First-Class Polyvariant Functions and Co-Arity Raising

Peter Thiemann

Universität Freiburg
thiemann@informatik.uni-freiburg.de

Abstract. In partial evaluation, a polyvariant function is specialized to different specialized versions, depending on the calling context. Usually, polyvariance is limited to named, top-level functions. We lift this restriction in a partial evaluator for a higher-order functional languages to obtain first-class polyvariant functions. These functions are partially static and specialize to a memoization table.
Arity raising splits a partially static parameter of a function into its dynamic components. Co-arity raising is the dual to arity raising. It splits a partially static result of a function into its dynamic components. A specializer that performs co-arity raising can transform a function returning one result into a function that returns any number of results, including zero. This technique enables polyvariant functions to return partially static results.
We demonstrate the use of these techniques in combination by specializing a typed interpreter for a simply-typed applied lambda calculus so that no run-time tagging operations remain in the specialized program.
The main technical contributions are the specification of a generalized binding-time analysis for polyvariant functions and co-arity raising, a structural operational semantics for the corresponding specializer, and a type soundness proof of the analysis with respect to the specializer.

1 Introduction

Mogensen [18] has argued that the development of partial evaluators has progressed in an evolutionary manner. Programs specialized by early systems have been subjected to many limitations, which were inherited from the program under specialization (the source program). Examples of these limitations are the number of functions in the specialized program, the number of parameters to functions, and the complexity of types. These limitations have been addressed by polyvariant specialization [3, 15], which generates different specialized variants of source functions, by arity raising [21] and partially static structures [19], and by type and constructor specialization [5, 12, 20].

Hughes's type specializer [13] (HTS) is probably the most radical implementation of all these features. Unfortunately, it is also the least well-understood partial evaluation technique. In addition, the current implementation suffers from some inefficiencies because it is based on non-deterministic type inference rules and performs a certain amount of backtracking.

In return for this sophistication, HTS solve Jones's optimal type specialization problem (TSP) [14]: In a typed language, specialize a self-interpreter with respect to a program so that the specialized program is a renaming of the original one. Such a self-interpreter must encode the typed values of the language using a universal type. The challenge is to remove this encoding completely.

Meanwhile, other researchers have attacked the problem using projections and type-based postprocessing [4,17,22]. These works have identified the removal of run-time tags as the central problem. Hence, it is not clear, which features of HTS are *required* to solve the TSP. The present paper argues that first-class polyvariant functions and co-arity raising are sufficient to solve the TSP.

Contribution We have designed and implemented a partial evaluator for Scheme (as part of the PGG system [25]) that encorporates first-class polyvariant functions and co-arity raising. We specify its behavior using a structural operational semantics. We define a generalized binding-time analysis for the specializer and prove type soundness with respect to the operational semantics. We illustrate the use of this extension with tag removal.

The paper starts from an interpreter for a typed language and discusses its binding-time properties in a standard, monovariant analysis. Next, we introduce polyvariant functions, reexamine the binding-time properties of the interpreter, and discuss some implementation issues. Then we consider co-arity raising, its effect on the binding times in the interpreter, and its implementation. Section 5 provides the theoretical framework for specifying polyvariance and co-arity raising and proves type soundness with respect to the binding-time analysis. The appendix provides a glossary of partial evaluation terms.

Related work First-class polyvariance has first been implemented in HTS [12]. It specializes a single value of arbitrary type into a type-indexed collection of specialized variants of the value. HTS uses the static part of the *entire type* for indexing. In our approach, the specialized value must be a function and the index is computed from the argument types, only. In consequence, our specializer can represent dynamic functions with (partially) static parameters and results as a type-indexed collection of dynamic functions, *i.e.*, a memoization table.

Co-arity raising is dual to arity raising. Whereas arity raising [21] splits a partially static argument into its dynamic components, co-arity raising splits a partially static result into its dynamic components. Thus, it enables a memoized, polyvariant function to return partially static results. This removes a standard limitation of partial evaluators. (Hughes [13] achieves a similar effect using a postprocessing step that generalizes arity raising to tuple removal.)

Hornof and Noyé [11] define a binding-time analysis for imperative languages that is flow, context, and return sensitive. Flow sensitivity only makes sense for imperative languages (it assign different binding times to each use of a variable). Context sensitivity corresponds to polyvariant binding time analysis. But return sensitivity means that a different binding-time description is computed for the side-effects and the return value of a function. The authors write “Return sensitivity is not applicable for functional languages since pure functions have a returns value but do not contain side-effects.” The present work demonstrates that return sensitivity also makes sense for pure functional languages.

Thiemann [23] has given an interpretation of partial evaluation in terms of type theory. While that work required an extensive dependently typed system to model standard partial evaluation for the lambda calculus, the system presented in this work is conceptually much simpler. It replies only on singleton types [1,10] to model a much extended specialization framework.

2 A typed interpreter

Figure 1 shows an interpreter for a typed call-by-value lambda calculus with constants, conditionals, and primitive operations in ML notation. The entry point is the function `evalExp : exp -> [ident * univ] -> univ`.

Monovariant binding-time analysis finds that the function arrows are static, the expression of type `exp` is static, and the environment is partially static:

$$\texttt{evalExp : }\overline{\texttt{exp}}\ \overline{\texttt{->}}\ \overline{\texttt{[ident}}\ \overline{\texttt{*}}\ \texttt{univ}\overline{\texttt{]}}\ \overline{\texttt{->}}\ \texttt{univ}$$

where underlining (overlining) in types and expressions indicates dynamic (static) constructs. But what about the type `univ`? Since we do not wish the partial evaluator to actually evaluate the input expression, the integer and the function component of `univ` must be dynamic. The well-formedness criterion of binding-time analysis forces the arguments to this function to dynamic, too.

$$\texttt{type univ = Int of }\underline{\texttt{int}}\texttt{ | Fun of }\underline{\texttt{univ}}\ \underline{\texttt{->}}\ \underline{\texttt{univ}}$$

The evaluation of function application, `App (fe, ae)`, passes the value of `ae`, `valofae : univ`, to the untagged value of the function, `exFun valoffe`. Hence, `univ` is dynamic throughout the interpreter, giving rise to the following type:

$$\texttt{evalExp : }\overline{\texttt{exp}}\ \overline{\texttt{->}}\ \overline{\texttt{[ident}}\ \overline{\texttt{*}}\ \underline{\texttt{univ}}\overline{\texttt{]}}\ \overline{\texttt{->}}\ \underline{\texttt{univ}}$$

With this annotation, partial evaluation of

```
evalExp (App (Lambda ("x", Var "x"), Const 5)) []
```

yields the unsatisfactory answer (all tagging operations remain)

```
  let valoffe = Fun (fun x -> x) in
  let valofae = Int 5 in
  exFun valoffe valofae
```

3 Introducing first-class polyvariance

First-class polyvariance overcomes the well-formedness restriction. Instead of representing functions in the source language by functions in the target language, we represent them by polyvariant functions.

```
type ident = string
type exp =
    Var of ident
  | Const of int
  | If of exp * exp * exp
  | Lambda of ident * exp
  | App of exp * exp
  | Op of ident * exp list

type univ =
    Int of int
  | Fun of univ -> univ

let exInt (Int i) = i
let exFun (Fun f) = f

let rec evalExp e env =
  match e with
    Var s ->
      applyEnv env s
  | Const i ->
      Int i
  | If (ie, te, ee) ->
      let valofie = evalExp ie env in
      if exInt valofie = 1
      then evalExp te env
      else evalExp ee env
  | Lambda (s, e) ->
      Fun (fun x -> evalExp e (extendEnv env s x))
  | App (fe, ae) ->
      let valoffe = evalExp fe env in
      let valofae = evalExp ae env in
      exFun valoffe valofae
  | Op (s, es) ->
      applyOp s (evalExps es env)
and evalExps es env =
  match es with
    [] -> []
  | e :: es -> evalExp e env :: evalExps es env
and applyOp o valofes =
  match (o, valofes) with
    ("+", [Int i1; Int i2]) -> Int (i1+i2)
  | ("-", [Int i1; Int i2]) -> Int (i1-i2)
  | ("*", [Int i1; Int i2]) -> Int (i1*i2)
  | ("/", [Int i1; Int i2]) -> Int (i1/i2)
```

Fig. 1. Typed interpreter

Polyvariant functions A polyvariant function, `poly x -> e`, specializes to a table of functions *and* gives rise to a partially static function value. The table contains one variant for each application of the function value to different static arguments. At each application, the specialized code fetches the appropriate variant from the table and applies it to the argument. In contrast to a dynamic function, it can be applied to static arguments and it can give static results.

For example, the annotated program

```
let f = poly (x, y) ->  lift x + y in
fun y -> (f (3, y), f (4, y))
```

specializes to

```
let poly_1 = (fun y -> 3 + y, fun y -> 4 + y) in
fun y -> (fst poly_1 y, snd poly_1 y)
```

Coming back to our typed interpreter, we revise the interpreter as follows:

```
type univ = Int of int | Fun of univ => univ    (* => is polyvariant function *)

let rec evalExp e env = match e with
    ...
  | Lambda (s, e) ->
      Fun (poly x -> evalExp e (extendEnv env s x))
```

Unfortunately, this modification does not solve the problem, yet. Since the body of each polyvariant function is still specialized to code, its return value must be code, by the well-formedness restriction. Since every expression can occur in the body of the lambda, the datatype `univ` remains dynamic.

Application to cps-interpretation Before we shall see (in the next section) that co-arity raising attacks exactly this restriction, let us first remark that we could consider our job done at this point. Suppose we had started with an interpreter in continuation-passing style. All functions in a cps-interpreter return values of a fixed result type, for example, `int`. This return value would be dynamic but continuations and functions could be represented by polyvariant functions.

Figure 2 shows an excerpt from the binding-time analyzed CPS-interpreter. The annotated type of the interpreter is

```
cevalExp : exp -> [ident * cuniv] -> (cuniv => answer) => answer
```

With this annotation, partial evaluation of

```
cevalExp (App (Lambda ("x", Var "x"), Const 5)) [] (poly x -> exCInt x)
```

yields the answer

```
type answer = int
type cuniv =
    CInt of int
  | CFun of cuniv * (cuniv => answer) => answer

let rec cevalExp e env = fun c ->
  match e with
    ...
  | Lambda (s, e) ->
      c (CFun (poly (x, c) -> cevalExp e (extendEnv env s x) c))
  | App (fe, ae) ->
      cevalExp fe env
        (poly valoffe ->
          cevalExp ae env
            (poly valofae ->
              exCFun valoffe (valofae, c)))
```

Fig. 2. Annotated cps-interpreter

```
let poly_1 = fun x -> x in      (* poly x -> exCInt x *)
let poly_2 = fun valoffe ->
  let poly_4 = fun valofae -> valoffe (valofae, poly_1) in
  poly_4 5
let poly_3 = fun (x, c) -> c x in
poly_2 poly_3
```

where each poly-table contains exactly one element. All type tags have disappeared, which is quite satisfactory. Unfortunately, the specialized program is now in continuation-passing style, which is not acceptable.

Implementation of polyvariance The execution of a poly-lambda `poly x -> e` generates a named code stub for a sequence of dynamic values and at the same time returns a partially static value. The value consists of the static closure of `fun x -> e`, a static pointer to a freshly allocated, empty memoization table, and the dynamic name for the sequence of values. Here, a closure is a code pointer and a vector of the values of the free variables. While a standard memo table maps static values to names of top-level functions, our memo table maps static values to indices into the sequence of values.

Polyvariance of polyvariance Since polyvariant functions are first-class, they can be parameters and results or other polyvariant functions. Therefore, memo tables may contain memo tables and we have to clarify what is the static part of a memo table and how to compare two memo tables.

Suppose $M_1 = (c_1, m_1, d_1)$ and $M_2 = (c_2, m_2, d_2)$ are two values representing memo tables, where c_i is the function closure, m_i is the reference to the static table, and d_i is the dynamic reference to the table. Clearly, if $m_1 = m_2$ for the static references then $c_1 = c_2$ and the two tables are equal.

If $m_1 \neq m_2$ but $c_1 = c_2$ (this can happen if the same `poly` is evaluated multiple times), then M_1 and M_2 should be considered equal to avoid useless code duplication in the code using M_1 and M_2. However, table m_1 may contain entries that are not present in m_2 and vice versa. In addition, the order of entries may be different in m_1 and m_2. Hence, to equate the two tables, we must

- generate in m_1 the entries present in m_2 but not in m_1;
- generate in m_2 the entries present in m_1 but not in m_2;
- reorder one of the tables so that the sequence of entries in both tables is identical. This involves changing already generated code (or delaying its generation) for creating the dynamic tables and their accesses.

The resulting tables m_1' and m_2' have the same index set and the same ordering, but a new entry in m_1' generates code in a different dynamic table than a new entry in m_2'. In consequence, the representation contains not just one closure, but rather a list of closures. While the closures have identical static behavior, they generate different dynamic code in different contexts.

This merge operation is also present in HTS. Since polyvariance in HTS is more general, its implementation unifies entries before they are fully known. Hence, it sometimes has to backtrack and retry. In contrast, our system does not require backtracking because the keys into the memo tables are always known.

4 Co-arity raising

Arity raising is the process of transforming a partially static argument into zero or more dynamic arguments. During specialization, it happens at every memoization point. The specializer splits its partially static argument list into a static part and its dynamic components. It uses the static part as the key for its memoization table and generates the body of the specialized function from the static part and fresh variables substituted for the dynamic components.

Usually, the binding-time analysis ensures that the result of the body of a specialized function is dynamic. Co-arity raising makes this restriction obsolete. The specializer applies the splitting procedure to the partially static value resulting from specializing the body, thus splitting it into a static part and dynamic parts. The static part must be returned to each call-site of the specialized function at specialization-time. Hence, the static part must be stored in the memoization table. The dynamic parts are transmitted in a tuple. The code generated at the call-site must be equipped to dissect this tuple again. Subsequent uses of the same specialized function just extract the return value from the memoization table and generate the corresponding function call.

The interpreter, finally The remaining problem in Sec. 3 was that the return value of a polyvariant function had to be code. Using co-arity raising, this

```
exInt (let valoffe = Fun (poly x -> x) in
       let valofae = Int (lift 5) in
       exFun valoffe_valofae)
==
let poly_1 = (fun x -> x) in
poly_1 (lift 5)
```

Fig. 3. Example with co-arity raising

is no longer necessary. The type `univ` can be equipped with the binding-time decoration

```
type univ = Int of int | Fun of univ => univ
```

Looking back at the code of the interpreter in Fig. 1, it means that all applications of `Int`, `Fun`, `exInt`, and `exFun` can be classified static, now.

The result of partially evaluating `exInt (evalExp (App (Lambda ("x", Var "x"), Const 5)) [])` is shown in Fig. 3.

Discussion The specializer can only use a function with partially static result *after* its first specialization is complete. The specializer checks if an entry in a memo table has its return value before it uses it. Otherwise, it stops with an error message. A similar phenomenon occurs in a specializer with static state [26].

Many specializers introduce memoization points at dynamic conditionals [2] to increase the chance of termination. Usually, dynamic conditionals and memoization points both return dynamic values. With co-arity raising, a dynamic conditional returning only a dynamic value effectively destroys all information.

With continuation-based partial evaluation [16], a dynamic conditional can return arbitrary values, including static ones. The specializer processes the context of the conditional once for the result of each branch. But suppose there is a memoization point in the context: the body of

```
poly x -> if x = 0 then 1 else 2
```

returns once with static value `1` and once with static value `2`. Which of the values must be put into the result part of the `poly`'s table? Both or none?

Putting both values into the memoization table means that the specialized function must have more than one exit. Consequently, specialized programs must be in continuation-passing style with multiple continuations. Moreover, due to nesting of dynamic conditionals, an expression can have an arbitrary (potentially infinite) number of different (partially) static return values. This is not viable.

Alternatively, the specializer compares the results of the branches and registers one of them in the table only if their static parts are equal. Otherwise, it signals an error. The dynamic parts, which are not available at specialization-time, can still be different.

Our specializer implements the latter behavior like HTS. In contrast to HTS, which unifies the types of the branches, our specializer compares readily computed values. It detects failure immediately, but it cannot transmit information between the branches since values only flow forward.

5 Formalization

For reasons of space, the exposition starts directly with an annotated two-level language. The source language, a call-by-value lambda calculus enriched with numbers and constructed data, and its semantics and be extracted from the semantics of the annotated language without many problems.

Language Figure 4 defines the type structure of the language. The usual terms for binding-time annotated types are used as *kinds* in our framework. A kind is a pair $(\sigma; b)$ where σ is a raw kind and b is a binding time. Raw kinds are `int`, functions, and sum-of-product kinds.

The type language has static types that carry information about a value and dynamic types that do not. Static types comprise type variables, singleton types $\overline{\mathtt{m}}$ for integer constants m, singleton function types $S\{\lambda x.e\}$, a type $\overline{o}(\tau_1 \dots \tau_n)$ denoting the application of primitive operator o on types $\tau_1 \dots \tau_n$, a type $\overline{c}(\tau_1 \dots \tau_n)$ denoting the application of data constructor c to types $\tau_1 \dots \tau_n$, the type $[\tau_1 \to \tau_1', \dots, \tau_n \to \tau_n']$ of a polyvariant function with n variants, and a type for the result of a `case` expression $\mathtt{case}\ \tau\ \mathtt{of}\ c_i(\alpha_{i1} \dots \alpha_{in_i})\,\mathtt{=>}\,\tau_i \dots$. Dynamic types comprise the standard integer type, standard function types, and the sum-of-product type $\sum c_i(\tau_{i1} \dots \tau_{in_i})$.

The well-formedness criterion of type-based binding-time analysis [6] is expressed as a kinding judgement $\Delta, \Gamma \vdash \tau : \rho$. The rules themselves (Fig. 4) are mostly unsurprising. Due to the singleton type for functions, the judgement is mutually recursive with the one for typing expressions and consequently depends on a typing context Γ. The rule for polyvariant function types assumes that the functions are static. There is no rule for dynamic polyvariant function types; it would be identical to the standard function type.

The formalization of type equality $\Delta, \Gamma \vdash \tau = \tau$ enables reduction of non-primitive types like $\overline{o}$-types and `case`-types to simpler forms. These rules are applicable in any context. For type variables, the kinding judgement can provide information about the type. The equality rule for singleton types is much weaker than similar rules in the literature [1, 10].

Annotated expressions carry their binding time on each construct, except `lift` that converts a static integer into a dynamic one, and `poly` and `spec` that are assumed static. The `poly` term carries all argument types as an annotation. This is a technical device to achieve subject reduction. Figure 5 shows the typing rules for the expression judgement $\Delta, \Gamma \vdash e : \tau$. The rules for the dynamic types are the standard ones, except for the rule (*d-case*): The result type of a dynamic conditional is unrestricted, requiring a continuation-based specializer [16].

The static rules are similar to the ones for HTS, with subtle differences. The rule (*top*) allows to give a static conditional a type if the chosen branch has a type. The rule (*s-abs*) defers typing of the abstraction to the point where the function is applied as shown in (*a-app*). The rule (*s-poly*) types the lambda abstraction for all requested argument types. The rules (*s-prim*), (*s-con*), (*s-case*) reflect the application of the primitive (data constructor, case distinction) on the type level. The rule (*conv*) allows type conversions everywhere in a derivation.

Results To define a structural operational semantics, we need to introduce new terms that denote intermediate stages of the computation. Figure 8 summarizes those terms and their typing rules. Figure 6 defines the reduction relation and Figure 7 defines the splitting relation that takes partially static values apart. The judgement $v \stackrel{\gamma}{\leadsto} v'$ means that v' is v with all dynamic component replaced by variables and γ is a substitution so that $v = \gamma(v')$. As it stands, the reduction rule for `poly` non-deterministically guesses which variants are needed. This is a simplification with respect to the actual implementation that generates the required variants on demand.

Our type soundness proof follows the syntactic approach of Wright and Felleisen [27] as refined by Harper [7].

Definition 1. *For an expression e, $free(e)$ is the set of free variables of e. Its usual definition extends to the non-standard expressions as follows:*

$$\begin{array}{ll}
free(\texttt{poly}^{\tau_1\cdots\tau_n}\ x.e) & = free(e) \setminus \{x\} \\
free(\texttt{spec}\ e_1 @ e_2) & = free(e_1) \cup free(e_2) \\
free(\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle)) & = \{x\} \cup \bigcup_i (free(a_i) \setminus dom(\Gamma_i)) \\
free(\texttt{work}\langle \Gamma_i, v_i \mapsto e_i \rangle) & = \bigcup_i (free(e_i) \setminus dom(\Gamma_i))
\end{array}$$

Definition 2. *An environment Γ is* dynamically closed *with respect to Δ if, for all $x \in dom(\Gamma)$, $\Delta, \Gamma \vdash \Gamma(x) : (\sigma, D)$. Write $\Delta \vdash_d \Gamma$ in this case.*

Proposition 1 (Subject reduction).
Suppose $\Delta, \Gamma \vdash e : \tau$ and, for all $x \in free(e)$, $\Delta, \Gamma \vdash \Gamma(x) : (\sigma, D)$.
If $e \longrightarrow e'$ then $\Delta, \Gamma \vdash e' : \tau$.

Proof. If $\tau = \top$ then there is nothing to prove. Hence, assume that $\tau \neq \top$. The proof is by induction on the definition of $\longrightarrow$.

Case $(\lambda^S x.e)@^S v \longrightarrow e[x \mapsto v]$.

Since $\Delta, \Gamma \vdash (\lambda^S x.e)@^S v : \tau$, rule (*s-app*) indicates that $\Delta, \Gamma \vdash \lambda^S x.e : S\{\lambda x'.e'\}$ and $\Delta, \Gamma \vdash v : \tau_2$ and $\Delta, \Gamma[x : \tau_2] \vdash e : \tau$. By rule (*s-abs*), it must be that $x = x'$ and $e = e'$. The substitution lemma 3 implies that $\Delta, \Gamma \vdash e[x \mapsto v] : \tau$, as required.

Case $\texttt{if}^S\ 1\ e_2\ e_3 \longrightarrow e_2$.

Since $\Delta, \Gamma \vdash \texttt{if}^S\ 1\ e_2\ e_3 : \tau$, rule (*s-if*) requires that $\Delta, \Gamma \vdash 1 : \tau_1$, $\Delta, \Gamma \vdash e_2 : \tau_2$, $\Delta, \Gamma \vdash e_3 : \tau_3$, $\Delta \vdash \tau_1 : (\texttt{int}; S)$, and $\tau = \texttt{if}\ \tau_1\ \tau_2\ \tau_3$. By rule (*const*), it must be that $\tau_1 = \overline{1}$. Hence, $\Delta \vdash \tau = \texttt{if}\ \overline{1}\ \tau_2\ \tau_3$, which is equivalent to $\Delta \vdash \tau = \tau_2$. In

Types τ, pre-kinds σ, kinds ρ, and binding times b

$$\begin{array}{rl}
\tau ::= & \alpha \mid \overline{\mathtt{m}} \mid S\{\lambda x.e\} \mid \overline{o}(\tau \ldots \tau) \mid \overline{c}(\tau \ldots \tau) \mid [\tau \to \tau, \ldots, \tau \to \tau] \mid \\
& \mathtt{int} \mid \tau \to \tau \mid \sum c(\tau \ldots \tau) \mid \mathtt{case}\ \tau\ \mathtt{of}\ \ c(\alpha \ldots \alpha)\ \mathtt{=>}\ \tau \ldots \\
\sigma ::= & \mathtt{int} \mid \rho \to \rho \mid \sum c(\rho \ldots \rho) \\
\rho ::= & (\sigma; b) \\
b ::= & S \mid D \qquad \text{where} \qquad S < D
\end{array}$$

Kinding $\Delta, \Gamma \vdash \tau : \rho$

$$\Delta, \Gamma \vdash \overline{\mathtt{m}} : (\mathtt{int}; S) \qquad \Delta, \Gamma \vdash \mathtt{int} : (\mathtt{int}; D) \qquad \frac{\Delta(\alpha) = \rho}{\Delta, \Gamma \vdash \alpha : \rho}$$

$$\frac{\Delta, \Gamma \vdash \tau : \rho \quad \Delta, \Gamma \vdash \tau' : \rho' \quad \Delta, \Gamma[x : \tau] \vdash e : \tau'}{\Delta, \Gamma \vdash S\{\lambda x.e\} : (\rho \to \rho'; S)} \qquad \frac{\Delta, \Gamma \vdash \tau_i : \rho \quad \Delta, \Gamma \vdash \tau_i' : \rho'}{\Delta, \Gamma \vdash [\tau_i \to \tau_i'] : (\rho \to \rho'; S)}$$

$$\frac{\Delta, \Gamma \vdash \tau_1 : \rho_1 \quad \Delta, \Gamma \vdash \tau_2 : \rho_2 \quad \rho_i = (\sigma_i; D)}{\Delta, \Gamma \vdash \tau_1 \to \tau_2 : (\rho_1 \to \rho_2; D)}$$

$$\frac{\begin{array}{c}(\exists i)\ c = c_i \quad n = n_i \\ \Delta, \Gamma \vdash \tau_1 : \rho_{i1} \quad \ldots \quad \Delta, \Gamma \vdash \tau_n : \rho_{in}\end{array}}{\Delta, \Gamma \vdash \overline{c}(\tau_1 \ldots \tau_n) : (\sum c_i(\rho_{i1} \ldots \rho_{in_i}); S)} \qquad \frac{\Delta, \Gamma \vdash \tau_{ij} : \rho_{ij} \quad \rho_{ij} = (\sigma_{ij}; D)}{\Delta, \Gamma \vdash \sum c_i(\tau_{i1} \ldots \tau_{in_i}) : (\sum c_i(\rho_{i1} \ldots \rho_{in_i}); D)}$$

$$\frac{\Delta, \Gamma \vdash \tau_1 : (\mathtt{int}; S) \quad \ldots \quad \Delta, \Gamma \vdash \tau_n : (\mathtt{int}; S)}{\Delta, \Gamma \vdash \overline{o}(\tau_1 \ldots \tau_n) : (\mathtt{int}; S)}$$

$$\frac{\Delta, \Gamma \vdash \tau_0 : (\sum c_i(\rho_{i1} \ldots \rho_{in_i}); S) \quad \Delta[\alpha_{ij} : \rho_{ij}], \Gamma \vdash \tau_i : \rho}{\Delta, \Gamma \vdash \mathtt{case}\ \tau_0\ \mathtt{of}\ \ c_i(\alpha_{i1} \ldots \alpha_{in_i})\ \mathtt{=>}\ \tau_i \ldots : \rho}$$

Type equality $\Delta, \Gamma \vdash \tau = \tau$

$$\frac{\Delta, \Gamma \vdash \tau = \tau'}{\Delta, \Gamma \vdash \tau' = \tau} \qquad \frac{\Delta, \Gamma \vdash \tau = \tau' \quad \Delta, \Gamma \vdash \tau' = \tau''}{\Delta, \Gamma \vdash \tau' = \tau''} \qquad \Delta, \Gamma \vdash \tau = \tau \qquad \frac{\Delta, \Gamma \vdash \tau : \Delta(\alpha)}{\Delta, \Gamma \vdash \tau = \alpha}$$

$$\frac{e_1 = e_2}{\Delta, \Gamma \vdash S\{\lambda x.e_1\} = S\{\lambda x.e_2\}} \qquad \frac{\Delta, \Gamma \vdash \tau_{1i} = \tau_{2i} \quad \Delta, \Gamma \vdash \tau_{1i}' = \tau_{2i}'}{\Delta, \Gamma \vdash [\tau_{1i} \to \tau_{1i}'] = [\tau_{2i} \to \tau_{2i}']}$$

$$\frac{\Delta, \Gamma \vdash \tau_1 = \tau_1' \quad \Delta, \Gamma \vdash \tau_2 = \tau_2'}{\Delta, \Gamma \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2'} \qquad \frac{\Delta, \Gamma \vdash \tau_i = \tau_i'}{\Delta, \Gamma \vdash \overline{o}(\tau_1 \ldots \tau_n) = \overline{o}(\tau_1' \ldots \tau_n')}$$

$$\frac{\Delta, \Gamma \vdash \tau_i = \tau_i'}{\Delta, \Gamma \vdash \overline{c}(\tau_1 \ldots \tau_n) = \overline{c}(\tau_1' \ldots \tau_n')} \qquad \frac{\Delta, \Gamma \vdash \tau_{ij} = \tau_{ij}'}{\Delta, \Gamma \vdash \sum c_i(\tau_{i1} \ldots \tau_{in_i}) = \sum c_i(\tau_{i1}' \ldots \tau_{in_i}')}$$

$$\frac{\Delta, \Gamma \vdash \tau_0 = \tau_0' \quad \Delta[\alpha_j : \rho_j], \Gamma \vdash \tau_i = \tau_i'}{\Delta, \Gamma \vdash \mathtt{case}\ \tau_0\ \mathtt{of}\ \ \ldots c_i(\alpha_1 \ldots \alpha_n)\ \mathtt{=>}\ \tau_i \ldots = \mathtt{case}\ \tau_0'\ \mathtt{of}\ \ \ldots c_i(\alpha_1 \ldots \alpha_n)\ \mathtt{=>}\ \tau_i' \ldots}$$

Conversion rules

$$\Delta, \Gamma \vdash \overline{o}(\overline{m_1} \ldots \overline{m_n}) = \overline{[\![o]\!](m_1 \ldots m_n)}$$

$$\Delta, \Gamma \vdash \mathtt{case}\ \overline{c}(\tau_1 \ldots \tau_n)\ \ \mathtt{of}\ \ \ldots c(\alpha_1 \ldots \alpha_n)\ \mathtt{=>}\ \tau \ldots = \tau[\alpha_i \mapsto \tau_i]$$

Contexts

$$\frac{}{\Delta \vdash \varepsilon\ \text{ctx}} \qquad \frac{\Delta \vdash \Gamma\ \text{ctx} \quad x \notin dom(\Gamma) \quad \Delta, \Gamma \vdash \tau : \rho}{\Delta \vdash \Gamma[x : \tau]\ \text{ctx}}$$

Fig. 4. Annotated types

Expressions $e ::= x \mid m \mid \lambda^b x.e \mid e@^b e \mid o^b(e \dots e) \mid \texttt{lift}\ e \mid$
$\texttt{poly}^{\tau\dots\tau}\ x.e \mid \texttt{spec}\ e@e \mid c^b(e \dots e) \mid \texttt{case}^b\ e\ \texttt{of}\ c(x \dots x) \texttt{=>} e \dots$

Typing rules

$$(var)\frac{\Delta \vdash \Gamma\ \text{ctx} \quad \Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau} \qquad (const)\frac{\Delta \vdash \Gamma\ \text{ctx}}{\Delta, \Gamma \vdash m : \overline{\texttt{m}}} \qquad (top)\frac{\Delta \vdash \Gamma\ \text{ctx}}{\Delta, \Gamma \vdash e : \top}$$

$$(s\text{-}abs)\frac{dom(\Gamma) \supseteq free(\lambda x.e)}{\Delta, \Gamma \vdash \lambda^S x.e : S\{\lambda x.e\}}$$

$$(d\text{-}abs)\frac{\Delta, \Gamma[x : \tau'] \vdash e : \tau \quad \Delta, \Gamma \vdash \tau' \to \tau : \rho'}{\Delta, \Gamma \vdash \lambda^D x.e : \tau' \to \tau}$$

$$(s\text{-}app)\frac{\Delta, \Gamma \vdash e_1 : S\{\lambda x.e\} \quad \Delta, \Gamma \vdash e_2 : \tau_2 \quad \Delta, \Gamma[x : \tau_2] \vdash e : \tau}{\Delta, \Gamma \vdash e_1 @^S e_2 : \tau}$$

$$(d\text{-}app)\frac{\Delta, \Gamma \vdash e_1 : \tau_2 \to \tau \quad \Delta, \Gamma \vdash e_2 : \tau_2}{\Delta, \Gamma \vdash e_1 @^D e_2 : \tau}$$

$$(s\text{-}poly)\frac{(\forall 1 \leq i \leq n)\ \Delta, \Gamma[x \mapsto \tau_i] \vdash e : \tau_i'}{\Delta, \Gamma \vdash \texttt{poly}^{\tau_1 \dots \tau_n}\ x.e : [\tau_i \mapsto \tau_i']} \qquad (s\text{-}spec)\frac{\Delta, \Gamma \vdash e_1 : [\tau_i \to \tau_i'] \quad \Delta, \Gamma \vdash e_2 : \tau_j}{\Delta, \Gamma \vdash \texttt{spec}\ e_1 @ e_2 : \tau_j'}$$

$$(s\text{-}prim)\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Delta, \Gamma \vdash e_n : \tau_n \quad \Delta, \Gamma \vdash \overline{o}(\tau_1 \dots \tau_n) : (\texttt{int}; S)}{\Delta, \Gamma \vdash o^S(e_1 \dots e_n) : \overline{o}(\tau_1 \dots \tau_n)}$$

$$(d\text{-}prim)\frac{\Delta, \Gamma \vdash e_1 : \texttt{int} \quad \dots \quad \Delta, \Gamma \vdash e_n : \texttt{int}}{\Delta, \Gamma \vdash o^D(e_1 \dots e_n) : \texttt{int}}$$

$$(lift)\frac{\Delta, \Gamma \vdash e : \tau \quad \Delta, \Gamma \vdash \tau : (\texttt{int}; S)}{\Delta, \Gamma \vdash \texttt{lift}\ e : \texttt{int}}$$

$$(s\text{-}con)\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Delta, \Gamma \vdash e_n : \tau_1}{\Delta, \Gamma \vdash c^S(e_1 \dots e_n) : \overline{c}(\tau_1 \dots \tau_n)}$$

$$(d\text{-}con)\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Delta, \Gamma \vdash e_n : \tau_n \quad \Delta, \Gamma \vdash \tau_i : (\sigma_i; D)}{\Delta, \Gamma \vdash c^D(e_1 \dots e_n) : c(\tau_1 \dots \tau_n) + \sum c_i(\dots)}$$

$$(s\text{-}case)\frac{\begin{array}{c}\Delta, \Gamma \vdash e_0 : \tau_0 \quad \Delta, \Gamma \vdash \tau_0 : (\sum c_i^S(\rho_{i1} \dots \rho_{in_i}); S) \\ \Delta[\alpha_{ij} : \rho_{ij}], \Gamma[x_{ij} : \alpha_{ij}] \vdash e_i : \tau_i \quad (1 \leq i \leq n)\end{array}}{\Delta, \Gamma \vdash \texttt{case}^S\ e_0\ \texttt{of}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} e_i \dots : \texttt{case}\ \tau_0\ \texttt{of}\ c_i(\alpha_{i1} \dots \alpha_{in_i}) \texttt{=>} \tau_i \dots}$$

$$(d\text{-}case)\frac{\Delta, \Gamma \vdash e_0 : \sum c_i(\tau_{i1} \dots \tau_{in_i}) \quad \Delta, \Gamma[x_{ij} : \tau_{ij}] \vdash e_i : \tau' \quad (1 \leq i \leq n)}{\Delta, \Gamma \vdash \texttt{case}^D\ e_0\ \texttt{of}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} e_i \dots : \tau'}$$

$$(conv)\frac{\Delta, \Gamma \vdash e : \tau \quad \Delta \vdash \tau = \tau'}{\Delta, \Gamma \vdash e : \tau'}$$

Fig. 5. Annotated expressions

Values, answers, and answer contexts

$$
\begin{array}{lll}
v, w & ::= & x \mid m \mid \lambda^S x.e \mid \texttt{lift}\ m \mid \texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle) \mid c^S(v \ldots v) \mid \\
 & & \lambda^D x.a \mid a \underline{@} a \mid \underline{\texttt{case}}\ a\ \underline{\texttt{of}}\ c_i(x_{i1} \ldots x_{in_i}) \texttt{=>} a_i \ldots \mid \\
 & & o^D(a \ldots a) \mid c^D(a \ldots a) \\
a & ::= & A[v] \\
A & ::= & [\,] \mid \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ A
\end{array}
$$

Reduction rules

$$
\begin{array}{ll}
(\lambda^S x.e) @^S v & \longrightarrow e[x \mapsto v] \\
\texttt{case}^S\ c^S(v_1 \ldots)\ \texttt{of}\ c(x_1 \ldots) \texttt{=>} e \ldots & \longrightarrow e[x_i \mapsto v_i] \\
 & \\
(\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_1) @^S e_2 & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_1 @^S e_2 \\
v_1 @^S (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_2) & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ v_1 @^S e_2 \\
c^S(v_1 \ldots (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_k) \ldots e_n) & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ c^S(v_1 \ldots e_k \ldots e_n) \\
o^S(v_1 \ldots (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_k) \ldots e_n) & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ o^S(v_1 \ldots e_k \ldots e_n) \\
\texttt{lift}\ (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e) & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ (\texttt{lift}\ e) \\
\texttt{spec}^S\ (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_1) @ e_2 & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ (\texttt{spec}^S\ e_1 @ e_2) \\
\texttt{spec}^S\ v_1 @ (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e_2) & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ (\texttt{spec}^S\ v_1 @ e_2) \\
\texttt{case}^S\ (\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e)\ \texttt{of}\ c(x_1 \ldots) \texttt{=>} e' \ldots & \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ \texttt{case}^S\ e\ \texttt{of}\ c(x_1 \ldots) \texttt{=>} e' \ldots \\
 & \\
a_1 @^D a_2 & \longrightarrow \underline{\texttt{let}}\ x = a_1 \underline{@} a_2\ \underline{\texttt{in}}\ x
\end{array}
$$

$$
\frac{(\forall 1 \le i \le n)\ v_i \overset{\gamma_i}{\leadsto} v' \quad \boldsymbol{x} = dom(\gamma_i)}{\begin{array}{l}\texttt{case}^D\ a_0\ \texttt{of}\ c_i(x_{i1} \ldots x_{in_i}) \texttt{=>} A_i[v_i] \ldots \\ \longrightarrow \underline{\texttt{let}}\ \boldsymbol{x} = \underline{\texttt{case}}\ a_0\ \underline{\texttt{of}}\ c_i(x_{i1} \ldots x_{in_i}) \texttt{=>} A_i[\gamma_i \boldsymbol{x}] \ldots\ \underline{\texttt{in}}\ v'\end{array}}
$$

$$
\frac{(\forall 1 \le i \le n. \exists v_i', \Gamma_i)\ v_i' \overset{\gamma_i}{\leadsto} v_i \quad \emptyset, \Gamma_i \vdash v_i : \tau_i \quad \emptyset \vdash_d \Gamma_i}{\texttt{poly}^{\tau_1 \ldots \tau_n}\ x.e \longrightarrow \texttt{work}\langle \Gamma_i, v_i \mapsto e[x \mapsto v_i] \rangle}
$$

$$
\frac{\forall 1 \le i \le n. \exists v_i'. v_i' \overset{\gamma_i}{\leadsto} v_i \quad \boldsymbol{x_i} = dom(\gamma_i) \quad w_i \overset{\gamma_i'}{\leadsto} w_i' \quad \boldsymbol{y_i} = dom(\gamma_i')}{\begin{array}{l}\texttt{work}\langle \Gamma_i, v_i \mapsto A_i[w_i] \rangle \\ \longrightarrow \underline{\texttt{let}}\ t = (\lambda \boldsymbol{x_1}.A_1[\gamma_1' \boldsymbol{y_1}], \ldots, \lambda \boldsymbol{x_n}.A_n[\gamma_n' \boldsymbol{y_n}])\ \underline{\texttt{in}}\ \texttt{POLY}(t, \langle \Gamma_i, v_i \mapsto A_i[w_i] \rangle)\end{array}}
$$

$$
\frac{v_j \overset{\gamma}{\leadsto} v' \quad v \overset{\gamma'}{\leadsto} v' \quad \boldsymbol{x} = dom(\gamma') \quad w_j \overset{\gamma''}{\leadsto} w \quad \boldsymbol{y} = dom(\gamma'')}{\texttt{spec POLY}(t, \langle \Gamma_i, v_i \mapsto A_i[w_i] \rangle) @ v \longrightarrow \underline{\texttt{let}}\ \boldsymbol{y} = \pi_j(t) \underline{@} \gamma'(\boldsymbol{x})\ \underline{\texttt{in}}\ w}
$$

$$
\frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']}
$$

Evaluation contexts

$$
\begin{array}{ll}
C ::= & [\,] @^b e \mid v @^S [\,] \mid a @^D [\,] \mid \texttt{case}^b\ [\,]\ \texttt{of}\ \ldots \mid c^S(v \ldots [\,] e \ldots) \mid c^D(a \ldots [\,] e \ldots) \mid \\
 & o^S(v \ldots [\,] e \ldots) \mid o^D(a \ldots [\,] e \ldots) \mid \texttt{lift}\ [\,] \mid \lambda^D x.[\,] \mid \\
 & \texttt{case}^D\ a\ \texttt{of}\ c(x \ldots x) \texttt{=>} [\,] \mid \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ [\,] \mid \\
 & \texttt{spec}\ [\,] @ e \mid \texttt{spec}\ v @ [\,] \mid \texttt{work}\langle \Gamma, v \mapsto a; \ldots; \Gamma, v \mapsto [\,]; \Gamma, v \mapsto e; \ldots \rangle
\end{array}
$$

Fig. 6. Operational semantics of two-level language

Splitting of answers

$$\frac{a \overset{\gamma}{\leadsto} a'}{\underline{\texttt{let}}\ x = a''\ \underline{\texttt{in}}\ a \overset{\gamma}{\leadsto} \underline{\texttt{let}}\ x = a''\ \underline{\texttt{in}}\ a'}$$

$$x \overset{[x' \mapsto x]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$m \overset{[]}{\leadsto} m$$

$$\lambda^D x.e \overset{[x' \mapsto \lambda^D x.e]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$v_1 @^D v_2 \overset{[x' \mapsto v_1 @^D v_2]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$o^D(v_1 \dots v_n) \overset{[x' \mapsto o^D(v_1 \dots v_n)]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$c^D(v_1 \dots v_n) \overset{[x' \mapsto c^D(v_1 \dots v_n)]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$\texttt{case}^D\ v\ \texttt{of}\ c(x \dots x)\ \texttt{=>}\ v \dots \overset{[x' \mapsto \texttt{case}^D\ v\ \texttt{of}\ c(x \dots x)\ \texttt{=>}\ v \dots]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$\frac{\{x_1 \dots x_n\} = \mathit{free}(\lambda^S x.e) \quad x_i' \text{ fresh}}{\lambda^S x.e \overset{[x_i' \mapsto x_i]}{\leadsto} \lambda^S x.e[x_i \mapsto x_i']}$$

$$\texttt{lift}\ m \overset{[x' \mapsto \texttt{lift}\ m]}{\leadsto} x' \text{ where } x' \text{ fresh}$$

$$\frac{\{x, x_1, \dots, x_n\} = \mathit{free}(\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle)) \quad x' \text{ fresh}}{\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle) \overset{[x' \mapsto x, x_i \mapsto x_i']}{\leadsto} \texttt{POLY}(x', \langle \Gamma_i, v_i \mapsto a_i[x_i \mapsto x_i'] \rangle)}$$

$$\frac{v_i \overset{\gamma_i}{\leadsto} v_i' \quad (1 \le i \le n) \quad i \ne j \Rightarrow \mathit{dom}(\gamma_i) \cap \mathit{dom}(\gamma_j) = \emptyset}{c^S(v_1 \dots v_n) \overset{\gamma_1 \cup \dots \cup \gamma_n}{\leadsto} c^S(v_1' \dots v_n')}$$

Splitting of typed values

$$\frac{\Delta, \Gamma \vdash a'' : \tau'' \quad \Delta, \Gamma[x : \tau''] \vdash a : \tau \overset{\gamma}{\leadsto} \Delta, \Gamma' \vdash a' : \tau}{\Delta, \Gamma \vdash \underline{\texttt{let}}\ x = a''\ \underline{\texttt{in}}\ a : \tau \overset{\gamma}{\leadsto} \Delta, \Gamma' \vdash \underline{\texttt{let}}\ x = a''\ \underline{\texttt{in}}\ a' : \tau}$$

$$\Delta, \Gamma \vdash v : \overline{\texttt{m}} \overset{[]}{\leadsto} \emptyset, \emptyset \vdash v : \overline{\texttt{m}}$$

$$\Delta, \Gamma \vdash v : \overline{o}(\tau_1 \dots \tau_n) \overset{[]}{\leadsto} \emptyset, \emptyset \vdash v : \overline{o}(\tau_1 \dots \tau_n)$$

$$\Delta, \Gamma \vdash v : \texttt{int} \overset{[x' \mapsto v]}{\leadsto} \emptyset, [x' : \texttt{int}] \vdash x' : \texttt{int}$$

$$\frac{\{x_1, \dots, x_n\} = \mathit{free}(v)}{\Delta, \Gamma \vdash v : S\{v\} \overset{[x_i' \mapsto x_i]}{\leadsto} \emptyset, [x_i' : \Gamma(x_i)] \vdash v[x_i \mapsto x_i'] : S\{v\}}$$

$$\frac{\{x_1, \dots, x_n\} = \mathit{free}(v)}{\Delta, \Gamma \vdash v : [\tau_i \to \tau_i'] \overset{[x_i' \mapsto x_i]}{\leadsto} \emptyset, [x_i' : \Gamma(x_i)] \vdash v[x_i \mapsto x_i'] : [\tau_i \to \tau_i']}$$

$$\Delta, \Gamma \vdash v : \tau_1 \to \tau_2 \overset{[x' \mapsto v]}{\leadsto} \emptyset, [x' : \tau_1 \to \tau_2] \vdash x' : \tau_1 \to \tau_2$$

$$\frac{\Delta, \Gamma \vdash v_i : \tau_i \overset{\gamma_i}{\leadsto} \emptyset, \Gamma_i \vdash v_i' : \tau_i \quad i \ne j \Rightarrow \mathit{dom}(\Gamma_i) \cap \mathit{dom}(\Gamma_j) = \emptyset}{\Delta, \Gamma \vdash c^S(v_1 \dots v_n) : \overline{c}(\tau_1 \dots \tau_n) \overset{\gamma_1 \cup \dots \cup \gamma_n}{\leadsto} \emptyset, \Gamma_1 \cup \dots \cup \Gamma_n \vdash c^S(v_1' \dots v_n') : \overline{c}(\tau_1 \dots \tau_n)}$$

$$\Delta, \Gamma \vdash v : \sum c_i(\tau_{i1} \dots \tau_{in_i}) \overset{[x' \mapsto v]}{\leadsto} \emptyset, [x' : \sum c_i(\tau_{i1} \dots \tau_{in_i})] \vdash x' : \sum c_i(\tau_{i1} \dots \tau_{in_i})$$

Splitting of types

$$\Gamma \vdash \overline{\texttt{m}} \leadsto [] \qquad \Gamma \vdash \overline{o}(\tau_1 \dots \tau_n) \leadsto []$$

$$\frac{\forall 1 \le i \le n. \Gamma \vdash \tau_i \leadsto \underline{\tau}_i}{\Gamma \vdash \overline{c}(\tau_1 \dots \tau_n) \leadsto \underline{\tau}_1 + \dots + \underline{\tau}_n} \qquad \frac{\{x_1, \dots, x_n\} = \mathit{free}(v)}{\Gamma \vdash S\{v\} \leadsto [\Gamma(x_1), \dots, \Gamma(x_n)]}$$

$$\frac{\Gamma \vdash \tau_i \leadsto \underline{\tau}_i \quad \Gamma \vdash \tau_i' \leadsto \underline{\tau}_i'}{\Gamma \vdash [\tau_i \to \tau_i'] \leadsto (\underline{\tau}_1 \to \underline{\tau}_1' \times \dots \times \underline{\tau}_n \to \underline{\tau}_n') + \underline{\tau}_1' + \dots + \underline{\tau}_n'}$$

$$\Gamma \vdash \texttt{int} \leadsto [\texttt{int}] \qquad \Gamma \vdash \tau_1 \to \tau_2 \leadsto [\tau_1 \to \tau_2]$$

$$\Gamma \vdash \sum c_i(\tau_{i1} \dots \tau_{in_i}) \leadsto [\sum c_i(\tau_{i1} \dots \tau_{in_i})]$$

Fig. 7. Splitting

Intermediate terms

$$e ::= \ldots \mid \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e \mid a\underline{\mathtt{@}}a \mid \underline{\mathtt{case}}\ a\ \underline{\mathtt{of}}\ c_i(x_{i1} \ldots x_{in_i})\ \mathtt{=>}\ a_i \ldots \mid$$
$$\mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle) \mid \mathtt{work}\langle \Gamma_i, v_i \mapsto e_i \rangle$$

Typing rules

$$(\mathit{mk\text{-}let})\frac{\Delta, \Gamma \vdash a : \tau \quad \Delta \vdash \tau : (\sigma; D) \quad \Delta, \Gamma[x : \tau] \vdash e : \tau'}{\Delta, \Gamma \vdash \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e : \tau'}$$

$$(\mathit{mk\text{-}case})\frac{\Delta, \Gamma \vdash a_0 : \sum c_i(\tau_{i1} \ldots \tau_{in_i}) \quad \Delta, \Gamma[x_{ij} : \tau_{ij}] \vdash a_i : \tau'}{\Delta, \Gamma \vdash \underline{\mathtt{case}}\ a_0\ \underline{\mathtt{of}}\ c_i(x_{i1} \ldots x_{in_i})\ \mathtt{=>}\ a_i \ldots : \tau'}$$

$$(\mathit{mk\text{-}app})\frac{\Delta, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \quad \Delta, \Gamma \vdash a_2 : \tau_2}{\Delta, \Gamma \vdash a_1\underline{\mathtt{@}}a_2 : \tau_1}$$

$$(\mathit{poly})\frac{\begin{array}{c}\emptyset \vdash_d \Gamma_i \quad \emptyset, \Gamma_i \vdash v_i : \tau_i \quad \Delta, \Gamma\Gamma_i \vdash a_i : \tau_i' \quad \Gamma \vdash \tau_i \rightsquigarrow \underline{\tau}_i \quad \Gamma \vdash \tau_i' \rightsquigarrow \underline{\tau}_i' \\ \Delta, \Gamma \vdash x : (\underline{\tau}_1 \to \underline{\tau}_1') \times \ldots \times (\underline{\tau}_n \to \underline{\tau}_n')\end{array}}{\Delta, \Gamma \vdash \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle) : [\tau_i \mapsto \tau_i']}$$

$$(\mathit{work})\frac{\begin{array}{c}\emptyset, \Gamma_i \vdash v_i : \tau_i \quad \Delta, \Gamma\Gamma_i \vdash e_i : \tau_i' \quad \emptyset \vdash_d \Gamma_i \\ (\exists v_i', \Gamma_i')\ \Gamma_i' \vdash v_i' : \tau_i \overset{\gamma_i}{\rightsquigarrow} \Gamma_i \vdash v_i : \tau_i\end{array}}{\Delta, \Gamma \vdash \mathtt{work}\langle \Gamma_i, v_i \mapsto e_i \rangle : [\tau_i \mapsto \tau_i']}$$

Fig. 8. Intermediate terms and typing rules

consequence, applying (*conv*) to the typing of e_2 yields $\Delta, \Gamma \vdash e_2 : \mathtt{if}\ \overline{1}\ \tau_2\ \tau_3$, as required.

Case $\mathtt{if}^S\ 0\ e_1\ e_2 \longrightarrow e_2$. Analogous.

Case $\mathtt{case}^S\ c_j^S(v_1 \ldots v_{n_j})\ \mathtt{of}\ c_i(x_{i1} \ldots x_{in_i})\ \mathtt{=>}\ e_i \ldots \longrightarrow e_j[x_{ik} \mapsto v_k]$.

Since $\Delta, \Gamma \vdash \mathtt{case}^S\ c_j^S(v_1 \ldots v_{n_j})\ \mathtt{of}\ c_i(x_{i1} \ldots x_{in_i})\ \mathtt{=>}\ e_i \ldots \longrightarrow e_j[x_{ik} \mapsto v_k] : \tau$, rule (*s-case*) requires that $\Delta, \Gamma \vdash c_j^S(v_1 \ldots v_{n_j}) : \tau_0$, $\Delta, \Gamma \vdash \tau_0 : \sum c_i^S(\rho_{i1} \ldots \rho_{in_i})$, $\Delta[\alpha_{ij} : \rho_{ij}], \Gamma[x_{ij} : \alpha_{ij}] \vdash e_i : \tau_i$, and $\tau = \mathtt{case}\ \tau_0\ \mathtt{of}\ c_i(\alpha_{i1} \ldots \alpha_{in_i})\ \mathtt{=>}\ \tau_i \ldots$. By rule (*s-con*), $\tau_0 = c_j(\tau_1' \ldots \tau_{n_j}')$, for some τ_k'. Hence $\tau = \mathtt{case}\ c_j(\tau_1 \ldots \tau_{n_j})\ \mathtt{of}\ c_i(\alpha_{i1} \ldots \alpha_{in_i})\ \mathtt{=>}\ \tau_i \ldots$ and $\Delta \vdash \tau = \tau_j[\alpha_{jk} \mapsto \tau_k']$. By the substitution lemma 3 applied to the typing of e_j, $\Delta, \Gamma \vdash e_j[x_{jk} \mapsto v_k] : \tau_j[\alpha_{jk} \mapsto \tau_k']$. Using rule (*conv*), we can derive the type τ from this.

Case $(\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1)@^S e_2 \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1 @^S e_2$.

By assumption, $\Delta, \Gamma \vdash (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1)@^S e_2 : \tau$. By rule (*s-app*), it must be that $\Delta, \Gamma \vdash \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1 : S\{\lambda x.e\}$, $\Delta, \Gamma \vdash e_2 : \tau_2$, $\Delta, \Gamma[x : \tau_2] \vdash e : \tau$. By rule (*mk-let*), it must be that $\Delta, \Gamma \vdash a : \tau_{11}$, $\Delta \vdash \tau_{11} : (\sigma; D)$, and $\Delta, \Gamma[x : \tau_{11}] \vdash e_1 : S\{\lambda x.e\}$. Applying rule (*s-app*) to $\Delta, \Gamma[x : \tau_{11}] \vdash e_1 : S\{\lambda x.e\}$ $\Delta, \Gamma[x : \tau_{11}] \vdash e_2 : \tau_2$, and $\Delta, \Gamma[x : \tau_2] \vdash e : \tau$ yields $\Delta, \Gamma[x : \tau_{11}] \vdash e_1 @^S e_2 : \tau$. Applying rule (*mk-let*) to $\Delta, \Gamma \vdash a : \tau_{11}$, $\Delta \vdash \tau_{11} : (\sigma; D)$, and $\Delta, \Gamma[x : \tau_{11}] \vdash e_1 @^S e_2 : \tau$ yields $\Delta, \Gamma \vdash \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1 @^S e_2 : \tau$, as required.

Case $v_1 @^S (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_2) \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ v_1 @^S e_2$. Analogous.

Case $\mathtt{if}^S\ (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_0)\ e_1\ e_2 \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ \mathtt{if}^S\ e_0\ e_1\ e_2$. Analogous.

Case $c^S(v_1 \ldots (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_k) \ldots e_n) \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ c^S(v_1 \ldots e_k \ldots e_n)$. Analogous.

Case $o^S(v_1 \ldots (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_k) \ldots e_n) \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ o^S(v_1 \ldots e_k \ldots e_n)$. Analogous.

Case $\mathtt{lift}\ (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e) \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ (\mathtt{lift}\ e)$. Analogous.

Case $\mathtt{spec}^S\ (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_1)@e_2 \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ (\mathtt{spec}^S\ e_1@e_2)$. Analogous.

Case $\mathtt{spec}^S\ v_1@(\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e_2) \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ (\mathtt{spec}^S\ v_1@e_2)$. Analogous.

Case $\mathtt{case}^S\ (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ e)\ \mathtt{of}\ \ c(x_1 \ldots) \mathtt{=>} e' \ldots \longrightarrow \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ \mathtt{case}^S\ e\ \mathtt{of}\ \ c(x_1 \ldots) \mathtt{=>} e' \ldots$. Analogous.

Case $\dfrac{\forall 1 \le i \le n. \exists v_i', \Gamma_i . v_i' \overset{\gamma_i}{\rightsquigarrow} v_i \quad \emptyset, \Gamma_i \vdash v_i : \tau_i \quad \emptyset \vdash_d \Gamma_i}{\mathtt{poly}^{\tau_1 \ldots \tau_n}\ x.e \longrightarrow \mathtt{work}\langle \Gamma_i, v_i \mapsto e[x \mapsto v_i]\rangle}$.

By assumption, $\Delta, \Gamma \vdash \mathtt{poly}^{\tau_1 \ldots \tau_n}\ x.e : \tau$. By rule (*s-poly*), it must be that, for all $1 \le i \le n$, $\Delta, \Gamma[x \mapsto \tau_i] \vdash e : \tau_i'$ and $\tau = [\tau_i \mapsto \tau_i']$. By assumption, $\emptyset, \Gamma_i \vdash v_i : \tau_i$ and, by the substitution lemma 3, $\Delta, \Gamma\Gamma_i \vdash e[x \mapsto v_i] : \tau_i'$. Applying rule (*work*) yields $\Delta, \Gamma \vdash \mathtt{work}\langle \Gamma_i, v_i \mapsto e_i[x \mapsto v_i]\rangle : [\tau_i \mapsto \tau_i']$.

Case $\dfrac{\forall 1 \le i \le n. \exists v_i' . v_i' \overset{\gamma_i}{\rightsquigarrow} v_i \quad \boldsymbol{x_i} = dom(\gamma_i) \quad w_i \overset{\gamma_i'}{\rightsquigarrow} w_i' \quad \boldsymbol{y_i} = dom(\gamma_i')}{\begin{array}{l}\mathtt{work}\langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle \\ \longrightarrow \underline{\mathtt{let}}\ x = (\lambda \boldsymbol{x_1}.A_1[\gamma_1' \boldsymbol{y_1}], \ldots, \lambda \boldsymbol{x_n}.A_n[\gamma_n' \boldsymbol{y_n}])\ \underline{\mathtt{in}}\ \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle)\end{array}}$.

By assumption $\Delta, \Gamma \vdash \mathtt{work}\langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle : \tau$, that is, by rule (*work*) $\emptyset, \Gamma_i \vdash v_i : \tau_i$, and $\Delta, \Gamma\Gamma_i \vdash A_i[w_i] : \tau_i'$, and $\emptyset \vdash_d \Gamma_i$, and $(\exists v_i', \Gamma_i')\ \Gamma_i' \vdash v_i' : \tau_i \overset{\gamma_i}{\rightsquigarrow} \Gamma_i \vdash v_i : \tau_i$, and $\tau = [\tau_i \mapsto \tau_i']$. By assumption, $\Gamma_i' \vdash v_i' : \tau_i \overset{\gamma_i}{\rightsquigarrow} \Gamma_i \vdash v_i : \tau_i$. Since $w_i \overset{\gamma_i'}{\rightsquigarrow} w_i'$, it must be that $\Gamma\Gamma_i \vdash A_i[w_i] : \tau_i \overset{\gamma_i'}{\rightsquigarrow} \Gamma_i'' \vdash w_i' : \tau_i$, by Lemma 2. Further, by assumption $\boldsymbol{x_i} = dom(\gamma_i)$ and $\boldsymbol{y_i} = dom(\gamma_i')$. Lemma 2 yields that $\Gamma_i \vdash \tau_i \rightsquigarrow \underline{\tau}_i$ where $\underline{\tau}_i = \Gamma_i(\boldsymbol{x_i})$ and $\Gamma\Gamma_i \vdash \tau_i' \rightsquigarrow \underline{\tau}_i'$ where $\underline{\tau}_i' = \Gamma_i''(\boldsymbol{y_i})$. Therefore, $\Delta, \Gamma[\boldsymbol{x_i} : \underline{\tau}_i] \vdash \gamma_i'(\boldsymbol{y_i}) : \underline{\tau}_i'$.

By rule (*d-abs*), $\Delta, \Gamma \vdash \lambda^D \boldsymbol{x_i}.\gamma_i'(\boldsymbol{y_i}) : \underline{\tau}_i \to \underline{\tau}_i'$ and thus $\Delta, \Gamma \vdash (\lambda^D \boldsymbol{x_1}.\gamma_1'(\boldsymbol{y_1}), \ldots, \lambda^D \boldsymbol{x_n}.\gamma_1'(\boldsymbol{y_n})) : (\underline{\tau}_1 \to \underline{\tau}_1') \times \ldots \times (\underline{\tau}_n \to \underline{\tau}_n')$. Applying rule (*poly*) yields $\Delta, \Gamma \vdash \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle) : [\tau_i \mapsto \tau_i']$.

Case $\dfrac{v_j \overset{\gamma}{\rightsquigarrow} v' \quad v \overset{\gamma'}{\rightsquigarrow} v' \quad \boldsymbol{x} = dom(\gamma') \quad w_j \overset{\gamma''}{\rightsquigarrow} w \quad \boldsymbol{y} = dom(\gamma'')}{\mathtt{spec}\ \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle)@v \longrightarrow \underline{\mathtt{let}}\ \boldsymbol{y} = \pi_j(x)\underline{\mathtt{@}}\gamma'(\boldsymbol{x})\ \underline{\mathtt{in}}\ w}$.

By assumption, $\Delta, \Gamma \vdash \mathtt{spec}\ \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle)@v : \tau$. By rule (*s-spec*), $\Delta, \Gamma \vdash \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto A_i[w_i]\rangle) : [\tau_i \to \tau_i']$, and $\Delta, \Gamma \vdash v : \tau_j$, and $\tau = \tau_j'$. By rule (*poly*), $\emptyset \vdash_d \Gamma_i$ and $\emptyset, \Gamma_i \vdash v_i : \tau_i$ and $\Delta, \Gamma\Gamma_i \vdash A_[w_i] : \tau_i'$ and $\Gamma \vdash \tau_i \rightsquigarrow \underline{\tau}_i$ and $\Gamma \vdash \tau_i' \rightsquigarrow \underline{\tau}_i'$ and $\Delta, \Gamma \vdash x : (\underline{\tau}_1 \to \underline{\tau}_1') \times \ldots \times (\underline{\tau}_n \to \underline{\tau}_n')$.

Need to prove the following judgement: $\Delta, \Gamma \vdash \underline{\mathtt{let}}\ \boldsymbol{y} = \pi_j(x)\underline{\mathtt{@}}\gamma'(\boldsymbol{x})\ \underline{\mathtt{in}}\ w : \tau_j'$. By rule (*mk-let*), the preconditions are $\Delta, \Gamma \vdash \pi_j(x)\underline{\mathtt{@}}\gamma'(\boldsymbol{x}) : \underline{\tau}_j'$, and $\Delta \vdash \underline{\tau}_j' : (\sigma; D)$, and $\Delta, \Gamma[\boldsymbol{y} : \underline{\tau}_j'] \vdash w : \tau_j'$. The last two of these follow from the splitting lemma 2.

For the first precondition to hold, by (*mk-app*), it must be that $\Delta, \Gamma \vdash \pi_j(x) : \underline{\tau}_j \to \underline{\tau}_j'$ and $\Delta, \Gamma \vdash \gamma'(\boldsymbol{x}) : \underline{\tau}_j$. The first of these holds by assumption on x and the typing rule for products. The second one holds by the splitting lemma 2.

Case $e \longrightarrow e'$ implies $C[e] \longrightarrow C[e']$. For all contexts, C, the claim follows directly from the inductive hypothesis.

Lemma 1 (Static canoncial forms). *Suppose $\Delta, \Gamma \vdash v : \tau$ and $\Delta \vdash_d \Gamma$.*

1. *If $\Delta \vdash \tau = \overline{\mathtt{m}}$ then $v = m$.*
2. *If $\Delta \vdash \tau = \underline{S}\{\lambda x.e\}$ then $v = \lambda x.e$.*
3. *If $\Delta \vdash \tau = \overline{c}(\tau_1 \ldots \tau_n)$ then $v = c^S(v_1 \ldots v_n)$.*
4. *If $\Delta \vdash \tau = [\tau_i \to \tau_i']$ then $v = \mathtt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle)$.*

Lemma 2 (Splitting).

1. *Suppose $\Delta, \Gamma \vdash v : \tau$ and $v \overset{\gamma}{\rightsquigarrow} v'$.*
   Then $\Gamma \vdash v : \tau \overset{\gamma'}{\rightsquigarrow} \Gamma' \vdash v' : \tau$ and $\Delta, \Gamma' \vdash v' : \tau$.
2. *Suppose $\Delta, \Gamma \vdash v : \tau$ and $\Gamma \vdash v : \tau \overset{\gamma'}{\rightsquigarrow} \Gamma' \vdash v' : \tau$ and $\Gamma \vdash \tau \rightsquigarrow \underline{\tau}'$.*
   Then $\underline{\tau} = \Gamma'(dom(\gamma'))$.
3. *If $\Delta \vdash \Gamma$ ctx and $\Gamma \vdash \tau \rightsquigarrow \underline{\tau}'$ then $\Delta \vdash \underline{\tau}' : (\sigma, D)$.*
4. *If $\Delta, \Gamma \vdash v_j : \tau$, for $j \in \{1, 2\}$, then there exist v, γ_1, and γ_2 such that $v_1 \overset{\gamma_1}{\rightsquigarrow} v$ and $v_2 \overset{\gamma_2}{\rightsquigarrow} v$.*

Proof. **Ad item 1:** By induction on the definition of $v \overset{\gamma}{\rightsquigarrow} v'$, using Lemma 1.

Ad item 2: By induction on the definition of $\Gamma \vdash \tau \rightsquigarrow \underline{\tau}'$.

Ad item 3: By induction on the definition of $\Gamma \vdash \tau \rightsquigarrow \underline{\tau}'$.

Lemma 3 (Substitution). *Suppose $\Delta[\alpha : \rho], \Gamma[x : \alpha] \vdash e : \tau'$ $\Delta, \Gamma \vdash v : \tau$ and $\Delta, \Gamma \vdash \tau : \rho$.*

Then $\Delta, \Gamma \vdash e[x \mapsto v] : \tau'[\alpha \mapsto \tau]$.

Proof. Induction on e.

Case x. In this case $\Delta[\alpha : \rho], \Gamma[x : \alpha] \vdash x : \alpha$ hence $\tau' = \alpha$. Therefore, $\Delta, \Gamma \vdash e[x \mapsto v] : \tau'[\alpha \mapsto \tau]$.

Case $y \neq x$. In this case $\Delta[\alpha : \rho], \Gamma[x : \alpha] \vdash y : \tau'$. If $\alpha \notin \mathit{free}(\tau')$ then the claim is immediate. If $\alpha \in \mathit{free}(\tau')$ then since $\Delta, \Gamma \vdash \tau : \rho$ it follows that $\Delta, \Gamma[\alpha \mapsto \tau] \vdash y : \tau'[\alpha \mapsto \tau]$.

All other cases simply appeal to the inductive hypothesis.

Lemma 4 (Progress). *Suppose $\Delta, \Gamma \vdash e : \tau$, where $\tau \neq \top$ and $\Delta \vdash_d \Gamma$. Either e is an answer a or there exists e' such that $e \longrightarrow e'$.*

Proof. By induction on the derivation of $\Delta, \Gamma \vdash e : \tau$.

Case (var).

$$(\mathit{var})\frac{\Delta \vdash \Gamma \text{ ctx} \quad \Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

Clearly, x is a value and τ is a dynamic type, due to $\Delta \vdash_d \Gamma$.

Case (const).

$$(\mathit{const})\frac{\Delta \vdash \Gamma \text{ ctx}}{\Delta, \Gamma \vdash m : \overline{\mathtt{m}}}$$

The term m is a value.

Case (top).

$$(top)\frac{\Delta \vdash \Gamma \text{ ctx}}{\Delta, \Gamma \vdash e : \top}$$

Is not applicable.

Case (lift).

$$(\mathit{lift})\frac{\Delta, \Gamma \vdash e : \tau \quad \Delta \vdash \tau : (\mathtt{int}; S)}{\Delta, \Gamma \vdash \mathtt{lift}\ e : \mathtt{int}}$$

By induction, either $e \longrightarrow e'$, in which case $\mathtt{lift}\ e \longrightarrow \mathtt{lift}\ e'$, or e is an answer. If $e = \underline{\mathtt{let}}\ x = a'\ \underline{\mathtt{in}}\ a''$ then $\mathtt{lift}\ (\underline{\mathtt{let}}\ x = a'\ \underline{\mathtt{in}}\ a'')$ is a redex. If e is a value, then Lemma 1 yields that $e = m$ and $\mathtt{lift}\ m$ is a value, too.

Case $(s\text{-}abs)$.

$$(s\text{-}abs)\frac{\mathit{dom}(\Gamma) \supseteq \mathit{free}(\lambda x.e)}{\Delta, \Gamma \vdash \lambda^S x.e : S\{\lambda x.e\}}$$

$\lambda^S x.e$ is a value.

Case $(d\text{-}abs)$.

$$(d\text{-}abs)\frac{\Delta, \Gamma[x : \tau'] \vdash e : \tau \quad \Delta, \Gamma \vdash \tau' \to \tau : \rho}{\Delta, \Gamma \vdash \lambda^D x.e : \tau' \to \tau}$$

Since $\Delta \vdash \tau' \to \tau : \rho$, it must be that $\rho = (\rho_1 \to \rho_2; D)$ and $\Delta \vdash \tau' : \rho_1$ and $\Delta \vdash \tau : \rho_2$ $\rho_i = (\sigma_i; D)$. In particular, $\Delta \vdash_d \Gamma[x : \tau']$. By induction, for $\Delta, \Gamma[x : \tau'] \vdash e : \tau$, we obtain that either $e = a$ is an answer, in which case $\lambda^D x.a$ is a value, or $e \longrightarrow e'$, in which case $\lambda^D x.e \longrightarrow \lambda^D x.e'$.

Case $(s\text{-}app)$.

$$(s\text{-}app)\frac{\Delta, \Gamma \vdash e_1 : S\{\lambda x.e\} \quad \Delta, \Gamma \vdash e_2 : \tau_2 \quad \Delta, \Gamma[x : \tau_2] \vdash e : \tau}{\Delta, \Gamma \vdash e_1 @^S e_2 : \tau}$$

By induction, either $e_1 \longrightarrow e_1'$, in which case $e_1 @^S e_2 \longrightarrow e_1' @^S e_2$, or e_1 is an answer. If $e_1 = \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ a'$ then $(\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ a') @^S e_2$ is a redex. If e_1 is a value, Lemma 1 implies that $e_1 = \lambda x.e$. Further by induction, either $e_2 \longrightarrow e_2'$, in which case $(\lambda x.e) @^S e_2 \longrightarrow (\lambda x.e) @^S e_2'$, or e_2 is an answer. If $e_2 = \underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ a'$ then $(\lambda x.e) @^S (\underline{\mathtt{let}}\ x = a\ \underline{\mathtt{in}}\ a')$ is a redex. If $e_2 = v$ is a value, we also have a redex: $(\lambda x.e) @^S v \longrightarrow e[x := v]$.

Case $(d\text{-}app)$.

$$(d\text{-}app)\frac{\Delta, \Gamma \vdash e_1 : \tau_2 \to \tau \quad \Delta, \Gamma \vdash e_2 : \tau_2}{\Delta, \Gamma \vdash e_1 @^D e_2 : \tau}$$

By induction, either $e_1 \longrightarrow e_1'$, in which case $e_1 @^D e_2 \longrightarrow e_1' @^D e_2$, or $e_1 = a_1$ is an answer. Further, by induction, either $e_2 \longrightarrow e_2'$, in which case $a_1 @^D e_2 \longrightarrow a_1 @^D e_2'$, or $e_2 = a_2$ is an answer. In the latter case, the expression $a_1 @^D a_2$ is a value.

Case (*s-poly*).

$$(s\text{-}poly)\frac{(\forall 1 \le i \le n)\ \Delta, \Gamma[x \mapsto \tau_i] \vdash e : \tau_i'}{\Delta, \Gamma \vdash \texttt{poly}^{\tau_1 \cdots \tau_n}\ x.e : [\tau_i \mapsto \tau_i']}$$

$\texttt{poly}^{\tau_1 \cdots \tau_n}\ x.e$ is a redex.

Case (*s-spec*).

$$(s\text{-}spec)\frac{\Delta, \Gamma \vdash e_1 : [\tau_i \to \tau_i'] \quad \Delta, \Gamma \vdash e_2 : \tau_j}{\Delta, \Gamma \vdash \texttt{spec}\ e_1 @ e_2 : \tau_j'}$$

By induction, either $e_1 \longrightarrow e_1'$, in which case $\texttt{spec}\ e_1@e_2 \longrightarrow \texttt{spec}\ e_1'@e_2$, or e_1 is an answer. If $e_1 = \underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a''$ then $\texttt{spec}\ (\underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a'')@e_2$ is a redex. If e_1 is a value then, by Lemma 1, $e_1 = \texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle)$. By further induction, either $e_2 \longrightarrow e_2'$, in which case $\texttt{spec}\ (\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle))@e_2 \longrightarrow \texttt{spec}\ (\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle))@e_2'$, or e_2 is an answer. If $e_2 = \underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a''$ then $\texttt{spec}\ (\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto w_i \rangle))@(\underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a'')$ is a redex. Otherwise, if $e_2 = v$ is a value, since $v_j \overset{\gamma}{\leadsto} v'$, it must be that $v \overset{\gamma'}{\leadsto} v'$, by Lemma 2. Hence, $\texttt{spec}\ (\texttt{POLY}(x, \langle \Gamma_i, v_i \mapsto a_i \rangle))@v$ is a redex.

Case (*s-prim*).

$$(s\text{-}prim)\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Delta, \Gamma \vdash e_n : \tau_n \quad \Delta \vdash \overline{o}(\tau_1 \ldots \tau_n) : (\texttt{int}; S)}{\Delta, \Gamma \vdash o^S(e_1 \ldots e_n) : \overline{o}(\tau_1 \ldots \tau_n)}$$

Since $\Delta \vdash \overline{o}(\tau_1 \ldots \tau_n) : (\texttt{int}; S)$ it must be that $\Delta \vdash \tau_i : (\texttt{int}; S)$. By induction, either $e_1 \longrightarrow e_1'$, in which case $o^S(e_1 \ldots e_n) \longrightarrow o^S(e_1' \ldots e_n)$, or e_1 is an answer. If $e_1 = \underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a''$ then $o^S((\underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a'') \ldots e_n)$ is a redex. If e_1 is a value, then proceed inductively up to the case where e_n is a value, in which case $o^S(v_1 \ldots v_n)$ is a redex.

Case (*d-prim*).

$$(d\text{-}prim)\frac{\Delta, \Gamma \vdash e_1 : \texttt{int} \quad \ldots \quad \Delta, \Gamma \vdash e_n : \texttt{int}}{\Delta, \Gamma \vdash o^D(e_1 \ldots e_n) : \texttt{int}}$$

By induction, either $e_1 \longrightarrow e_1'$, in which case $o^D(e_1 \ldots e_n) \longrightarrow o^D(e_1' \ldots e_n)$, or e_1 is an answer, so that it forms an evaluation context for e_2. Proceed inductively up to the case where e_n is an answer. In this case, $o^D(a_1 \ldots a_n)$ is a value.

Case (*s-con*).

$$(s\text{-}con)\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Delta, \Gamma \vdash e_n : \tau_1}{\Delta, \Gamma \vdash c^S(e_1 \ldots e_n) : \overline{c}(\tau_1 \ldots \tau_n)}$$

Similar to (*s-prim*), except in the end, when all arguments are values, the term $c^S(v_1 \ldots v_n)$ is a value.

Case (*d-con*). Analogous to (*d-prim*).

Case (*s-case*).

$$(s\text{-}case)\frac{\begin{array}{c}\Delta, \Gamma \vdash e_0 : \tau_0 \quad \Delta \vdash \tau_0 : \sum c_i^S(\rho_{i1} \ldots \rho_{in_i}) \\ \Delta[\alpha_{ij} : \rho_{ij}], \Gamma[x_{ij} : \alpha_{ij}] \vdash e_i : \tau_i \quad (1 \le i \le n)\end{array}}{\Delta, \Gamma \vdash \texttt{case}^S\ e_0\ \texttt{of}\ c_i(x_{i1} \ldots x_{in_i}) \texttt{=>} e_i \ldots : \texttt{case}\ \tau_0\ \texttt{of}\ c_i(\alpha_{i1} \ldots \alpha_{in_i}) \texttt{=>} \tau_i \ldots}$$

By induction, either $e_0 \longrightarrow e'_0$, in which case the whole expression reduces, or e_0 is an answer. If $e_0 = \underline{\texttt{let}}\ x = a'\ \underline{\texttt{in}}\ a''$ then the whole expression is a redex. If e_0 is a value, by Lemma 1, it must have the form $c_j^S(v_1 \dots v_{n_j})$. In this case, the whole expression is a redex.

Case (*d-case*).

$$(d\text{-}case)\frac{\Delta, \Gamma \vdash e_0 : \sum c_i(\tau_{i1} \dots \tau_{in_i}) \quad \Delta, \Gamma[x_{ij} : \tau_{ij}] \vdash e_i : \tau' \quad (1 \le i \le n)}{\Delta, \Gamma \vdash \texttt{case}^D\ e_0\ \texttt{of}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} e_i \dots : \tau'}$$

By induction, either $e_0 \longrightarrow e'_0$, in which case $\texttt{case}^D\ e_0\ \texttt{of}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} e_i \dots \longrightarrow \texttt{case}^D\ e'_0\ \texttt{of}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} e_i \dots$, or $e_0 = a_0$ is an answer. Since $\Delta \vdash \sum c_i(\tau_{i1} \dots \tau_{in_i}) : (\sigma, D)$ it must be that $\Delta \vdash \tau_{ij} : (\sigma_{ij}, D)$, hence $\Delta \vdash_d \Gamma[x_{ij} : \tau_{ij}]$. Therefore, by induction either $e_1 \longrightarrow e'_1$, in which case the whole term makes a reduction step, or e_1 is an answer. Proceed inductively to the case where e_n is an answer. If all e_i are answers then the whole term is a redex.

Case (*conv*). Immediate, by applying the inductive hypothesis.

Case (*mk-let*).

$$(mk\text{-}let)\frac{\Delta, \Gamma \vdash a : \tau \quad \Delta \vdash \tau : (\sigma; D) \quad \Delta, \Gamma[x : \tau] \vdash e : \tau'}{\Delta, \Gamma \vdash \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e : \tau'}$$

By induction, either $e \longrightarrow e'$, in which case $\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e \longrightarrow \underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ e'$, or $e = a'$ is an answer, in which case $\underline{\texttt{let}}\ x = a\ \underline{\texttt{in}}\ a'$ is an answer, too.

Case (*mk-case*). All subterms are syntactically restricted to be answers. Hence, $\underline{\texttt{case}}\ a_0\ \underline{\texttt{of}}\ c_i(x_{i1} \dots x_{in_i}) \texttt{=>} a_i \dots$ is a value.

Case (*mk-app*). All subterms are syntactically restricted to be answer. Hence, $a_1 \underline{\texttt{@}} a_2$ is a value.

Case (*poly*). This term is always a value.

Case (*work*).

$$(work)\frac{\begin{array}{c}\emptyset, \Gamma_i \vdash v_i : \tau_i \quad \Delta, \Gamma\Gamma_i \vdash e_i : \tau'_i \quad \emptyset \vdash_d \Gamma_i \\ (\exists v'_i, \Gamma'_i)\ \Gamma'_i \vdash v'_i : \tau_i \overset{\gamma_i}{\rightsquigarrow} \Gamma_i \vdash v_i : \tau_i\end{array}}{\Delta, \Gamma \vdash \texttt{work}\langle \Gamma_i, v_i \mapsto e_i \rangle : [\tau_i \mapsto \tau'_i]}$$

Since $\emptyset \vdash_d \Gamma_i$ it follows that $\Delta \vdash_d \Gamma\Gamma_i$. Hence, by induction, either $e_1 \longrightarrow e'_1$, which leads to the corresponding reduction of the whole term, or e_1 is an answer. Proceed inductively up to the case where e_n is an answer. Then the whole term is a redex.

Theorem 1 (Type soundness). *Suppose $\emptyset, \emptyset \vdash e : \tau$. Either there exists v so that $e \xrightarrow{*} v$ and $\emptyset, \emptyset \vdash v : \tau$, or for each $e \xrightarrow{*} e'$ there exists e'' so that $e' \longrightarrow e''$.*

Proof. Immediate from subject reduction (Proposition 1) and the progress Lemma 4.

6 Extended example

It is interesting to consider the results of particular uses of polyvariance. For example, specializing `evalExp (Lambda ("x", Var "x"))` gives rise to

```
   Fun (poly x -> x)
==
   let poly_1 = () in
   Fun poly_1
```

which means that (apparently) dead code is not specialized at all.

Another outcome is also possible: consider `evalExp (App (Lambda ("f", App (App (Var "f", Var "f"), Const 3)), Lambda ("x", Var "x"))) []`, which specializes to

```
      let valoffe = evalExp (Lambda ("f", App (App (Var "f", Var "f"), Const 3))) [] in
      let valofae = evalExp (Lambda ("x", Var "x")) [] in
      exFun valoffe valofae
==
      let valoffe = Fun (poly x -> evalExp (App (App (Var "f", Var "f"), Const 3)) (extendEnv [] "f" x)) in
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let valoffe = Fun (poly x ->
        let valoffe = evalExp (App (Var "f", Var "f"), Const 3) [("f", x)] in
        let valofae = evalExp (Const 3) [("f", x)] in
        exFun valoffe valofae
      in
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let valoffe = Fun (poly x ->
        let valoffe =
          let valoffe = evalExp (Var "f") [("f", x)] in
          let valofae = evalExp (Var "f") [("f", x)] in
          exFun valoffe valofae
        in
        let valofae = Int 3 in
        exFun valoffe valofae
      in
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let valoffe = Fun (poly x ->
        let valoffe =
          let valoffe = x in
          let valofae = x in
          exFun valoffe valofae
        in
        let valofae = Int 3 in
        exFun valoffe valofae
      in
```

```
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let valoffe = Fun (poly x ->
        let valoffe = exFun x x in
        exFun valoffe (Int 3))
      in
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let poly_1 = (fun f -> let valoffe = exFun (Fun (proj 1 f)) (Fun (proj 2 f)) in
                             exFun valoffe (Int 3))
      let valoffe = Fun poly_1 in
      let valofae = Fun (poly x -> x) in
      exFun valoffe valofae
==
      let poly_1 = (fun f -> let valoffe = Fun ((proj 1 f) (proj 2 f)) in
                             exFun valoffe (Int 3))
      let poly_2 = (fun x -> x, fun x -> x)
      let valoffe = Fun poly_1 in
      let valofae = Fun poly_2 in
      exFun valoffe valofae
==
      let poly_1 = (fun f -> ((proj 1 f) (proj 2 f)) (Int 3)) =
      let poly_2 = (fun x -> x, fun x -> x) =
      poly_1 (Fun poly_2)
==
      let poly_1 = fun f -> ((proj 1 f) (proj 2 f)) 3 in
      let poly_2 = (fun x -> x, fun x -> x) in
      Int (poly_1 poly_2)
```

7 Conclusions

We have demonstrated that an extension of standard partial evaluation by first-class polyvariant functions and co-arity raising is sufficient to perform tag removal. This extension has been formalized using a type system with singleton types and a structural operational semantics. Type soundness has been proved. The extension is implemented in the author's partial evaluation system for Scheme [25].

In another work, we have shown that HTS can insert security checks according to a safety policy into potentially unsafe code [24]. The structure of this problem is very similar to the TSP addressed in the present work. In the other work, we have to extend HTS by intersection types to address excessive code duplication. We conjecture that a similar extension is possible for the PGG system, but we leave this as future work.

References

1. David Aspinall. Subtyping with singleton types. In *Computer Science Logic, CSL'94*, Lecture Notes in Computer Science, Kazimierz, Poland, 1994. Springer-Verlag.
2. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.
3. Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
4. Olivier Danvy. A simple solution to type specialization. Technical Report RS-98-1, BRICS, University of Aarhus, Denmark, January 1998. To appear in ICALP98.
5. Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialization. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 54–63, La Jolla, CA, June 1995. ACM Press.
6. Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.
7. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [8].
8. Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See [7].
9. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, 1999. Springer-Verlag.
10. Susumu Hayashi. Singleton, union, and intersection types for program extraction. *Information and Computation*, 109, 1994.
11. Luke Hornof and Jacques Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, October 2000.
12. John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.
13. John Hughes. A type specialisation tutorial. In Hatcliff et al. [9], pages 293–325.
14. Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1987. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.
15. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
16. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.

17. Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *Lecture Notes In Computer Science*, pages 129–148, Montreal, Canada, 20 September 2000. Springer-Verlag.
18. Torben Æ. Mogensen. Inherited limits. In Hatcliff et al. [9], pages 189–202.
19. Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347, Amsterdam, 1988. North-Holland.
20. Torben Æ. Mogensen. Constructor specialization. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 22–32, Copenhagen, Denmark, June 1993. ACM Press.
21. Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360, Copenhagen, Denmark, 1990. Springer-Verlag.
22. Walid Taha. Tag elimination or type specialization is a type-indexed effect. In Gilles Barthe, editor, *APPSEM Workshop on Subtyping & Dependent Types in Programming*, Ponte de Lima, Portugal, July 2000. Available at http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.
23. Peter Thiemann. Interpreting specialization in type theory. In Olivier Danvy, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, pages 30–43, San Antonio, Texas, USA, January 1999. BRICS Notes Series NS-99-1.
24. Peter Thiemann. Enforcing safety properties using type specialization. Technical report, Universität Freiburg, July 2000. Available from http://www.informatik.uni-freiburg.de/ thiemann/papers/espps-het.ps.gz.
25. Peter Thiemann. *The PGG System—User Manual*. Universität Freiburg, Freiburg, Germany, March 2000. Available from http://www.informatik.uni-freiburg.de/proglang/software/pgg/.
26. Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997. http://www.informatik.uni-freiburg.de/ thiemann/papers/mlpe.ps.gz.
27. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

A Glossary of partial evaluation terms

binding time staging information of a value; the earliest time that a value is available; usually specialization distinguishes two binding times, *static* (early, available at specialization-time) and *dynamic* (late, available at run-time).

dynamic *binding time* of a value available at run-time.

dynamic conditional a conditional expression where the *binding time* of the condition is *dynamic*.

memoization In general, a memoized function remembers the arguments and results of previous calls to deliver the results of repeated calls with the same arguments by a table lookup. In partial evaluation, memoization applies to certain recursive calls of the specializer. The static parts of the arguments serve as key for the table lookup because the specialized code only depends on them. Memoization improves the sharing of specialized code and improves termination.

static *binding time* of a value available at specialization-time.