Polymorphic Specialization for Polymorphic Programming Languages*

Simon Helsen and Peter Thiemann

Universität Freiburg**, Germany

Abstract. We present a framework for offline partial evaluation for functional programming languages with an ML-style type discipline. The framework comprises a polymorphic binding-time analysis with polymorphic recursion. It is based on the region calculus of Tofte and Talpin. A novel aspect is to regard binding times as properties of regions. This results in a modular analysis since the binding-time analysis reduces to a constraint analysis on top of the region analysis.
The technical contributions include a small-step operational semantics for evaluation as well as specialization in the region calculus. We specify the binding-time analysis as an extension of the static semantics of the region calculus. Using the method of Felleisen and Wright we prove the type soundness of evaluation with respect to the standard rules and type soundness of specialization with respect to the binding-time rules. In addition, we prove that specialization preserves the (call-by-value) semantics by showing that the reductions of the specializer are operational equivalences in the region calculus.

Key words: semantics, regions, partial evaluation

1 Introduction

Partial evaluation [5, 11] is a technique for factorizing the evaluation of a program in two phases. In the first, *static*, phase all operations depending only on known data are executed, leaving behind a *specialized program.* The second, *dynamic*, phase executes the remaining operations in the specialized program. Partial evaluation technology owes much to the introduction of the concept of binding-time analysis [12]. This program analysis determines which constructs in a program can safely be executed in the static phase. It yields an annotation of each construct with a binding time, ie, the phase in which it is executed. Many studies and systems consider such *monovariant* analyses and many applications have been studied in the context of these systems.

The concept of binding-time analysis as described above is unnecessarily conservative because each construct in the program has one fixed binding time throughout the whole specialization. A more expressive policy would consider a binding-time annotation for functions which depends on the context of each particular call of a function, say. This is basically a *polyvariant* analysis [4].

However, a polyvariant analysis cannot know in advance how many differently annotated versions of a function are required. It relies on user guidance or ad-hoc techniques to ensure a finite number of annotations. Another alternative is to consider polymorphic binding-time analysis which transfers the idea of parametric polymorphism from types to binding times [7, 10]. Previous results following this approach yield an analysis for a simply typed applied lambda calculus. The analysis itself is binding-time polymorphic and admits polymorphic recursion on the level of binding times [7]. Further, it employs a notion of subtyping (where only the binding times change, but not the structure of the type itself) and expresses dependencies between as yet unknown binding times using constrained type schemes.

Contributions and outline The contribution of this work is to define a polymorphic binding-time analysis (with polymorphic recursion) for a core language of ML with polymorphic types. The novelty of our approach lies in the choice of Talpin and Tofte's region calculus [19, 20] as the basis for a binding-analysis. Since the region calculus already provides polymorphic recursion on the level of regions, we obtain a binding-time analysis with the same properties by annotating regions with binding times. The result is a modular analysis which starts with the result of a region analyzer and adds binding-time constraints on top of that.

The technical contributions are the definition of a small-step operational semantics for the region calculus and a proof of its type soundness using the method of Wright and Felleisen [21] (in Section 2). Then, in Section 3, we define a binding-time analysis by extending the static semantics of the region calculus with constraints on the binding times of regions. Next, in Section 4, we give a small-step operational semantics for terms in the binding-time analyzed region calculus. In Section 5, we prove the type soundness of specialization with respect to the binding-time analysis rules. In addition, we prove the semantic soundness of specialization in the sense that every transformation step performed by the specializer is an equivalence in the (call-by-value) region calculus. Finally, we given an example (in Section 6) and conclude.

Due to lack of space, we can only include a very limited account of the region calculus. For the problem of region inference, on which our analysis builds, we refer to the respective publications [3, 18].

Related work In earlier work, Thiemann [16] considered another region calculus [15] as the basis of a binding-time analysis. However, that work covered a monovariant analysis for a simply typed language. In addition, it was geared at the treatment of impure effects on reference cells. It would be interesting to combine the two systems, but there are some notable differences in the treatment of effects on arrows.

Compared to the work of Dussart and others [7] (which supersedes [10]), we did not include a notion of binding-time subtyping—except explicitly on base types—because it complicates the analysis. In addition, its practical use is debatable since it may easily lead to code duplication in specialized programs.

There is an indirect connection between binding-time analysis and the region calculus in the work on the Dependency Core Calculus [1] and the polymorphic region calculus [2]. In both works, the underlying semantic model is quite similar. Since we basically show that a binding-time analysis can be based on region analysis, it seems that the region calculus is the more general model.

2 The region calculus

The region calculus, $\lambda^{\mathrm{region}}$, is a slight extension of the calculus given by Tofte and Talpin [19, 20]. The following grammar generates its terms:

$$\begin{aligned} e ::= {} & x \mid c \ \mathtt{at}\ \rho \mid \lambda x.\, e \ \mathtt{at}\ \rho \mid e_1 \mathbin{@} e_2 \mid \mathtt{letrec}\ f\,[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = e_1\ \mathtt{in}\ e_2 \mid \\ & \mathtt{letregion}\ \rho\ \mathtt{in}\ e \mid \gamma\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \end{aligned}$$

$$\gamma ::= f \mid \Lambda[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = e$$

The $\lambda^{\mathrm{region}}$-calculus extends the λ-calculus with explicit *region* annotations. The term, `letregion` ρ `in` e, allocates a new region of memory and binds it to the region variable ρ. Each data construction is annotated with the region, `at` ρ, in which the data is put. We distinguish `let` or λ-bound variables x from `letrec`-bound variables f. A `letrec` defines a recursive function with additional region parameters. A *region application* $\gamma\ [\rho_1, \ldots, \rho_n]$ `at` ρ applies the *rator* γ to its region arguments and puts the resulting lambda into region ρ. In the Tofte-Talpin calculus, a rator is always a `letrec`-bound variable f. In $\lambda^{\mathrm{region}}$, γ can be an anonymous region abstraction to simplify the small-step semantics defined below.

A region variable occurs free in e if there is no intervening `letregion` or `letrec` which binds ρ. The set, $\mathit{frv}(e)$, contains the free region variables of e. The set, $\mathit{free}(e)$, of the free variables of an expression e and is defined as usual. Substitutions, $e[e' / x]$, replace all free occurrences of x for e'. All variables (including region variables) are subject to α conversion, which is tacitly assumed wherever a capture might occur.

For use in the specialization phase, there is a predefined function `lift`, which is the identity on integer values, but transfers the value from one region to another. It behaves like a free function variable, but has special reduction rules.

Figure 1 defines the small-step evaluation relation $\to_r$ in the usual way by defining a subset of the terms as syntactic values, giving reduction rules, and defining a grammar for evaluation contexts. We write $\twoheadrightarrow_r$ for the reflexive transitive closure of $\to_r$.

Syntactic values $v ::= c \ \mathtt{at}\ \rho \mid \lambda x.\, e \ \mathtt{at}\ \rho \mid \mathtt{lift}\ [\rho_1, \rho_2]\ \mathtt{at}\ \rho$

Evaluation contexts $\mathcal{E} ::= [\,] \mid \mathcal{E} \mathbin{@} e \mid v \mathbin{@} \mathcal{E} \mid \mathtt{let}\ x = \mathcal{E}\ \mathtt{in}\ e \mid \mathtt{letregion}\ \rho\ \mathtt{in}\ \mathcal{E}$

(1) $(\lambda x.\, e\ \mathtt{at}\ \rho) \mathbin{@} v \rightarrow_r \mathtt{let}\ x = v\ \mathtt{in}\ e$

(2) $(\mathtt{lift}\ [\rho_1, \rho_2]\ \mathtt{at}\ \rho) \mathbin{@} (c\ \mathtt{at}\ \rho_1) \rightarrow_r c\ \mathtt{at}\ \rho_2$

(3) $\mathtt{let}\ x = v\ \mathtt{in}\ e \rightarrow_r e[v \,/\, x]$

(4) $\mathtt{letregion}\ \rho\ \mathtt{in}\ v \rightarrow_r v \qquad \mathit{if}\ \rho \notin \mathit{frv}(v)$

(5) $\mathtt{letrec}\ f\,[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = e_1\ \mathtt{in}\ e_2 \rightarrow_r$
$e_2[(\Lambda[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = (\mathtt{letrec}\ f\,[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = e_1\ \mathtt{in}\ e_1))\,/\,f]$

(6) $(\Lambda[\rho_1, \ldots, \rho_n](x)\ \mathtt{at}\ \rho = e)\ [\rho_1', \ldots, \rho_n']\ \mathtt{at}\ \rho_0 \rightarrow_r$
$\lambda x.\, e[\rho_1' \,/\, \rho_1, \ldots, \rho_n' \,/\, \rho_n]\ \mathtt{at}\ \rho_0$

(7) $\dfrac{e \rightarrow_r e'}{\mathcal{E}[e] \rightarrow_r \mathcal{E}[e']}$

Fig. 1. Small-step Semantics

Rules (1) and (3) define β-reduction, rule (2) formalizes the semantics of `lift` and rule (7) extends the reduction to all evaluation contexts. Rule (4) discharges a region-bound variable if it does not occur in the value anymore. Rules (5) and (6) reduce region abstractions. The substitution in rule (5) unrolls the `letrec` once.

2.1 Static semantics of $\boldsymbol{\lambda}^{\mathrm{region}}$

We briefly summarize the relevant semantic objects for the type system. A detailed account is found in Tofte and Talpin's original paper [20].

An *effect* φ is a finite set of region variables ρ and *effect variables* ϵ. An effect describes which regions are affected when evaluating a construct. A *type* τ and *type with place* μ is defined by

$$\tau ::= \mathtt{int} \mid \alpha \mid \mu \xrightarrow{\epsilon.\varphi} \mu$$
$$\mu ::= (\tau, \rho)$$

where $\epsilon.\varphi$ is an arrow effect and α is a type variable. The arrow effect is the latent effect which occurs when applying the function. An effect variable *stands* for an arrow effect. This is used to describe dependencies between effects. A type with place indicates where the object of given type is stored. Integer and function types are as usual.

A type substitution, S_t, is a map from type variables to types, a region substitution, S_r, is a map from region variables to region variables and an effect substitution, S_e, is a map from effect variables to arrow effects. A substitution,

$S_s = (S_t, S_r, S_e)$, is then defined as follows :

$$S_s(\varphi) = \{S_r(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \epsilon \in \varphi \wedge S_e(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}$$

$$S_s(\texttt{int}) = \texttt{int} \quad S_s(\alpha) = S_t(\alpha) \quad S_s(\rho) = S_r(\rho) \quad S_s(\tau, \rho) = (S_s(\tau), S_r(\rho))$$

$$S_s(\mu \xrightarrow{\epsilon.\varphi} \mu') = S_s(\mu) \xrightarrow{\epsilon'.(\varphi' \cup S_s(\varphi))} S_s(\mu') \quad \textit{where } S_e(\epsilon) = \epsilon'.\varphi'$$

Type schemes σ extend those of Damas and Milner [6]: $\sigma ::= \forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.\tau$, where we write $\boldsymbol{\alpha}$ for a sequence of distinct type variables, $\alpha_1, \ldots, \alpha_n$ and similar for regions and effect variables. A type scheme binds type, effect, and region variables. A substitution extends naturally to a type scheme, renaming bound variables to avoid capture.

A type τ is an *instance* of a type scheme $\sigma = \forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.\tau'$ via substitution S_s, written $\tau \prec \sigma$ *via* S_s, if the domain of S_s is $\{\boldsymbol{\alpha}, \boldsymbol{\rho}, \boldsymbol{\epsilon}\}$ and $S_s(\tau') = \tau$.

A type environment, TE, is a finite map from expression variables to pairs of the form (σ, ρ).

Free variables of the above semantic objects are defined in the usual way. For a semantic object O, we write $ftv(O)$, $frv(O)$ and $fev(O)$ for the free type, region and effect variables of O respectively. If we need the union, we simply write $fv(O)$.

A typing judgement has the form $TE \vdash_{tt} e : \mu, \varphi$, which means that given a type environment TE, expression e has type μ and effect φ. The typing rules are defined by stripping off all constraint annotations from the rules in Fig. 3 (see also [20]), we do not repeat them here.

The *initial type environment* contains a type scheme for $\texttt{lift}$: $TE_i = \{\texttt{lift} \mapsto (\forall \rho_1, \rho_2 .((\texttt{int}, \rho_1) \xrightarrow{\epsilon.\{\rho_1, \rho_2\}} (\texttt{int}, \rho_2)), \rho)\}$. An expression e is a λ^{region}-*program* if it is typable in the initial environment, ie, $TE_i \vdash_{tt} e : (\tau, \rho), \varphi$. Lemma 5.2 from [20] guarantees that such a judgement is derivable if e stripped of the region annotations has an ML type.

2.2 Properties of $\boldsymbol{\lambda}^{\text{region}}$

In this section, we prove type soundness of evaluation for λ^{region}. The formal development follows Wright and Felleisen's general framework [21]. These results are special cases of more general results for the binding-time annotated λ^{region}-calculus in Section 5.

Proposition 1. (Subject reduction) *Suppose $TE \vdash_{tt} e : \mu, \varphi$. If $e \rightarrow_r e'$ then $TE \vdash_{tt} e' : \mu, \varphi'$ where $\varphi' \subseteq \varphi$.*

An erroneous term, $k \in \mathit{ErrTerm}$, is defined by

$$\begin{aligned} k ::= {} & \texttt{letregion}\ \rho\ \texttt{in}\ v \quad \textit{with } \rho \in frv(v) \mid (c\ \texttt{at}\ \rho) \ @\ v \mid \\ & (\texttt{lift}\ [\rho_1, \ldots, \rho_n]\ \texttt{at}\ \rho) \ @\ e \quad \textit{with } n \neq 2 \mid \\ & (\texttt{lift}\ [\rho_1, \rho_2]\ \texttt{at}\ \rho) \ @\ v \quad \textit{with } v \neq c\ \texttt{at}\ \rho_1 \mid \\ & (\Lambda[\rho_1, \ldots, \rho_n](x)\ \texttt{at}\ \rho = e)\ [\rho'_1, \ldots, \rho'_m]\ \texttt{at}\ \rho' \quad \textit{with } m \neq n \end{aligned}$$

Proposition 2. (Classification) *For each expression e either*

1. *e is a value v*
2. *e has the form $\mathcal{E}[r]$ where r is a redex*
3. *e has the form $\mathcal{E}[k]$ where k is erroneous*
4. *e has the form $\mathcal{E}[x]$ where $x \in \mathit{free}(e)$*
5. *e has the form $\mathcal{E}[f\ [\rho_1, \ldots, \rho_n]$ **at** $\rho]$ where $f \in \mathit{free}(e)$ and $f \neq$ **lift***

Proposition 3. (Progress) *If e is a λ^{region}-program then e must be a value or e has the form $\mathcal{E}[r]$ for a redex r.*

Theorem 1. Type Soundness *If e is a λ^{region}-program e, then either there exists a value v for which $e \twoheadrightarrow_r v$ or e diverges.*

3 Binding-time analysis

To specify binding-time constraints between regions, we require a tiny addition to λ^{region}:

$$\begin{aligned}\dot{e} ::= {} & x \mid c\ \texttt{at}\ \rho \mid \lambda\, x.\, \dot{e}\ \texttt{at}\ \rho \mid \dot{e}_1 \mathbin{@} \dot{e}_2 \mid \texttt{letrec}\ f\,[\rho_1, \ldots, \rho_n](x)\ \texttt{at}\ \rho = \dot{e}_1\ \texttt{in}\ \dot{e}_2 \mid \\ & \texttt{letregion}\ \rho : C\ \texttt{in}\ \dot{e} \mid \dot{\gamma}\,[\rho_1, \ldots, \rho_n]\ \texttt{at}\ \rho \mid \texttt{let}\ x = \dot{e}_1\ \texttt{in}\ \dot{e}_2 \mid\end{aligned}$$

$$\dot{\gamma} ::= f \mid \Lambda[\rho_1, \ldots, \rho_n](x)\ \texttt{at}\ \rho = \dot{e}$$

The only change is the addition of the constraint set C to the `letregion` expression. A *constraint set* is a set of pairs of region variables $\rho \leq \rho'$. Such constraints impose a *binding-time ordering* on regions. Intuitively, a constraint $\rho_1 \leq \rho_2$ means that the objects in region ρ_2 is not available earlier than the objects in region ρ_1. Constraints are necessary to guarantee sound specialization, as we shall see in Section 4 where we specify their meaning.

A constraint set C is *transitive closed* if, whenever $\rho' \leq \rho \in C$ and $\rho \leq \rho'' \in C$, then $\rho' \leq \rho'' \in C$. A constraint set C' is *consistent* with respect to a constraint set C (written $C \vdash C'$) if $C' \subseteq C$ and C is transitive closed.

In the constraint analysis a type scheme is now constrained by a constraint set C: $\forall\, \boldsymbol{\alpha}\,.\forall\, \boldsymbol{\rho}\,.\forall\, \boldsymbol{\epsilon}\,.C \Rightarrow \tau$. A region substitution extends naturally to constraint sets. Write $\mathit{frv}(C)$ for the set of region variables in C.

A type-based constraint analysis guides constraint generation by the type of an expression. Figure 2 specifies the well-formedness of a type with places with respect to a constraint set. The implied constraints are standard in binding-time analysis [7].

The following two projections apply to constraint sets:

$$C^{\uparrow\{\rho_1,\ldots,\rho_n\}} = \{\rho' \leq \rho'' \mid \exists i : \rho' = \rho_i \vee \rho' = \rho_i\} \qquad C_{\downarrow\{\rho_1,\ldots,\rho_n\}} = C \setminus C^{\uparrow\{\rho_1,\ldots,\rho_n\}}$$

Informally, $C_{\downarrow\{\rho_1,\ldots,\rho_n\}}$ removes all constraints mentioning some ρ_i, whereas $C^{\uparrow\{\rho_1,\ldots,\rho_n\}}$ keeps only constraints mentioning any of the ρ_i. We write $C^{\uparrow\rho}$ for $C^{\uparrow\{\rho\}}$ and $C_{\downarrow\rho}$ for $C_{\downarrow\{\rho\}}$.

$$C \vdash_{wft} (\mathtt{int}, \rho) \qquad \frac{C \vdash_{wft} \mu_1 \quad C \vdash_{wft} \mu_2 \qquad \mu_1 = (\tau_1, \rho_1) \quad \mu_2 = (\tau_2, \rho_2) \quad C \vdash \{\rho \leq \rho_1, \rho \leq \rho_2\}}{C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)}$$

Fig. 2. Well-formed types

Lemma 1. *If C is transitive closed then so is $C_{\downarrow\{\rho_1,\ldots,\rho_n\}}$.*

A *constraint problem* is a pair (C, R) where R is a set of region variables such that $frv(C) \subseteq R$.

Constraint analysis rules have the form $TE, C \vdash_c \dot{e} : \mu, \varphi$, which reads "given type environment TE and constraint set C, constraint annotated expression $\dot{e}$ has constraint-type μ and effect φ". The actual rules are given in figure 3.

Most rules are trivial extensions of the region inference rules of Tofte and Talpin. Rule *(letregion)* removes the constraints involving ρ in the type judgement when it is not free anymore. Also, the annotated set only keeps the regions involving ρ. Something similar is done in rule *(letrec)*, but then for the abstracted (quantified) variables. Rules *(recvar1)* and *(recvar2)* impose some additional constraints relative to the region of the region abstraction. They guarantee that the constructed lambda is available at the same time as the region abstraction and that none of the region arguments is available before the region abstraction.

Finally, we list a number of simple properties of judgements.

Lemma 2. *If $TE, C \vdash_c \dot{e} : \mu, \varphi$ then C is transitive closed.*

Lemma 3. *If $TE, C \vdash_c \dot{e} : (\tau, \rho), \varphi$, then $\rho \in (frv(\dot{e}) \cup frv(TE))$.*

The *initial constraint type environment*, TE_c, is defined as the map $\{\mathtt{lift} \mapsto (\forall \rho_1, \rho_2 .\{\rho_1 \leq \rho_2, \rho \leq \rho_1, \rho \leq \rho_2\} \Rightarrow ((\mathtt{int}, \rho_1) \xrightarrow{\epsilon.\{\rho_1,\rho_2\}} (\mathtt{int}, \rho_2)), \rho)\}$. An expression $\dot{e}$ is a *constraint-annotated program* if it is typable in TE_c, ie, $TE_c, C \vdash_c \dot{e} : (\tau, \rho), \varphi$. Moreover, by lemma 3, $\rho \in frv(\dot{e})$.

Lemma 4. *Suppose $TE_i \vdash_{tt} e : \mu, \varphi$. There exists C and $\dot{e}$ such that $|\dot{e}| = e$ (the erasure $\dot{e}$ removes the constraint set from the* **`letregion`** *expression) and $TE_c, C \vdash_c \dot{e} : \mu, \varphi$ and $frv(C) \subseteq frv(\dot{e}) \cup frv(TE_c)$.*

4 Region-based specialization

Binding-time analysis stages evaluation in two phases [11], by annotating each term with a binding time, b. It is either static (s), indicating a specialization-time action or dynamic (d), indicating a runtime action, ordered by $\mathsf{s} < \mathsf{d}$. For technical reasons, we include a symbolic binding time β which is unrelated to any b.

$$(const)\qquad \frac{C \vdash \emptyset}{TE, C \vdash_c c \;\mathtt{at}\; \rho : (\mathtt{int}, \rho), \{\rho\}}$$

$$(var)\qquad \frac{TE(x) = \mu \qquad C \vdash \emptyset}{TE, C \vdash_c x : \mu, \emptyset}$$

$$(abstr)\qquad \frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c \dot{e} : \mu_2, \varphi \\ C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho) \quad \varphi \subseteq \varphi' \end{array}}{TE, C \vdash_c \lambda\, x.\, \dot{e} \;\mathtt{at}\; \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}}$$

$$(app)\qquad \frac{TE, C \vdash_c \dot{e}_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \quad TE, C \vdash_c \dot{e}_2 : \mu_1, \varphi_2}{TE, C \vdash_c \dot{e}_1 \mathbin{@} \dot{e}_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

$$(let)\qquad \frac{\begin{array}{c} TE, C \vdash_c \dot{e}_1 : \mu_1, \varphi_1 \\ TE + \{x \mapsto \mu_1\}, C \vdash_c \dot{e}_2 : \mu_2, \varphi_2 \end{array}}{TE, C \vdash_c \mathtt{let}\; x = \dot{e}_1 \;\mathtt{in}\; \dot{e}_2 : \mu_2, \varphi_1 \cup \varphi_2}$$

$$(letregion)\qquad \frac{TE, C \vdash_c \dot{e} : \mu, \varphi \quad \rho \notin \mathit{frv}(TE, \mu)}{TE, C_{\downarrow\rho} \vdash_c \mathtt{letregion}\; \rho : C^{\uparrow\rho} \;\mathtt{in}\; \dot{e} : \mu, \varphi \setminus \{\rho\}}$$

$$(letrec)\qquad \frac{\begin{array}{c} \{\rho_1 \ldots, \rho_k, \epsilon_1, \ldots, \epsilon_m\} \cap \mathit{fv}(TE, \varphi) = \emptyset \\ R = \{\rho_1, \ldots, \rho_k\} \quad \hat{\sigma} = \forall\, \epsilon_1, \ldots, \epsilon_m . \forall\, \rho_1 \ldots \rho_k . C^{\uparrow R} \Rightarrow \tau \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \lambda\, x.\, \dot{e}_1 \;\mathtt{at}\; \rho : (\tau, \rho), \varphi \\ \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{ftv}(TE) = \emptyset \quad \sigma = \forall\, \alpha_1, \ldots, \alpha_n . \hat{\sigma} \\ TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow R} \vdash_c \dot{e}_2 : \mu', \varphi' \end{array}}{TE, C_{\downarrow R} \vdash_c \mathtt{letrec}\; f\,[\rho_1, \ldots, \rho_k](x) \;\mathtt{at}\; \rho = \dot{e}_1 \;\mathtt{in}\; \dot{e}_2 : \mu', \varphi'}$$

$$(recvar1)\qquad \frac{\begin{array}{c} TE(f) = (\sigma, \rho') \quad \sigma = \forall\, \boldsymbol{\alpha} . \forall\, \boldsymbol{\epsilon} . \forall\, \rho'_1, \ldots, \rho'_n . C' \Rightarrow \tau' \\ \tau \prec \sigma \;\; via \;\; (S_t, S_r, S_e) \quad S_r = \{\rho'_1 \mapsto \rho_1, \ldots, \rho'_n \mapsto \rho_n\} \\ C \vdash S_r(C') \cup \{\rho' \le \rho, \rho \le \rho', \rho' \le \rho_1, \ldots, \rho' \le \rho_n\} \end{array}}{TE, C \vdash_c f\; [\rho_1, \ldots, \rho_n] \;\mathtt{at}\; \rho : (\tau, \rho), \{\rho, \rho'\}}$$

$$(recvar2)\qquad \frac{\begin{array}{c} TE, C' \vdash_c \lambda\, x.\, \dot{e} \;\mathtt{at}\; \rho' : (\tau', \rho'), \{\rho'\} \\ S_r = \{\rho'_1 \mapsto \rho_1, \ldots, \rho'_n \mapsto \rho_n\} \quad \tau = S_r(\tau') \\ C \vdash S_r(C') \cup \{\rho' \le \rho, \rho \le \rho', \rho' \le \rho_1, \ldots, \rho' \le \rho_n\} \end{array}}{TE, C \vdash_c (\Lambda[\rho'_1, \ldots, \rho'_n](x) \;\mathtt{at}\; \rho' = \dot{e})\; [\rho_1, \ldots, \rho_n] \;\mathtt{at}\; \rho : (\tau, \rho), \{\rho, \rho'\}}$$

$$(effect)\qquad \frac{TE, C \vdash_c \dot{e} : \mu, \varphi \quad \epsilon \notin \mathit{fev}(TE, \mu)}{TE, C \vdash_c \dot{e} : \mu, \varphi \setminus \{\epsilon\}}$$

Fig. 3. Constraint analysis

The specializer works on an expression with annotated region variables:

$$
\begin{aligned}
E ::= {} & x \mid c \;\texttt{at}\; \rho^{\mathsf{b}} \mid \lambda\, x.\, E \;\texttt{at}\; \rho^{\mathsf{b}} \mid E_1 \mathbin{@} E_2 \mid \texttt{let}\; x = E_1 \;\texttt{in}\; E_2 \mid \\
& \texttt{letregion}\; \rho^{\mathsf{b}} : C \;\texttt{in}\; E \mid \texttt{letrec}\; f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x) \;\texttt{at}\; \rho_0^{\mathsf{d}} = E_1 \;\texttt{in}\; E_2 \mid \\
& \texttt{letrec}\; f\,[\rho_1^{\beta}, \ldots, \rho_n^{\beta}](x) \;\texttt{at}\; \rho_0^{\mathsf{b}_0} = E_1 \;\texttt{in}\; E_2 \mid \Gamma\,[\rho_1^{\mathsf{b}_1}, \ldots, \rho_n^{\mathsf{b}_n}] \;\texttt{at}\; \rho_0^{\mathsf{b}_0}
\end{aligned}
$$

$$
\Gamma ::= f \mid \Lambda[\rho_1^{\beta}, \ldots, \rho_n^{\beta}](x) \;\texttt{at}\; \rho_0^{\mathsf{b}_0} = E \mid \Lambda[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x) \;\texttt{at}\; \rho_0^{\mathsf{d}} = E
$$

We consider the language of constraint-annotated expressions from Section 3 as the subset of those E, in which all regions are annotated β.

A *static value*, $V \in SVal$ is defined by :

$$
V ::= c \;\texttt{at}\; \rho^{\mathsf{s}} \mid \lambda\, x.\, E \;\texttt{at}\; \rho^{\mathsf{s}} \mid \texttt{lift}\;[\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}] \;\texttt{at}\; \rho^{\mathsf{s}} \quad \textit{where}\; \mathsf{b}_i \neq \beta
$$

A specializer produces an *answer* A, which is either a *static answer*:

$$
\begin{aligned}
A_s ::= {} & V \mid \texttt{letregion}\; \rho^{\mathsf{d}} : C \;\texttt{in}\; A_s \mid \texttt{let}\; x = W \;\texttt{in}\; A_s \mid \\
& \texttt{letrec}\; f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x) \;\texttt{at}\; \rho^{\mathsf{d}} = W \;\texttt{in}\; A_s
\end{aligned}
$$

or a *residual term*:

$$
\begin{aligned}
W ::= {} & x \mid c \;\texttt{at}\; \rho^{\mathsf{d}} \mid \lambda\, x.\, W \;\texttt{at}\; \rho^{\mathsf{d}} \mid W_1 \mathbin{@} W_2 \mid \texttt{let}\; x = W_1 \;\texttt{in}\; W_2 \mid \\
& \texttt{letregion}\; \rho^{\mathsf{d}} : C \;\texttt{in}\; W \mid \Gamma_{\mathsf{d}}\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}] \;\texttt{at}\; \rho^{\mathsf{d}} \mid \\
& \texttt{letrec}\; f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x) \;\texttt{at}\; \rho^{\mathsf{d}} = W_1 \;\texttt{in}\; W_2
\end{aligned}
$$

$$
\Gamma_{\mathsf{d}} ::= f \mid \Lambda[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x) \;\texttt{at}\; \rho^{\mathsf{d}} = W
$$

A *model* of a constraint problem (C, R), written $\sigma \models (C, R)$, is a substitution $\sigma : R \to \{\mathsf{s}, \mathsf{d}\}$ such that $\sigma(\rho^{\mathsf{b}}) = \mathsf{b}$ if $\mathsf{b} \neq \beta$ and for all $\rho_1 \leq \rho_2 \in C$ it holds that $\sigma(\rho_1) \leq \sigma(\rho_2)$. If $\forall\, \rho_i^{\mathsf{b}_i} : \mathsf{b}_i \neq \beta$ and $\{\rho_i \mapsto \mathsf{b}_i\} \models (C, R)$, then we simply write $\models (C, R)$

Figures 4 and 5 specify the small-step specialization relation, $E \to_s E'$. As before, we write $E \twoheadrightarrow_s E'$ for the reflexive transitive closure.

Rules (1), (3), (5), (6) and (8) are similar to the standard semantics. Rule (2) reveals the purpose of `lift`: its argument is moved to a region with a potentially later binding time. In other words, a `lift` can *coerce* a static value into a dynamic one. Its effectiveness depends on the placement of `lift`s present in the original term. It is well-known how to place them to obtain an optimal result [9]. Rules (1)-(3) have primed companion rules which cater for the reduction of static answers which are no values. They move a potential redex inside the static answer to allow the specializer to continue. These rules are inspired by work on partial evaluation for call-by-value calculi [8, 13]. Rule (4) annotates a bound region: if all free variables have an annotation, then, by Lemma 4, all variables of C have annotations except ρ itself, which is assigned a binding-time consistent

(1) $(\lambda x.\, E \ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} A \rightarrow_s \mathtt{let}\ x = A\ \mathtt{in}\ E$

(1′) $(\mathtt{let}\ x = W\ \mathtt{in}\ A_s) \mathbin{@} A' \rightarrow_s \mathtt{let}\ x = W\ \mathtt{in}\ (A_s \mathbin{@} A')$

(1″) $(\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ A_s) \mathbin{@} A' \rightarrow_s$
$\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ (A_s \mathbin{@} A')$

(1‴) $(\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ A_s) \mathbin{@} A' \rightarrow_s$
$\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ (A_s \mathbin{@} A')$

(2) $$\frac{\mathsf{b}_1 \leq \mathsf{b}_2}{(\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} (c\ \mathtt{at}\ \rho_1^{\mathsf{b}_1}) \rightarrow_s c\ \mathtt{at}\ \rho_2^{\mathsf{b}_2}}$$

(2′) $(\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} (\mathtt{let}\ x = W\ \mathtt{in}\ A_s) \rightarrow_s$
$\mathtt{let}\ x = W\ \mathtt{in}\ ((\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} A_s)$

(2″) $(\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho_0^{\mathsf{s}}) \mathbin{@} (\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ A_s) \rightarrow_s$
$\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ ((\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho_0^{\mathsf{s}}) \mathbin{@} A_s)$ *where* $\rho \notin \{\rho_0, \rho_1, \rho_2\}$

(2‴) $(\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} (\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ A_s) \rightarrow_s$
$\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ ((\mathtt{lift}\ [\rho_1^{\mathsf{b}_1}, \rho_2^{\mathsf{b}_2}]\ \mathtt{at}\ \rho^{\mathsf{s}}) \mathbin{@} A_s)$

(3) $\mathtt{let}\ x = V\ \mathtt{in}\ E \rightarrow_s E[V \,/\, x]$

(3′) $\mathtt{let}\ x = (\mathtt{let}\ y = W\ \mathtt{in}\ A_s)\ \mathtt{in}\ E \rightarrow_s$
$\mathtt{let}\ y = W\ \mathtt{in}\ (\mathtt{let}\ x = A_s\ \mathtt{in}\ E)$

(3″) $\mathtt{let}\ x = (\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ A_s)\ \mathtt{in}\ E \rightarrow_s$
$\mathtt{letregion}\ \rho^{\mathsf{d}} : C\ \mathtt{in}\ (\mathtt{let}\ x = A_s\ \mathtt{in}\ E)$

(3‴) $\mathtt{let}\ x = (\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ A_s)\ \mathtt{in}\ E \rightarrow_s$
$\mathtt{letrec}\ f\,[\rho_1^{\mathsf{d}}, \ldots, \rho_n^{\mathsf{d}}](x)\ \mathtt{at}\ \rho^{\mathsf{d}} = W\ \mathtt{in}\ (\mathtt{let}\ x = A_s\ \mathtt{in}\ E)$

(4) $$\frac{\sigma \models (C, \{\rho\}) \quad \sigma(\rho) = \mathsf{b}}{\mathtt{letregion}\ \rho^{\beta} : C\ \mathtt{in}\ E \rightarrow_s \mathtt{letregion}\ \rho^{\mathsf{b}} : \sigma(C)\ \mathtt{in}\ E[\rho^{\mathsf{b}} \,/\, \rho^{\beta}]}$$

(5) $\mathtt{letregion}\ \rho^{\mathsf{s}} : C\ \mathtt{in}\ A \rightarrow_s A \quad$ *if* $\rho^{\mathsf{s}} \notin \mathit{frv}(A)$

(6) $(\Lambda[\rho_1^{\beta}, \ldots, \rho_n^{\beta}](x)\ \mathtt{at}\ \rho^{\mathsf{s}} = E)\ [{\rho'_1}^{\mathsf{b}_1}, \ldots, {\rho'_n}^{\mathsf{b}_n}]\ \mathtt{at}\ \rho_0^{\mathsf{s}} \rightarrow_s$
$\lambda x.\, E[{\rho'_1}^{\mathsf{b}_1} \,/\, \rho_1^{\beta}, \ldots, {\rho'_n}^{\mathsf{b}_n} \,/\, \rho_n^{\beta}]\ \mathtt{at}\ \rho_0^{\mathsf{s}}$

Fig. 4. Specialization Semantics - part 1

$$(7)\quad \begin{array}{l}(\Lambda[\rho_1^\beta,\ldots,\rho_n^\beta](x)\ \texttt{at}\ \rho^{\mathsf{d}} = E)\ [{\rho'}_1^{\mathsf{d}},\ldots,{\rho'}_n^{\mathsf{d}}]\ \texttt{at}\ {\rho'}^{\mathsf{d}} \rightarrow_s \\ (\Lambda[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = E[\rho_1^{\mathsf{d}} / \rho_1^\beta,\ldots,\rho_n^{\mathsf{d}} / \rho_n^\beta])\ [{\rho'}_1^{\mathsf{d}},\ldots,{\rho'}_n^{\mathsf{d}}]\ \texttt{at}\ {\rho'}^{\mathsf{d}}\end{array}$$

$$(8)\quad \begin{array}{l}\texttt{letrec}\ f\,[\rho_1^\beta,\ldots,\rho_n^\beta](x)\ \texttt{at}\ \rho^{\mathsf{s}} = E_1\ \texttt{in}\ E_2 \rightarrow_s \\ E_2[(\Lambda[\rho_1^\beta,\ldots,\rho_n^\beta](x)\ \texttt{at}\ \rho^{\mathsf{s}} = (\texttt{letrec}\ f\,[\rho_1^\beta,\ldots,\rho_n^\beta](x)\ \texttt{at}\ \rho^{\mathsf{s}} = E_1\ \texttt{in}\ E_1)) / f]\end{array}$$

$$(9)\quad \begin{array}{l}\texttt{letrec}\ f\,[\rho_1^\beta,\ldots,\rho_n^\beta](x)\ \texttt{at}\ \rho^{\mathsf{d}} = E_1\ \texttt{in}\ E_2 \rightarrow_s \\ \texttt{letrec}\ f\,[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = E_1[\rho_1^{\mathsf{d}} / \rho_1^\beta,\ldots,\rho_n^{\mathsf{d}} / \rho_n^\beta]\ \texttt{in}\ E_2\end{array}$$

$$(10)\quad \frac{E \rightarrow_s E'}{\mathcal{S}[E] \rightarrow_s \mathcal{S}[E']}$$

where

$$\begin{array}{rl}\mathcal{S} ::= & [\,] \mid \mathcal{S} \mathbin{@} E \mid A \mathbin{@} \mathcal{S} \mid \texttt{let}\ x = \mathcal{S}\ \texttt{in}\ E \mid \texttt{letregion}\ \rho^{\mathsf{b}} : C\ \texttt{in}\ \mathcal{S} \mid \\ & \texttt{let}\ x = W\ \texttt{in}\ \mathcal{S} \mid \texttt{letrec}\ f\,[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = \mathcal{S}\ \texttt{in}\ E \mid \\ & \texttt{letrec}\ f\,[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = W\ \texttt{in}\ \mathcal{S} \mid \lambda x.\, \mathcal{S}\ \texttt{at}\ \rho^{\mathsf{d}} \mid \\ & (\Lambda[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = \mathcal{S})\ [{\rho'}_1^{\mathsf{d}},\ldots,{\rho'}_n^{\mathsf{d}}]\ \texttt{at}\ {\rho'}^{\mathsf{d}}\end{array}$$

Fig. 5. Specialization Semantics - part 2

with the constraint set C. Rules (7) and (9) generate code for dynamic region abstractions and `letrec`s, respectively. Finally, rule (10) extends specialization to specialization contexts. Since specialization reduces inside dynamic lambdas it may not terminate, even though the original program did.

The *constraint erasure* of an annotated expression E (written $|E|_c$) is the constraint-annotated expression $\dot{e}$ obtained by removing all binding-time annotations from E. The *complete erasure* of a annotated expression E (written $|E|$) is the λ^{region}-expression e obtained by removing all binding-time annotations and constraints from E.

If $TE_c, C \vdash_c |E|_c : \mu, \varphi$ and $\models (C, \mathit{frv}(E))$, then E is an *annotated program.* If $\mu = (\tau, \rho)$ and $\rho^{\mathsf{d}} \in \mathit{frv}(E)$, then E is a *binding-time completion.* For every constraint-annotated term, there exists a binding-time completion where all free region variables are annotated d.

5 Correctness

Lemma 5. *If* $TE + \{x \mapsto (\tau', \rho')\}, C \vdash_c \dot{e} : (\tau, \rho), \varphi$ *and* $TE, C \vdash_c v : (\tau', \rho'), \varphi'$, *then* $TE, C \vdash_c \dot{e}[v / x] : (\tau, \rho), \varphi$.

Proof. Induction on the proof tree and case analysis on the conclusion. Analogous to Lemma 4.5 in [21].

Lemma 6. *Given a substitution $S_s = (S_t, S_r, S_e)$ with $S_r = [\,]$, it holds that if $TE, C \vdash_c \dot{e} : (\tau, \rho), \varphi$, then also $S_s(TE), C \vdash_c \dot{e} : (S_s(\tau), \rho), S_s(\varphi)$*

Proof. Induction on the length of the proof of $TE, C \vdash_c \dot{e} : (\tau, \rho), \varphi$ and case analysis of the last step. It is an adaptation of the proof of lemma 4.2 in [17] for constraint analysis derivations.

Lemma 7. *If $TE + \{f \mapsto (\forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.C^{\uparrow\boldsymbol{\rho}} \Rightarrow \tau', \rho')\}, C_{\downarrow\boldsymbol{\rho}} \vdash_c \dot{e} : (\tau, \rho), \varphi$ and $\{\boldsymbol{\alpha}, \boldsymbol{\epsilon}, \boldsymbol{\rho}\} \cap fv(TE, \varphi, C) = \emptyset$ and also $TE + \{f \mapsto (\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.C^{\uparrow\boldsymbol{\rho}} \Rightarrow \tau', \rho')\}, C \vdash_c \lambda x.\, \dot{e}'$ **at** $\rho' : (\tau', \rho'), \{\rho'\}$, then it holds that $TE + \{f \mapsto (\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.C^{\uparrow\boldsymbol{\rho}} \Rightarrow \tau', \rho')\}, C_{\downarrow\boldsymbol{\rho}} \vdash_c \dot{e}[(\Lambda[\boldsymbol{\rho}](x)$ **at** $\rho' = \dot{e}') \,/\, f] : (\tau, \rho), \varphi$*

Proof. Induction on the length of the derivation for $TE + \{f \mapsto (\forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \boldsymbol{\rho}.C^{\uparrow\boldsymbol{\rho}} \Rightarrow \tau', \rho')\}, C_{\downarrow\boldsymbol{\rho}} \vdash_c \dot{e} : (\tau, \rho), \varphi$ and case analysis on the conclusion. Most cases are quite similar to those of Lemma 4.4 in [21]. The only slightly deviating case is for $\dot{e} = f\ [\rho_1', \ldots, \rho_n']$ `at` ρ, which can be proven by Lemma 7 and rules *(recvar1)* and *(recvar2)* from Figure 3. □

The following lemma shows that if we take the projections of a transitive closed constraint set C with respect to some variable ρ then the solutions of the projections compose to a solution of C.

Lemma 8. *Let C be transitive closed and let $\rho \in R$. If $\sigma_1 \models (C_{\downarrow\rho}, R \setminus \rho)$ then there exists some σ_2 so that $\sigma_2 \models (\sigma_1(C)^{\uparrow\rho}, \{\rho\})$ and $\sigma_2 \circ \sigma_1 \models (C, R)$.*

If we are only interested in the minimal solution, then we could refine $C^{\uparrow\{\rho_1, \ldots, \rho_n\}}$ to $\{\rho' \leq \rho'' \mid \exists i : \rho_i = \rho''\}$. In this case, σ_2 simplifies to the least upper bound of the values on the left of $\leq$.

Proposition 4. (Subject reduction) *Let $TE, C \vdash_c |E|_c : \mu, \varphi$, $TE(\textbf{\textit{lift}}) = TE_c(\textbf{\textit{lift}})$ and $\models (C, frv(E))$. If $E \to_s E'$ then $TE, C' \vdash_c |E'|_c : \mu, \varphi'$ with $\varphi' \subseteq \varphi$ and $C \subseteq C'$. Moreover, $\models (C', frv(E'))$.*

Proof. Induction on the length of a specialization and case analysis for rules (1) to (9) in figures 4 and 5. Most cases are straightforward. Lemma 5 is needed for specialization rule (3), Lemma 7 for rule (8) and Lemma 8 for rule (5).

A annotated erroneous term, $K \in$ *2ErrTerm*, is defined by

$$
\begin{aligned}
K ::= {} & \texttt{letregion}\ \rho^{\beta} : C\ \texttt{in}\ E \quad \textit{with}\ \nexists \sigma : \sigma \models (C, \{\rho\}) \mid \\
& \texttt{letregion}\ \rho^{\mathsf{S}} : C\ \texttt{in}\ A \quad \textit{with}\ \rho^{\mathsf{S}} \in frv(A) \mid \quad c\ \texttt{at}\ \rho^{\beta} \mid \\
& \texttt{lift}\ [\rho_1^{\mathsf{b}_1}, \ldots, \rho_n^{\mathsf{b}_n}]\ \texttt{at}\ \rho_0^{\mathsf{b}_0} \quad \textit{with}\ (n \neq 2) \vee (\exists\, i \in \{0, \ldots, n\} : \mathsf{b}_i = \beta) \mid \\
& (\texttt{lift}\ [\rho_1^{\mathsf{S}}, \rho_2^{\mathsf{b}_2}]\ \texttt{at}\ \rho^{\mathsf{S}})\ @\ V \quad \textit{with}\ V \neq c\ \texttt{at}\ \rho_1^{\mathsf{S}} \mid \quad W\ @\ A_s \mid \\
& (\texttt{lift}\ [\rho_1^{\mathsf{d}}, \rho_2^{\mathsf{d}}]\ \texttt{at}\ \rho^{\mathsf{S}})\ @\ W \quad \textit{with}\ W \neq c\ \texttt{at}\ \rho_1^{\mathsf{d}} \mid \quad (c\ \texttt{at}\ \rho^{\mathsf{S}})\ @\ A \mid \\
& (\texttt{lift}\ [\rho_1^{\mathsf{d}}, \rho_2^{\mathsf{S}}]\ \texttt{at}\ \rho^{\mathsf{S}})\ @\ A \mid \quad \lambda\, x.\, E\ \texttt{at}\ \rho^{\beta} \mid \quad \lambda\, x.\, A_s\ \texttt{at}\ \rho^{\mathsf{d}} \mid \\
& \texttt{letrec}\ f\, [\rho_1^{\beta}, \ldots, \rho_n^{\beta}](x)\ \texttt{at}\ \rho^{\beta} = E_1\ \texttt{in}\ E_2 \mid
\end{aligned}
$$

$$
\begin{array}{l}
\texttt{letrec}\ f\,[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = A_s\ \texttt{in}\ E \mid \\
(\Lambda[\rho_1^{\mathsf{b}_1},\ldots,\rho_n^{\mathsf{b}_n}](x)\ \texttt{at}\ \rho_0^{\mathsf{b}_0} = E)\ [{\rho'}_1^{\mathsf{b}'_1},\ldots,{\rho'}_m^{\mathsf{b}'_m}]\ \texttt{at}\ {\rho_0}^{\mathsf{b}'_0} \\
\textit{with}\ (m \neq n) \vee (\exists\, i \in \{0,\ldots,n\} : \mathsf{b}'_i = \beta) \mid \\
(\Lambda[\rho_1^{\mathsf{b}_1},\ldots,\rho_n^{\mathsf{b}_n}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = E)\ [{\rho'}_1^{\mathsf{b}'_1},\ldots,{\rho'}_n^{\mathsf{b}'_n}]\ \texttt{at}\ {\rho_0}^{\mathsf{b}'_0} \\
\textit{with}\ \exists\, i : \mathsf{b}'_i \neq \mathsf{d} \mid \quad (\Lambda[\rho_1^{\mathsf{d}},\ldots,\rho_n^{\mathsf{d}}](x)\ \texttt{at}\ \rho^{\mathsf{d}} = A_s)\ [{\rho'}_1^{\mathsf{d}},\ldots,{\rho'}_n^{\mathsf{d}}]\ \texttt{at}\ {\rho_0}^{\mathsf{d}} \mid \\
(\Lambda[\rho_1^{\beta},\ldots,\rho_n^{\beta}](x)\ \texttt{at}\ \rho^{\mathsf{s}} = E)\ [{\rho'}_1^{\mathsf{b}'_1},\ldots,{\rho'}_n^{\mathsf{b}'_n}]\ \texttt{at}\ {\rho_0}^{\mathsf{d}}
\end{array}
$$

Proposition 5. (Classification) *For each annotated expression E either*

1. *E is an answer A*
2. *E has the form $\mathcal{S}[R]$ where R is a specialization-redex*
3. *E has the form $\mathcal{S}[K]$ where K is erroneous*
4. *E has the form $\mathcal{S}[x]$ where $x \in \textit{free}(E)$*
5. *E has the form $\mathcal{S}[f\ [\rho_1^{\beta},\ldots,\rho_n^{\beta}]\ \textbf{\textit{at}}\ \rho_0^{\mathsf{b}_0}]$ where $f \in \textit{free}(E)$ and $f \neq \textbf{\textit{lift}}$*

Proposition 6. (Progress) *If E is an annotated program then E must be an answer or E has the form $\mathcal{E}[R]$ for a redex R.*

Proof. By definition of program and specialization context, it suffices to show that if E is erroneous either $TE_c, C \vdash_c |E|_c : \mu, \varphi$ does not hold or $\not\models(C, \textit{frv}(E))$. This is done by case analysis on the grammar for K. For the first term of K, one needs lemma 8. The other cases are easy to check by looking at the type rules and inspecting the generated constraint sets. □

The soundness of the binding-time analysis is the annotated equivalent for type soundness of the standard type system.

Theorem 2. (Soundness of the Binding-time Analysis) *If E is a annotated program, then either there exists an answer A for which $E \twoheadrightarrow_s A$ or specialization does not terminate.*

Proof. Immediate consequence of propositions 4 and 6.

We expect the specializer to produce a completely residual program when properly annotated.

Theorem 3. (Residual Code) *Suppose E is a binding-time completion and $E \twoheadrightarrow_s A$ for A an answer, then A is residual code, ie, of the form W.*

Proof. Take $\mu = (\tau, \rho)$. Then ρ has binding time d since E is a binding-time completion. By subject reduction (Proposition 4) we have $TE_c, C \vdash_c |A|_c : \mu, \varphi$. Now, suppose $A = A_s$, then $\rho \in \textit{frv}(|A_s|_c)$ and has static binding time. This is a contradiction, so A can only be of the form W.

At last, we consider the soundness of the specialization itself. We take this to mean that all specialization-time reductions are observational equivalences with respect to ordinary evaluation $\twoheadrightarrow_r$. More specifically, we consider the relation $\Rightarrow$ on λ^{region}-expressions defined by $e \Rightarrow e'$ if $E \rightarrow_s E'$ and $e = |E|$ and $e' = |E'|$. Let $\Rightarrow^*$ be the reflexive and transitive closure of $\Rightarrow$.

Theorem 4. (Soundness of the specializer) *If $e_1 \twoheadrightarrow_r e_2$ and $e_1 \Rightarrow^* e_1'$ then there exists e_2' such that $e_2 \Rightarrow^* e_2'$ and $e_1' \twoheadrightarrow_r e_2'$.*

Proof. Tedious induction on the definition of $\Rightarrow^*$ and on $e_1 \twoheadrightarrow_r e_2$.

Since specialization $\twoheadrightarrow_s$ is a subset of $\Rightarrow^*$ (ignoring the annotations), we obtain the following corollary.

Corollary 1. *Let $e = |E|$. If $e \twoheadrightarrow_r (c \;\textbf{\textit{at}}\; \rho)$ and $E \twoheadrightarrow_s A$ then $|A| \twoheadrightarrow_r (c \;\textbf{\textit{at}}\; \rho)$.*

6 Example

We now give a small example which illustrates how polymorphism in the type language is specialized using polymorphic binding times via region polymorphism. Consider the following expression without region annotations:

$$
\begin{array}{l}
\lambda d.\, \texttt{letrec}\ \mathit{apply}\ (f) = \lambda x.\, f \,@\, x\ \texttt{in} \\
\quad \texttt{let}\ y = (\mathit{apply} \,@\, (\lambda n.\, 5)) \,@\, (\texttt{lift} \,@\, 3)\ \texttt{in} \\
\quad \texttt{let}\ z = (\mathit{apply} \,@\, (\lambda n.\, n)) \,@\, (\texttt{lift} \,@\, d)\ \texttt{in} \\
\quad\quad \mathit{if}\ z\ \texttt{then}\ (\texttt{lift} \,@\, y)\ \texttt{else}\ (\texttt{lift} \,@\, 0)
\end{array}
$$

The variable d is free and the `if` construct is standard. Below, we give a binding-time completion, but for clarity, we leave out the constraints and add the binding times to the regions without specializing:

$$
\begin{array}{l}
\lambda d.(\texttt{letregion}\ \rho_4^{\mathsf{s}}, \rho_5^{\mathsf{s}}, \rho_{18}^{\mathsf{d}}, \rho_{21}^{\mathsf{s}}, \rho_{22}^{\mathsf{s}}\ \texttt{in} \\
\qquad \texttt{letrec}\ \mathit{apply}[\rho_6, \rho_7, \rho_8, \rho_9](f)\ \texttt{at}\ \rho_5^{\mathsf{s}} = \lambda x.\, (f \,@\, x)\ \texttt{at}\ \rho_6\ \texttt{in} \\
\qquad\quad \texttt{let}\ y = \texttt{letregion}\ \rho_{10}^{\mathsf{s}}, \rho_{11}^{\mathsf{s}}, \rho_{12}^{\mathsf{s}}\ \texttt{in} \\
\qquad\qquad\qquad (\texttt{letregion}\ \rho_{13}^{\mathsf{s}}\ \texttt{in} \\
\qquad\qquad\qquad\quad (\mathit{apply}\ [\rho_{10}^{\mathsf{s}}, \rho_{11}^{\mathsf{s}}, \rho_4^{\mathsf{s}}, \rho_{12}^{\mathsf{s}}]\ \texttt{at}\ \rho_{13}^{\mathsf{s}}) \\
\qquad\qquad\qquad\quad @\, ((\lambda n.\, 5\ \texttt{at}\ \rho_4^{\mathsf{s}})\ \texttt{at}\ \rho_{11}^{\mathsf{s}})) \\
\qquad\qquad\qquad @\ \texttt{letregion}\ \rho_{14}^{\mathsf{s}}, \rho_{19}^{\mathsf{s}}\ \texttt{in} \\
\qquad\qquad\qquad\quad (\texttt{lift}\ [\rho_{14}^{\mathsf{s}}, \rho_{12}^{\mathsf{s}}]\ \texttt{at}\ \rho_{19}^{\mathsf{s}}) \,@\, (3\ \texttt{at}\ \rho_{14}^{\mathsf{s}}) \\
\qquad\quad \texttt{in}\ \ \texttt{let}\ z = \texttt{letregion}\ \rho_{15}^{\mathsf{s}}, \rho_{16}^{\mathsf{s}}\ \texttt{in} \\
\qquad\qquad\qquad\quad (\texttt{letregion}\ \rho_{17}\mathsf{s}\ \texttt{in} \\
\qquad\qquad\qquad\qquad (\mathit{apply}\ [\rho_{15}^{\mathsf{s}}, \rho_{16}^{\mathsf{s}}, \rho_2^{\mathsf{d}}, \rho_2^{\mathsf{d}}]\ \texttt{at}\ \rho_{17}^{\mathsf{s}}) \\
\qquad\qquad\qquad\qquad @\, (\lambda m.\, m\ \texttt{at}\ \rho_{10}^{\mathsf{s}})) \\
\qquad\qquad\qquad\quad @\ \texttt{letregion}\ \rho_{20}^{\mathsf{s}}\ \texttt{in} \\
\qquad\qquad\qquad\qquad (\texttt{lift}\ [\rho_2^{\mathsf{d}}, \rho_{18}^{\mathsf{d}}]\ \texttt{at}\ \rho_{20}^{\mathsf{s}}) \,@\, d \\
\qquad\qquad\quad \texttt{in}\ \ \texttt{if}\ z\ \texttt{then}\ (\texttt{lift}\ [\rho_4^{\mathsf{s}}, \rho_3^{\mathsf{d}}]\ \texttt{at}\ \rho_{21}^{\mathsf{s}}) \,@\, y \\
\qquad\qquad\qquad \texttt{else}\ (\texttt{lift}\ [\rho_4^{\mathsf{s}}, \rho_3^{\mathsf{d}}]\ \texttt{at}\ \rho_{22}^{\mathsf{s}}) \,@\, (0\ \texttt{at}\ \rho^{\mathsf{s}}))\ \texttt{at}\ \rho_4^{\mathsf{d}}
\end{array}
$$

The above expression reduces to $\lambda d.\, (\texttt{if}\ \ d\ \texttt{then}\ 5\ \texttt{else}\ 0)$. To our knowledge no existing partial evaluation framework can obtain this result.

7 Conclusion and future work

We have established a formal framework for a program specializer which can handle polymorphically typed programming languages. Polymorphism in the region calculus naturally induces polymorphic binding times, giving polymorphic recursion for free. The connection between regions and binding times does not come as a surprise: memory management based on region inference [20] tries to separate regions as much as possible (to allow more aggressive memory recycling). This coincides with the desire to separate binding times when specializing.

Previous use of regions in program specialization was targeted towards the handling of side-effects at specialization time [16]. It is one of our main objectives to integrate this technique into our new scheme.

Although this paper focussed on soundness properties of the specializer, extracting an algorithm from the specification is not difficult due to the clear separation between region inference, constraint generation, and specialization. In fact, we are currently working on a partial evaluator for the Standard ML [14] programming language, based on the framework of this paper.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In Alex Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, USA, January 1999. ACM Press.
2. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proc. of the 14th Annual IEEE symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Computer Society Press.
3. Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, to appear, 1999.
4. Charles Consel. Polyvariant binding-time analysis for applicative languages. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
5. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
6. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.
7. 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.
8. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

9. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472, Cambridge, MA, 1991. Springer-Verlag.
10. Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.
11. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
12. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
13. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.
14. Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
15. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.
16. Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: http://www.elsevier.nl/locate/entcs/volume6.html.
17. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
18. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
19. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.
20. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.