation between Object and Meta Symbols:

$bind$	$::= f \Rightarrow g$	binding between object and meta functions

Fig. 2 The Core Language

$$
\begin{array}{ll}
\mathcal{E}\llbracket e \rrbracket & :: \ \mathbf{Env} \rightarrow [\mathbf{Val}] \\
 & \\
\mathcal{E}\llbracket v \rrbracket \rho & = [\rho \ v] \\
\mathcal{E}\llbracket n \rrbracket \rho & = [n] \\
\mathcal{E}\llbracket \lambda v.\, e \rrbracket \rho \ e' & = \mathcal{E}\llbracket e \rrbracket \ (\rho \oplus \{v \mapsto e'\}) \\
\mathcal{E}\llbracket e_1 \ e_2 \rrbracket \rho & = [e'_1 \ e'_2 \mid e'_1 \leftarrow \mathcal{E}\llbracket e_1 \rrbracket \rho, \ e'_2 \leftarrow \mathcal{E}\llbracket e_2 \rrbracket \rho] \\
\mathcal{E}\llbracket \textbf{case} \ e \ \textbf{of} \ p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \rrbracket \rho & = [\textbf{case} \ e' \ \textbf{of} \ p_1 \rightarrow e'_1 \ldots; p_n \rightarrow e'_n \\
 & \quad \mid e' \leftarrow \mathcal{E}\llbracket e \rrbracket \rho, e'_1 \leftarrow \mathcal{E}\llbracket e_1 \rrbracket \rho, \ldots, e'_n \leftarrow \mathcal{E}\llbracket e_n \rrbracket \rho] \\
\mathcal{E}\llbracket < e > \rrbracket \rho & = [< \rho \ e >] \\
\mathcal{E}\llbracket < e >^{\$} \rrbracket \rho & = [\rho \ e] \\
\mathcal{E}\llbracket \lambda^m < e_p > .\, e_b \rrbracket \rho \ < e_t > & = \sqcup \ [\mathcal{E}\llbracket e_b \rrbracket \ (\rho \oplus \phi) \mid \phi \leftarrow \mathit{matching} \ e_p \ (\mathit{reduce} \ e_b)] \\
\mathcal{E}\llbracket \textbf{letm} \ e_p = e_b \Leftarrow e_c \ \textbf{in} \ e \rrbracket \rho & = \sqcup[\mathcal{E}\llbracket e \rrbracket \ (\rho \oplus \phi) \mid \phi \leftarrow \mathit{matching} \ e_p \ (\mathit{reduce} \ (\rho \, e_t)), \\
 & \qquad \qquad \mathcal{E}\llbracket e_c \rrbracket \ (\rho \oplus \phi) == [\mathit{True}]] \\
\mathcal{E}\llbracket \textbf{casem} \ e \ \textbf{of} \ e_{p_1} \rightarrow e_1; \ldots; e_{p_n} \rightarrow e_n \rrbracket \rho & = \sqcup \ [\ \mathcal{E}\llbracket e_1 \rrbracket \ (\rho \oplus \phi_1) \mid \phi_1 \leftarrow \mathit{matching} \ e_{p_1} \ (\mathit{reduce} \ (\rho \, e))] \ +\!\!+ \\
 & \qquad \cdots \ +\!\!+ \\
 & \qquad [\mathcal{E}\llbracket e_n \rrbracket \ (\rho \oplus \phi_n) \mid \phi_n \leftarrow \mathit{matching} \ e_{p_n} \ (\mathit{reduce} \ (\rho \, e))]]
\end{array}
$$

Fig. 3 The Semantics of the Core Expressions

— Object Expression —

$$\frac{\Gamma(x) = \sigma}{\Gamma \ \vdash \ x :: \sigma}\ \textsf{variable} \qquad \frac{}{\Gamma \ \vdash \ n :: \mathit{baseType}(n)}\ \textsf{constant}$$

$$\frac{\Gamma \oplus \{x \mapsto \sigma_1\} \ \vdash \ e :: \sigma_2}{\Gamma \ \vdash \ \lambda x.\, e :: \sigma_1 \to \sigma_2}\ \textsf{abstraction} \qquad \frac{\Gamma \ \vdash \ e_1 :: \sigma_1 \to \sigma_2 \quad \Gamma \ \vdash \ e_2 :: \sigma_1}{\Gamma \ \vdash \ e_1\ e_2 :: \sigma_2}\ \textsf{application}$$

$$\frac{\begin{array}{l}\Gamma \ \vdash \ e :: \sigma \\ \Gamma \oplus \cup_{v_{1i} \in FV(p_1)} \{v_{1i} \mapsto \sigma_{1i}\} \ \vdash \ p_1 :: \sigma, \ e_1 :: \sigma' \\ \cdots \\ \Gamma \oplus \cup_{v_{ni} \in FV(p_n)} \{v_{ni} \mapsto \sigma_{ni}\} \ \vdash \ p_n :: \sigma, \ e_n :: \sigma'\end{array}}{\Gamma \ \vdash \ \textbf{case}\ e\ \textbf{of}\ p_1 \to e_1; \cdots, p_n \to e_n :: \sigma'}\ \textsf{case}$$

— Meta Expression —

$$\frac{\Gamma \ \vdash \ e :: \sigma}{\Gamma \ \vdash \ < e > :: < \sigma >}\ \textsf{code} \qquad \frac{\Gamma \ \vdash \ e :: < \sigma >}{\Gamma \ \vdash \ e^{\$} :: \sigma}\ \textsf{decode}$$

$$\frac{\Gamma \oplus \cup_{v_i \in FV(e_p)} \{v_i \mapsto \sigma_{pi}\} \ \vdash \ e_p :: < \sigma_1 >, \ e_b :: \sigma_2}{\Gamma \ \vdash \ \lambda^m e_p.\, e_b :: < \sigma_1 > \to \sigma_2}\ \textsf{metaAbstraction}$$

$$\frac{\begin{array}{l}\Gamma \ \vdash \ e_b :: \sigma \\ \Gamma \oplus \cup_{v_i \in FV(e_p)} \{v_i \mapsto \sigma_{pi}\} \ \vdash \ e_p :: \sigma, \ e_c :: \mathit{Bool}, \ e :: \sigma'\end{array}}{\Gamma \ \vdash \ \textbf{letm}\ e_p = e_b \Leftarrow e_c\ \textbf{in}\ e :: \sigma'}\ \textsf{letm}$$

$$\frac{\begin{array}{l}\Gamma \ \vdash \ e :: \sigma \\ \Gamma \oplus \cup_{v_{1i} \in FV(e_{p_1})} \{v_{1i} \mapsto \sigma_{1i}\} \ \vdash \ e_{p_1} :: \sigma, \ e_1 :: \sigma' \\ \cdots \\ \Gamma \oplus \cup_{v_{ni} \in FV(e_{p_n})} \{v_{ni} \mapsto \sigma_{ni}\} \ \vdash \ e_{p_n} :: \sigma, \ e_n :: \sigma'\end{array}}{\Gamma \ \vdash \ \textbf{casem}\ e\ \textbf{of}\ e_{p_1} \to e_1; \cdots, e_{p_n} \to e_n :: \sigma'}\ \textsf{casem}$$

Fig. 4 Typing Rules