Polymorphic Specialization for ML

Simon Helsen and Peter Thiemann

University of Freiburg*, Germany

Abstract. We present a novel technique for offline partial evaluation of functional languages with an ML-style typing discipline. Our program specialization method comprises a polymorphic binding-time analysis with polymorphic recursion. Based on the region calculus of Tofte and Talpin, we develop a binding-time analysis as a constraint analysis on top of region inference. Our insight is to regard binding times as properties of regions.

Specialization is specified as a small-step semantics, extending previous work on syntactic type soundness results for the region calculus [28]. Using similar syntactic proof techniques, we prove soundness of the binding-time analysis with respect to the specializer. In addition, we prove that specialization preserves the call-by-value semantics of the region calculus by showing that the reductions of the specializer are contextual equivalences in the region calculus [26].

Key Words: regions, binding-time analysis, specialization

1 Introduction

1.1 Offline Partial Evaluation

Partial evaluation [13, 33] is a well-explored program specialization technique, which factorizes the evaluation of a program in two phases. In the *static* or *specialization* phase, all operations depending only on known data are executed. The result of the specialization phase is a *residual program*. In the *dynamic* or *runtime* phase, this residual program is executed to obtain the actual results. Usually, the runtime phase is faster than standard evaluation, thereby improving overall speed if the residual program is executed several times.

Offline partial evaluation is a flavor of program specialization where the specialization phase is preceded by a *binding-time analysis* or BTA [29, 43, 53]. A BTA is a program analysis that determines what constructs in a program can be executed at specialization time. It yields an annotation of each construct with a binding time indicating in which phase it is to be executed. A binding-time annotated expression is often called a *two-level* term [44] and a binding-time analysis can be understood as the static semantics[1] for offline program specialization.

* Institut für Informatik, Georges-Köhler-Allee Geb. 079, D-79110 Freiburg, Germany. Tel: +49-7612038053, Fax: +49-7612038052, e-mail: shelsen@computer.org

[1] The term "static semantics" is not to be confused with "static phase". In fact, the dynamic semantics of offline partial evaluation coincides with the static or specialization phase

A BTA that assigns each construct a fixed binding time is a *monovariant* analysis. Many existing partial evaluators rely on a monovariant BTA [9, 29, 43] and many applications have been studied in the context of these systems. However, such a BTA 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 that depends on the context of each particular call of a function, say. This is basically a *polyvariant* analysis [10]. Several applications benefit from binding-time polyvariance, for instance modular program specialization [15], separate compilation via specialization [24, 27] and enforcing security properties by specialization [54]. However, in its most general form, 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 a *polymorphic* binding-time analysis which transfers the idea of parametric polymorphism from types to binding times. Previous results by Dussart, Henglein and Mossin [16, 30] yield such an analysis for a simply typed lambda calculus. The analysis itself is binding-time polymorphic and admits polymorphic recursion on the level of binding times. Further, it employs a notion of subtyping (where only the binding times change, but not the structure of the type itself). Dependencies between as yet unknown binding times are expressed by means of qualified type schemes.

In this work, we depart from the ad-hoc approach and design a polymorphic binding-time analysis as an extension of *region inference*.

1.2 The Region Calculus

A region calculus is an annotated polymorphically typed lambda calculus that makes memory allocation and deallocation explicit using regions [18, 34, 49, 60]. A prominent example is the region calculus by Tofte and Talpin [61], which we will refer to as λ^{region}. This calculus has been successfully used to implement a compiler for Standard ML [40] without trace-based garbage collection [59], and forms the basis for the offline partial evaluator of this paper.

A region is a part of memory of potentially unbounded size. In λ^{region}, regions are allocated and deallocated as a whole in a stack-like manner. Region inference [3, 57, 58] translates an ML-typeable program into a λ^{region}-expression. The following grammar defines its terms:

$$\begin{aligned}\mathsf{RegionTerm} \ni e ::= {} & x \mid c \;\texttt{at}\; \varrho \mid \lambda x.\, e \;\texttt{at}\; \varrho \mid e \,@\, e \mid \texttt{lift}\,[\varrho_1, \varrho_2]\, e \mid \\ & \Lambda\, \varrho.\, e \mid \texttt{let}\; x = e \;\texttt{in}\; e \mid f\,[\varrho_1, \ldots, \varrho_n] \;\texttt{at}\; \varrho \mid \\ & \texttt{letrec}\; f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, e \;\texttt{at}\; \varrho \;\texttt{in}\; e\end{aligned}$$

Constants and lambda expressions are values and the annotation $\texttt{at}\;\varrho$ indicates that the object is to be allocated in the region bound to ϱ. Regions are introduced through a region abstraction $\Lambda\, \varrho.\, e$. Evaluation of this expression first allocates a new region and binds it to ϱ. Then it determines the value of the expression e and, finally, it deallocates the region. The `lift`-construct is a canonical primitive, the identity function moving its argument (of base type) in ϱ_1 to the region ϱ_2. We will extend its meaning in section 3 where we discuss a specialization semantics for an extension of λ^{region}. The `letrec`-construct recursively binds a function f that additionally abstracts over regions $\varrho_1, \ldots, \varrho_n$. The expression $f\ [\varrho_1, \ldots, \varrho_n]$ `at` ϱ instantiates such a region abstraction and stores the resulting function object in ϱ. A region variable is bound if it occurs in the lexical scope of a region abstraction. An expression without free expression variables is called a *program*. Programs typically have free region variables, for instance, the initial memory region in which the program stores its result.

1.3 Contribution

We specify a binding-time analysis and specialization semantics for a call-by-value lambda calculus with ML-style polymorphic types. The analysis supports polymorphism over binding times and allows for polymorphic recursion, that is, the binding times of recursive calls of a function may be different at each call.

Our binding-time analysis is built as a constraint analysis on top of the region type system. There are several advantages to this approach. First, it allows for a clean separation between the underlying flow analysis and the additional well-formedness requirements, which is a desirable factorization for a BTA [53]. Secondly, the flow analysis part of our BTA is almost as accurate as the powerful data dependency analysis obtained by region inference. Moreover, since we can recycle the polynomial time region reconstruction algorithms by Birkedal and Tofte [3, 57], it is only necessary to design and implement a constraint analysis algorithm on top of region inference.

The result of the constraint analysis is a constraint-annotated λ^{region}-term, where constraints express binding-time dependencies between regions, containing actual runtime objects. Since region reconstruction with polymorphic recursion at the level of region variables is decidable, our approach implements a binding-time analysis with a high degree of binding-time polyvariance while guaranteeing the finiteness of binding-time descriptions [10]. Our technique builds on a polymorphically typed language, thus extending previous work that builds on the simply typed lambda calculus [16, 30]. Considering binding times as properties of regions leads to a novel and perhaps unexpected use of the region calculus.

Specialization is described with a Plotkin-style structural operational semantics [45]. Soundness of the constraint analysis and correctness of the specializer are proven by operational methods, similar to the approach of Hatcliff and Danvy for Moggi's computational meta-language [22]. Building on previous work [7,28], we use syntactic type soundness techniques [20,62] to show soundness of the constraint analysis with respect to the specialization semantics. We also prove that the specialization semantics is correct with respect to the standard semantics of the region calculus. We do so by showing that each reduction step taken by the specializer is in fact a contextual equivalence in the region calculus. To achieve this, we use an equational theory from earlier work [26].

1.4 Related Work

Dussart, Henglein and Mossin [16,30] specify a polymorphic binding-time analysis with polymorphic recursion as an ad-hoc extension of a monomorphic analysis. Their framework builds on top of a simply-typed lambda calculus. It includes a powerful notion of binding-time subtyping. Finally, the work contains constraint simplification rules and a polynomial-time fix-point algorithm. They do not have correctness results for the analysis (except for the first-order case [30], which unfortunately has some flaws). In contrast, our base language is polymorphically typed but we do not use subtyping since it may lead to code duplication.

Heldal and Hughes [23,25] extend the work by Dussart, Henglein and Mossin by considering a polymorphic base language and by representing coercions in their type language. While our work does not address subtyping and coercions, it provides a correctness proof of the BTA and specialization.

Glynn, Stuckey, Sulzmann and Søndergaard [19] generalize a polymorphic binding-time analysis to polymorphic programs, by encoding binding-time subtyping constraints as boolean formula. This is done within their HM(X)-framework, in order to recycle fast boolean constraint solvers. Their approach lacks a corresponding specialization semantics, thereby questioning the usefulness and correctness of such a BTA.

Thiemann [52] considers another variant of the region calculus [49] as the basis of a binding-time analysis for ML with references. In that work, the emphasis was on dealing with the imperative aspects of ML. The analysis is monovariant and the correctness proof is indirect through a translation to a pure lambda calculus.

The Dependency Core Calculus of Abadi and others [1] can be specialized to a monomorphic binding-time analysis. Continuing that work, Banerjee and others [2] define a polymorphic lambda calculus that provides a denotational proof for the soundness of the region calculus. However, they do not give a direct connection between binding times and regions.

Our syntactic soundness proof for the binding-time analysis is based on previous work on type soundness for the region calculus [7,28]. These papers discuss other syntactic approaches to the type soundness result. The present work adds the handling of constraints to the type system and binding times to the operational semantics.

Mogensen [41] was the first to specify a binding-time analysis for a language with ML-style polymorphism. However, except for the polyvariance introduced by type-polymorphism, *i.e.* different type instantiations allow for different binding-time descriptions, the analysis is monovariant. Also, Mogensen's binding-time analysis is not directly related (or proven correct) to an actual specializer.

The first use of effects in a binding-time analysis seems to be in a paper by Consel and others [14]. Starting from the simply typed lambda calculus, they annotate function arrows with boolean functions that determine the binding time of the result of the function from the binding times of the function's parameters. They prove a result comparable with subject reduction, give an inference algorithm and prove its soundness and completeness. The paper uses binding-time polyvariance to allow for a modular binding-time analysis. However, the paper does not consider recursion or polymorphism and there is no connection to regions. The authors suggest an extension to ML-style polymorphism by expanding `let`-expressions. However, this defeats their goal of modularity.

Hornof and Noyé [31] define a binding-time analysis for imperative languages that is flow, context, and return sensitive. Their context sensitivity corresponds to polyvariant binding time analysis, the other sensitivities are not applicable in our framework. However, their approach to polyvariance relies on duplicating function definitions and their corresponding data flow equations. It is not based on a notion of polymorphism. There is no formal correctness proof.

Earlier papers dealing with polyvariant analysis rely on abstract interpretation and use ad-hoc techniques to guarantee termination of the analysis [6, 8, 10, 11, 17, 46, 56]. Some of them do not formally relate the analysis with the specialization phase and some only cater for first-order functional languages.

1.5 Notation

Let $f \in A \to B$ be a partial function from set A to set B. The *domain* of f is $Dom(f) = \{x \in A \mid \exists y \in B \wedge f(x) = y\}$ and its *range* is $Ran(f) = \{y \in B \mid \exists x \in A \wedge f(x) = y\}$. We write $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ for a function f with $Dom(f) = \{x_1, \ldots, x_n\}$ and $\{\,\}$ for the empty function. A function $f \in A \to B$ is *total* whenever $Dom(f) = A$. The extension $f + \{x \mapsto a\}$ of f is a partial function where $(f + \{x \mapsto a\})(x') = f(x')$ if $x' \in Dom(f) \setminus \{x\}$ and $(f + \{x \mapsto a\})(x) = a$ otherwise. The *co-restriction* $f - A$ of a partial function f is defined by $(f - A)(x) = f(x)$ if $x \in Dom(f) \setminus A$ and undefined otherwise. A sequence of symbols a_i is written $\boldsymbol{a}$ and the set of the elements in a sequence $\boldsymbol{a}$ is denoted by $\{\boldsymbol{a}\}$. Finally, the symbol $=$ denotes equality of terms modulo α-conversion.

1.6 Overview

In section 2, we describe a constraint analysis on top of region inference: first, we introduce semantic objects (section 2.1) which we use to describe the constraint rules (section 2.2). We then formulate some basic properties (section 2.3) and define the notion of a constraint completion (section 2.4). Section 3 introduces the specialization semantics: we define a two-level extension of the region calculus (section 3.1) and give a small-step operational semantics (section 3.2). We illustrate specialization with two examples (section 3.3) and conclude by adding the missing type rules for the two-level extension of the region calculus (section 3.4). The correctness proof is split in two parts: Section 4 handles soundness of the constraint-analysis. Two substitution lemmas are needed to prove type preservation. Then, we prove progress and conclude with soundness of the constraint-analysis. Specialization soundness is proven in section 5. We first repeat an equational theory for λ^{region} and its soundness with respect to a notion of contextual equivalence from earlier work [26] (section 5.1). Then, we use this syntactic theory to prove specialization soundness (section 5.2). Finally, in section 6, we discuss future work and conclude.

2 Constraint Analysis

We specify the static semantics of the specializer in terms of a constraint analysis on top of region inference. Since the analysis builds on the type system for λ^{region}, we concentrate on the constraint aspects and refer to other work for a discussion of λ^{region}-typing [7, 28, 61]. The constraint analysis formalizes the generation of constraints between regions. Since λ^{region} employs polymorphic recursion for regions, we generally have to solve constraints at runtime. Therefore, we will use the term *constraint analysis* instead of binding-time analysis in the rest of the paper.

The result of the constraint analysis is a $\lambda_C^{\text{region}}$-expression, generated by the following grammar:

$$\begin{aligned}\mathsf{ConstraintTerm} \ni \mathbf{e} ::=\ & x \mid c \ \mathtt{at}\ \rho \mid \lambda x.\, \mathbf{e}\ \mathtt{at}\ \rho \mid \mathbf{e} @ \mathbf{e} \mid \mathtt{lift}\,[\rho_1, \rho_2]\, \mathbf{e} \mid \\ & \Lambda\, \varrho : C\, .\, \mathbf{e} \mid \mathtt{let}\ x = \mathbf{e}\ \mathtt{in}\ \mathbf{e} \mid f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \\ & \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, \mathbf{e}\ \mathtt{at}\ \rho\ \mathtt{in}\ \mathbf{e}\end{aligned}$$

$$\varrho \in \mathsf{RegionVar} \qquad x \in \mathsf{Var} \qquad \rho \in \mathsf{Placeholder} = \mathsf{RegionVar}$$
$$C \in \mathsf{ConstraintSet} = \mathcal{P}(\{\rho_1 \leq \rho_2 \mid \rho_1, \rho_2 \in \mathsf{Placeholder}\})$$

Except for the use of region placeholders and a constraint set in region abstractions, $\lambda_C^{\text{region}}$-terms are identical to λ^{region}-terms. The sets $\mathsf{RegionVar}$ and Var are infinite denumerable disjoint sets of region and expression variables respectively. A region placeholder ρ is a meta-level variable which serves as a notational convenience to recycle type and reduction rules. In $\lambda_C^{\text{region}}$, a placeholder is necessarily a region variable. When we define specialization, we will extend the grammar for placeholders to include other objects. Finally, a constraint set C contains ordered pairs of regions[2], written $\rho_1 \leq \rho_2$. They impose *binding-time constraints* as we shall see below. In the region abstraction $\Lambda\, \varrho : C\, .\, \mathbf{e}$, the constraint set C must be solved to determine the binding-time of the region ϱ. We will make this mechanism more precise when we discuss specialization.

The *erasure*, $|\mathbf{e}|$, of a $\lambda_C^{\text{region}}$-term is the λ^{region}-term e obtained by stripping the constraint sets C from the region abstractions.

2.1 Semantic objects

Most semantic objects required in the constraint analysis are taken from the type system of λ^{region} [61]. In this section, we briefly summarize them and provide definitions to handle constraints. The type language of $\lambda_C^{\text{region}}$ is generated by

[2] We use the terms *region*, *region variable* and *placeholder* interchangeable when no confusion is possible.

the following grammar, where $\mathsf{EffectVar}$ and $\mathsf{TypeVar}$ are disjoint denumerable sets:

$$\epsilon \in \mathsf{EffectVar} \qquad \varphi \subseteq \mathsf{Effect} = \mathsf{Placeholder} \cup \mathsf{EffectVar} \qquad \alpha \in \mathsf{TypeVar}$$

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

$$\mathsf{QualTypeScheme} \ni \sigma ::= \forall\, \boldsymbol{\alpha}\, .\forall\, \boldsymbol{\epsilon}\, .\forall\, \boldsymbol{\varrho}\, .C \Rightarrow \tau$$

An *effect* φ is a finite set of regions, ρ, and *effect variables*, ϵ. The effect of a term contains the set of regions that are affected by its evaluation. A *type* is either a type variable, α, an integer type, or a function type. Function types carry an *arrow effect* $\epsilon.\varphi$, which is a pair of an effect variable and a latent effect. Effect variables are a technical device for type reconstruction of λ^{region} [58]. They play no active role during constraint analysis. The effect variable identifies a group of functions that share at least one application. The use of effect variables corresponds to a simple control-flow analysis. The latent effect of an arrow effect contains the regions affected by the body of the function and is usually discharged upon function application. In our framework, they discharge upon function creation to account for specialization-time effects. We will explain this somewhat more conservative approach to the effect analysis when we discuss the actual constraint analysis.

A *type with place*, μ, is a pair of a type, τ, and a region, ρ. The region specifies where the object of type τ is stored. A *qualified type scheme*, σ, abstracts a type over type, effect, and region variables. Moreover, it contains a constraint set which transports constraints from the function definition to the function application.

A substitution, $S_s = (S_t, S_e, S_r)$, is a triple of total functions: a type substitution $S_t : \mathsf{TypeVar} \to \mathsf{Type}$, an effect substitution $S_e : \mathsf{EffectVar} \to \mathsf{EffectVar} \times \mathsf{Effect}$, and a region substitution $S_r : \mathsf{Placeholder} \to \mathsf{Placeholder}$. The definition of S_s on types and effects extends the usual definition [61] as follows:

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

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

$$S_s(\alpha) = S_t(\alpha) \quad S_s(\mathtt{int}) = \mathtt{int} \quad S_s(C) = \{S_s(\rho_1) \le S_s(\rho_2) \mid \rho_1 \le \rho_2 \in C\}$$

$$S_s(\forall\, \boldsymbol{\alpha}\, .\forall\, \boldsymbol{\epsilon}\, .\forall\, \boldsymbol{\varrho}\, .C \Rightarrow \tau) = \forall\, \boldsymbol{\alpha}\, .\forall\, \boldsymbol{\epsilon}\, .\forall\, \boldsymbol{\varrho}\, .S'_s(C) \Rightarrow S'_s(\tau) \quad \textit{where } S'_s = S_s - \{\boldsymbol{\epsilon}, \boldsymbol{\varrho}, \boldsymbol{\alpha}\}$$

The substitutions I_t and I_r are total identities on type and region variables, respectively, where the substitution I_e maps every effect variable ϵ to $\epsilon.\{\}$. We write S_r as a shorthand for (I_t, I_e, S_r) and similarly for the other substitutions. The *support* of a type substitution, $Supp(S_t)$, is the set $\{\alpha \mid S_t(\alpha) \neq \alpha\}$. By abuse of notation we sometimes write $\{\alpha_1 \mapsto S_t(\alpha_1), \ldots, \alpha_n \mapsto S_t(\alpha_n)\}$ whenever the $Supp(S_t) = \{\alpha_1, \ldots, \alpha_n\}$. The $Supp(S_r)$, $Supp(S_e)$ and $Supp(S_s)$ are defined analogously. For an object O, we also write $O[\rho_1 \mapsto \rho'_1, \ldots, \rho_n \mapsto \rho'_n]$ instead of $\{\rho_1 \mapsto \rho'_1, \ldots, \rho_n \mapsto \rho'_n\}(O)$. Similarly, we define the *image* of a type substitution S_t by $Img(S_t) = \{S_t(\alpha) \mid \alpha \in Supp(S_t)\}$ and analogously for S_e, S_r and S_s.

A pair (C, τ) is an *instance* of a qualified type scheme $\sigma = \forall \boldsymbol{\alpha} . \forall \boldsymbol{\epsilon} . \forall \boldsymbol{\varrho} . C' \Rightarrow \tau'$ via substitution S_s, written $(C, \tau) \prec \sigma \ \mathit{via} \ S_s$, if $\mathit{Supp}(S_t) \subseteq \{\boldsymbol{\alpha}\}$, $\mathit{Supp}(S_e) \subseteq \{\boldsymbol{\epsilon}\}$ and $\mathit{Supp}(S_r) \subseteq \{\boldsymbol{\varrho}\}$, where $S_s(\tau') = \tau$ and $S_s(C') = C$. The instance relation extends to type schemes by $\sigma \prec \sigma'$ iff $(C, \tau) \prec \sigma \ \mathit{via} \ S_s$ implies $(C, \tau) \prec \sigma' \ \mathit{via} \ S'_s$.

A type environment, $\mathit{TE} \in \mathsf{SpecTypeEnv}$, is a partial function that maps lambda-bound variables to pairs of the form (τ, ρ) and `letrec`-bound variables to pairs of the form (σ, ρ). Substitutions extend to type environments, and expressions in the obvious way, avoiding capture by renaming.

The notion of *erasure* for a $\lambda_C^{\text{region}}$-expression naturally carries over to type schemes and type environments (written $|\sigma|$ and $|\mathit{TE}|$ respectively), by stripping the constraint sets C. The corresponding syntactic categories are $\mathsf{TypeScheme}$ and $\mathsf{TypeEnv}$ respectively.

Free variables of semantic objects are defined in the usual way. For a semantic object O, the free type, region (placeholder), and effect variables of O are $\mathit{ftv}\,(O)$, $\mathit{frv}\,(O)$, and $\mathit{fev}\,(O)$, respectively. Furthermore, $\mathit{fv}\,(O) = \mathit{ftv}\,(O) \cup \mathit{frv}\,(O) \cup \mathit{fev}\,(O)$.

A constraint set, C, is *transitive closed* if $\rho_1 \leq \rho_2 \in C$ and $\rho_2 \leq \rho_3 \in C$ implies $\rho_1 \leq \rho_3 \in C$. The constraint set C' is *consistent* with respect to C ($C \rhd C'$) iff C is transitive closed and $C' \subseteq C$. Given some constraint set C, there is always a least set $\widetilde{C}$ such that $\widetilde{C} \rhd C$. It is constructed by closing C under transitivity and we always have $\mathit{frv}\,(C) = \mathit{frv}\,(\widetilde{C})$. Constraint sets have upper and lower projections with respect to a set of regions:

$$C^{\uparrow\{\rho_1,\ldots,\rho_n\}} = \{\rho_0 \leq \rho'_0 \in C \mid \exists i \in \{1,\ldots,n\} : \rho_0 = \rho_i \vee \rho'_0 = \rho_i\}$$

$$C_{\downarrow\{\rho_1,\ldots,\rho_n\}} = C \setminus C^{\uparrow\{\rho_1,\ldots,\rho_n\}} \qquad C_{\downarrow\rho} = C_{\downarrow\{\rho\}} \qquad C^{\uparrow\rho} = C^{\uparrow\{\rho\}}$$

A binding-time, β, ranges over the set $\{\mathsf{s}, \mathsf{d}\}$, which is ordered by the least partial order $\leq$ such that $\mathsf{s} \leq \mathsf{d}$. A *region annotation*, $\delta \in \mathsf{BtAnn} = (\mathsf{Placeholder} \to \{\mathsf{s}, \mathsf{d}\})$, is a partial function from regions to binding times.

A region annotation δ is a *solution* for a constraint set C (written $\delta \models C$) if $\mathit{frv}\,(C) \subseteq \mathit{Dom}(\delta)$ and for all $\rho_1 \leq \rho_2 \in C$ it holds that $\delta(\rho_1) \leq \delta(\rho_2)$. A constraint set C is *solvable* if it has a solution. The set of all solutions of C is closed under greatest lower bounds: if δ_1 and δ_2 are solutions, then so is $\min(\delta_1, \delta_2)$ (pointwise minimum). This implies that if C is solvable, then there exists a unique least solution, which is the minimum of all solutions.

2.2 Constraint Rules

A traditional specializer does not allow static values to be embedded in generated code. This property is enforced by the well-formedness of the binding-time annotations and is specified using constraints on the underlying type system. A type scheme, σ, is well-formed with respect to constraint set C if the judgment

$C \vdash_{wft} \sigma$ can be derived using the following rules:

$$\textit{(Wft-1)}\quad \frac{C \rhd \emptyset}{C \vdash_{wft} (\texttt{int}, \rho)} \qquad \textit{(Wft-2)}\quad \frac{\begin{array}{c} C \rhd \{\rho \leq \rho_1, \rho \leq \rho_2\} \quad C \vdash_{wft} \mu_1 \\ C \vdash_{wft} \mu_2 \quad \mu_1 = (\tau_1, \rho_1) \quad \mu_2 = (\tau_2, \rho_2) \end{array}}{C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)}$$

$$\textit{(Wft-3)}\quad \frac{C \vdash_{wft} (\tau, \rho)}{C_{\downarrow\{\boldsymbol{\varrho}\}} \vdash_{wft} (\forall \boldsymbol{\alpha} . \forall \boldsymbol{\epsilon} . \forall \boldsymbol{\varrho} . C^{\uparrow\{\boldsymbol{\varrho}\}} \Rightarrow \tau, \rho)}$$

This judgment only collects the necessary constraints for well-formedness, it does not guarantee that the resulting constraint set C is solvable.

For base types, rule *(Wft-1)* requires a transitive closed C, but no additional constraints. An arrow type is well-formed by rule *(Wft-2)*. It requires that the subparts of the arrow are well-formed and includes constraints in C which guarantee that the subparts of the arrow are dynamic whenever the function is dynamic. Rule *(Wft-3)* generalizes the well-formedness property to a pair of a qualified type scheme and region. Using this definition, we can define *a well-formed type environment* TE with respect to a constraint set C (written $C \vdash_{wft} TE$) as follows: for all $x \in Dom(TE)$, it holds that $C \vdash_{wft} TE(x)$.

A constraint-type judgment has the form $TE, C \vdash_c \textbf{e} : \mu, \varphi$, which is read "given a type environment TE and a constraint set C, the $\lambda_C^{\text{region}}$-expression $\textbf{e}$ has type μ and effect φ". The rules in Figure 1 specify the constraint analysis.

Rules *(Var)*, *(Const)*, *(App)*, *(Let)* and *(Effect)* are standard (see [61] for a good explanation). Rule *(Lift)* deals with copying a value from region ρ_1 to ρ_2. Accordingly, we require that ρ_1 is of earlier binding-time than ρ_2. This allows for (first-order) *binding-time coercions* as we shall see below.

Rule *(Regabst)* introduces a region ϱ. The body, $\textbf{e}$, of the region abstraction is type checked with the constraint set C. The region abstraction passes those constraints to the context that do not mention ρ, *i.e.* $C_{\downarrow\varrho}$. The binding-time of ϱ is determined by the constraints that only mention ρ, *i.e.* $C^{\uparrow\varrho}$.

Similar reasoning applies to rule *(Letrec)*: the body of f is type checked with a constraint set C, but the type scheme of f only needs the constraint set $C^{\uparrow P}$. The `letrec`-body and the context can be type-checked with the constraints in C *not* mentioning those in P, *i.e.* $C_{\downarrow P}$.

Rule *(Recvar)* types a region application. After instantiation, it shifts the constraint set of the type scheme into the constraint set of the context. Moreover, it adds constraints guaranteeing that the application has the same binding-time as the defining `letrec`. Additionally, if the defining `letrec`-bound function is dynamic, it cannot pass static arguments.

Rule *(Abstr)* slightly deviates from the standard type system for λ^{region} in its treatment of the effect. It types a lambda expression and introduces well-formedness constraints, which we need to obtain sound specialization. As usual, the latent effect φ' is a superset of the effect φ of the body. This potential enlargement is required when two expressions, evaluating to different functions, flow together and therefore need the same functional type. In the standard type

$$\textit{(Var)} \quad \frac{TE(x) = \mu \quad C \rhd \emptyset}{TE, C \vdash_c x : \mu, \emptyset}$$

$$\textit{(Const)} \quad \frac{C \vdash_{wft} (\mathtt{int}, \rho)}{TE, C \vdash_c c \ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}}$$

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

$$\textit{(Letrec)} \quad \frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\boldsymbol{\epsilon}\}) \cap (\mathit{fv}\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \boldsymbol{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\,.C^{\uparrow P} \Rightarrow \tau \quad \{\boldsymbol{\alpha}\} \cap \mathit{ftv}\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \lambda x.\, \mathbf{e}_1\ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \\ \sigma = \forall\, \boldsymbol{\alpha}\,.\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c \mathbf{e}_2 : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, \mathbf{e}_1\ \mathtt{at}\ \rho\ \mathtt{in}\ \mathbf{e}_2 : \mu, \varphi \cup \varphi'}$$

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

$$\textit{(Let)} \quad \frac{TE, C \vdash_c \mathbf{e}_1 : \mu', \varphi' \quad TE + \{x \mapsto \mu'\}, C \vdash_c \mathbf{e}_2 : \mu, \varphi}{TE, C \vdash_c \mathtt{let}\ x = \mathbf{e}_1\ \mathtt{in}\ \mathbf{e}_2 : \mu, \varphi \cup \varphi'}$$

$$\textit{(Recvar)} \quad \frac{\begin{array}{c} TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \boldsymbol{\alpha}\,.\forall\, \boldsymbol{\epsilon}\,.\forall\, \varrho_1, \ldots, \varrho_n\,.C'' \Rightarrow \tau \\ (C', \tau') \prec \sigma \ \mathit{via}\ (S_t, S_e, \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}) \\ C \rhd C' \cup \{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho'_1, \ldots, \rho' \leq \rho'_n\} \end{array}}{TE, C \vdash_c f\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

$$\textit{(Regabst)} \quad \frac{TE, C \vdash_c \mathbf{e} : \mu, \varphi \quad \varrho \notin \mathit{frv}\,(TE, \mu)}{TE, C_{\downarrow \varrho} \vdash_c \Lambda\, \varrho : C^{\uparrow \varrho}\,.\, \mathbf{e} : \mu, \varphi \setminus \{\varrho\}}$$

$$\textit{(Effect)} \quad \frac{TE, C \vdash_c \mathbf{e} : \mu, \varphi \quad \epsilon \notin \mathit{fev}\,(TE, \mu)}{TE, C \vdash_c \mathbf{e} : \mu, \varphi \setminus \{\epsilon\}}$$

$$\textit{(Lift)} \quad \frac{TE, C \vdash_c \mathbf{e} : (\mathtt{int}, \rho_1), \varphi \quad C \rhd \{\rho_1 \leq \rho_2\}}{TE, C \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\, \mathbf{e} : (\mathtt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}}$$

Fig. 1. Constraint Analysis

system for λ^{region}, the effect of a lambda expression is just $\{\rho\}$, indicating that a closure is allocated in the region ρ. However, since the constraint analysis is a static semantics for program *specialization*, the effect $\{\rho\}$ is not generally sufficient. Indeed, if the lambda turns out to be dynamic (where we stress that the actual binding-time is only known at specialization time), then the body $\mathbf{e}$ is further reduced. Hence, the effect of the lambda's body must be included to ensure soundness of the analysis, otherwise, a dangling pointer may be dereferenced.

Our rule *(Abstr)* from Figure 1 avoids this problem at the cost of delayed region deallocation. This approach is conservative because discharging the arrow effect in *(Abstr)* is not necessary if the lambda is static. Fortunately, it turns out that only dead code within a static lambda is de-allocated later. Indeed, if it is not dead, the body's effect will be discharged by rule *(App)*. Moreover, late deallocation is not so critical in a program specialization context. The main reason why early de-allocation is still desirable is to keep the constraint sets small as can be seen from rule *(Regabst)*.

Note that we cannot do away with region abstraction altogether. First, because of polymorphic recursion, regions may be introduced in every recursive function unfolding, which is only decided at runtime. Second, having region abstractions in $\lambda_C^{\text{region}}$ keeps a close relationship between the constraint analysis and the original λ^{region}-type system.

The attentive reader may find the change in rule *(Abstr)* problematic because it might invalidate known region reconstruction algorithms. However, it turns out that only minimal changes are required in the algorithms by Birkedal and Tofte [3, 57].

In this paper, we base the constraint analysis on a region derivation, as produced by a region inference algorithm. Hence, all expressions $\mathbf{e}$ of $\lambda_C^{\text{region}}$ have a fixed region type with place μ indicated by the notation $\mathbf{e}^{\mu}$. However, to avoid clutter, we usually leave out the annotation unless it is required by the context.

The constraint analysis of Figure 1 induces a type judgment $TE' \vdash_t e : \mu, \varphi$ for λ^{region} by dropping the constraint sets and well-formedness judgments. It says that the λ^{region}-expression e has type with place μ and effect φ given a type environment $TE' \in \mathsf{TypeEnv}$, where $TE' = |TE|$ for a type environment $TE \in \mathsf{SpecTypeEnv}$.

2.3 Properties

We formulate some technical lemmas.

Lemma 1. *Let $P = \{\rho_1, \dots, \rho_n\}$.*

1. *$C^{\uparrow P} \cup C_{\downarrow P} = C$ and $C^{\uparrow P} \cap C_{\downarrow P} = \emptyset$.*
2. *If C is transitive closed, then so is $C_{\downarrow P}$.*

Lemma 2. *Suppose $C \vdash_{wft} \mu$, then C is transitive closed.*

Lemma 3. *Suppose $TE, C \vdash_c \mathbf{e} : \mu, \varphi$, then C is transitive closed.*

Proof. The proof is a simple induction on the type derivation, using Lemma 1.2 and Lemma 2.

It is always safe to add constraints to a derivation after closing them under transitivity:

Lemma 4. *Suppose* $TE, C \vdash_c \mathbf{e} : \mu, \varphi$ *and* $C' \rhd C$ *then* $TE, C' \vdash_c \mathbf{e} : \mu, \varphi$.

A constraint type judgment exhibits a type substitution lemma:

Lemma 5. *If* $TE, C \vdash_c \mathbf{e} : \mu, \varphi$ *then, for all substitutions* S_s*, we have that* $S_s(TE), S_s(C) \vdash_c S_s(\mathbf{e}) : S_s(\mu), S_s(\varphi)$.

Proof. By induction on the derivation of $TE, C \vdash_c \mathbf{e} : \mu, \varphi$. Standard extension of Lemma 5.4 in [28].

It also obeys a weakening lemma:

Lemma 6. *If* $TE + \{f \mapsto (\sigma, \rho)\}, C \vdash_c \mathbf{e} : \mu, \varphi$*, and* $\sigma \prec \sigma'$*, then* $TE + \{f \mapsto (\sigma', \rho)\}, C \vdash_c \mathbf{e} : \mu, \varphi$.

Proof. A straightforward induction on $TE + \{f \mapsto (\sigma, \rho)\}, C \vdash_c \mathbf{e} : \mu, \varphi$. Simple extension of Lemma 5.5 in [28].

Lemma 7. *If* $TE, C \vdash_c \mathbf{e} : \mu, \varphi$ *and* $C \vdash_{wft} TE$*, then* $C \vdash_{wft} \mu$.

Proof. The proof follows by straightforward induction on the type derivation.

Closing a constraint set C does not affect its solution.

Lemma 8. *If* $\delta \models C$*, then* $\delta \models \widetilde{C}$.

A region annotation solving a transitive closed set can always be extended with one new region variable, without sacrifying solvability:

Lemma 9. *Suppose* C *is transitive closed and* $\delta \models C_{\downarrow\varrho}$ *with* $\varrho \notin Dom(\delta)$*. Then, there always exists a* β *such that* $\delta + \{\varrho \mapsto \beta\} \models C^{\uparrow\varrho}$.

Proof. By contradiction: assume there is no such β for which $\delta + \{\varrho \mapsto \beta\} \models C^{\uparrow\varrho}$. Then, since every constraint in $C^{\uparrow\varrho}$ mentions ϱ, there must be a pair of constraints $\{\varrho_1 \leq \varrho, \varrho \leq \varrho_2\} \subseteq C^{\uparrow\varrho}$ where $\varrho \notin \{\varrho_1, \varrho_2\}$ such that $\delta(\varrho_1) = \mathsf{d}$ and $\delta(\varrho_2) = \mathsf{s}$. However, since we have that C is transitive closed, it holds that $\varrho_1 \leq \varrho_2 \in C$ and hence, $\varrho_1 \leq \varrho_2 \in C_{\downarrow\varrho}$. By assumption we have that $\delta \models C_{\downarrow\varrho}$, so $\delta(\varrho_1) = \mathsf{d} \leq \mathsf{s} = \delta(\varrho_2)$, which leads to a contradiction.

In addition, such an extended solution works for the whole constraint set:

Lemma 10. *If* $\delta \models C_{\downarrow\varrho}$ *and* $\delta' = \delta + \{\varrho \mapsto \beta\}$ *so that* $\delta' \models C^{\uparrow\varrho}$*, then* $\delta' \models C$.

Proof. By Lemma 1.1, we have that $C = C^{\uparrow\varrho} \cup C_{\downarrow\varrho}$, so for any $\rho_1 \leq \rho_2 \in C$, we either have $\rho_1 \leq \rho_2 \in C^{\uparrow\varrho}$, which trivially implies $\delta \models C^{\uparrow\varrho}$ by assumption, or $\rho_1 \leq \rho_2 \in C_{\downarrow\varrho}$. Since $\delta \models C_{\downarrow\varrho}$ and $\varrho \notin frv\,(C_{\downarrow\varrho})$, by definition, we obviously have that $\delta' \models C_{\downarrow\varrho}$.

2.4 Constraint Completion

An offline partial evaluator requires a *constraint completion*[3] to specialize successfully. Starting from a region derivation $\{\ \} \vdash_t e : \mu, \varphi$, the constraint completion provides a $\lambda_C^{\text{region}}$-term $\mathbf{e}$ whose erasure is identical to the λ^{region}-term e and which is typeable in the constraint analysis with a solvable constraint set. The following definition makes this precise:

Definition 1. *Suppose* $\{\ \} \vdash_t e : \mu, \varphi$*. A* $\lambda_C^{\text{region}}$*-expression* $\mathbf{e}$ *with initial constraint set* C *and region annotation* δ_c *is a* constraint completion *of* e *iff* $|\mathbf{e}^\mu| = e$*, the judgment* $\{\ \}, C \vdash_c \mathbf{e}^\mu : \mu, \varphi$ *holds,* $\delta_c \models C$ *and for all* $\rho \in \varphi \cup frv(C, \mu)$ *it holds that* $\delta_c(\rho) = \mathsf{d}$*.*

A constraint completion always exists if region inference succeeds.

Theorem 1. *Suppose* $\{\ \} \vdash_t e : \mu, \varphi$*. Then there exists a constraint completion* $\mathbf{e}^\mu$ *with region annotation* δ_c*.*

Proof. See Appendix A

In partial evaluation, it is important to have an idea on the *quality* of the resulting binding-times. In our framework we actually want to know about the quality of the constraints since they determine the binding time during specialization. There are two factors influencing this.

The first factor is the accuracy of the underlying region inference. If only one region is allocated, no specialization will take place whatsoever as this region has to be dynamic. Generally, one would like to separate values in as many regions as possible since every region can have a different binding time. Fortunately, region separation coincides with the region inference goals in a memory management setting, where more regions allow more fine-grained garbage collection [61]. However, it is still an open question if every Hindley-Milner typeable expression has a principal (or most general) region type. As a consequence, we cannot guarantee a principal binding-time type, unlike Dussart et al. [16].

The second binding-time quality factor is the number of constraints in the set C. The proof of Theorem 1 specifies a constraint analysis algorithm which collects minimal constraint sets. Moreover, since there exists a unique minimal region annotation, we obtain optimal specialization, only limited by the results of region inference.

Finally, we observe that the total number of constraints that must be solved at region allocation time is bound by the number of regions in the program. This upper bound (pathological in practice) holds no matter how well region inference works.

3 Specialization

We now present a region-based program specializer as a small-step structural operational semantics [45]. As in earlier work, we will extend $\lambda_C^{\text{region}}$-expressions

[3] In partial evaluation literature, this is usually called a *binding-time completion* [29].

$$\mathsf{DeadRegionVar} = \{\varrho^\bullet \mid \varrho \in \mathsf{RegionVar}\} \qquad \mathsf{BTRegion} = \{\mathit{s}, \mathit{d}\}$$
$$\mathsf{Placeholder} = \mathsf{RegionVar} \cup \mathsf{DeadRegionVar} \cup \mathsf{BTRegion}$$

$$\begin{aligned}
\mathsf{TwolevelTerm} \ni E ::= {} & V \mid R \mid c\ \mathtt{at}\ \rho \mid \lambda x.\, E\ \mathtt{at}\ \rho \mid E \mathbin{@} E \mid \mathtt{lift}\,[\rho_1, \rho_2]\, E \mid \\
& \Lambda^\beta\, \varrho.\, E \mid \Lambda\, \varrho : C \,.\, E \mid \mathtt{let}\ x = E\ \mathtt{in}\ E \mid \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ E \mid \\
& f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E\rangle_\rho\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \\
& \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E\ \mathtt{at}\ \rho\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, R\ \mathtt{at}\ \rho\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E\rangle_\rho\ \mathtt{in}\ E
\end{aligned}$$

$$\mathsf{Value} \ni V ::= \langle c\rangle_\rho \mid \langle \lambda x.\, E\rangle_\rho$$

$$\begin{aligned}
\mathsf{ResidualTerm} \ni R ::= {} & x \mid c\ \underline{\mathtt{at}}\ \varrho \mid \underline{\lambda} x.\, R\ \underline{\mathtt{at}}\ \varrho \mid R \mathbin{\underline{@}} R \mid \underline{\mathtt{lift}}\,[\varrho_1, \varrho_2]\, R \mid \\
& \underline{\Lambda}\, \varrho.\, R \mid \underline{\mathtt{let}}\ x = R\ \underline{\mathtt{in}}\ R \mid f\ [\varrho_1, \ldots, \varrho_n]\ \underline{\mathtt{at}}\ \varrho \mid \\
& \underline{\mathtt{letrec}}\ f = \underline{\Lambda}\, \varrho_1, \ldots, \varrho_n.\, \underline{\lambda} x.\, R\ \underline{\mathtt{at}}\ \varrho\ \underline{\mathtt{in}}\ R
\end{aligned}$$

Fig. 2. The $\lambda 2^{\text{region}}$-language

with *computational* terms expressing intermediate state. In addition, we add *residual* and intermediate two-level terms.

3.1 Computational Two-Level Expressions

The computational two-level language, $\lambda 2^{\text{region}}$, is produced by the grammar in Figure 2. Two-level terms extend the terms of $\lambda_C^{\text{region}}$ with new region placeholders, values V, residual terms R and a few intermediate constructs.

Next to region variables, placeholders may stand for deallocated region variables (DeadRegionVar) as well as two unique global binding-time regions (BTRegion), where we assume that DeadRegionVar, BTRegion and RegionVar are mutually disjoint. In analogy to previous work [28], we deallocate a region by annotating the region variable with $\bullet$. Additionally, we introduce two *virtual* binding-time region variables, s and d. They are a technical device for proving soundness of the constraint-analysis as we will explain below. Often, we want to know if a set of region placeholders only contains region variables, *i.e.* no deallocated or binding-time regions. To this end, define the predicate $\mathit{Clean}(A)$ for some set A to be true iff $\mathit{frv}\,(A) \subseteq \mathsf{RegionVar}$ and false otherwise.

A value V is a *pointer* to a constant or a lambda abstraction. For example, the term $\langle c\rangle_\varrho$ denotes a pointer to a constant c allocated in region ϱ, whereas the expression $c\ \mathtt{at}\ \varrho$ performs the allocation of the constant. Similarly, we have a region closure pointer, $\langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, E\rangle_\rho$, however, unlike values, they cannot occur in isolation within a two-level expression. It is either bound to a variable in a `letrec`-expression or applied to some region arguments. Pointers and deallocated regions are extensively discussed elsewhere [7, 28].

The computational two-level calculus also defines *residual terms*. They are identical to the terms of λ^{region}, except that the keywords are underlined. Residual terms are the result of specialization of a constraint completion as we shall prove later on.

Finally, we have four intermediate constructs. The dynamic $\texttt{let}^{\mathsf{d}}$ and $\texttt{letrec}^{\mathsf{d}}$ terms are very much like `let` and `letrec` expressions, except that they require their bound expressions to be residual. The binding-time annotated region abstraction $\Lambda^{\beta}\,\varrho.E$ is the result of reducing a constraint qualified region abstraction $\Lambda\,\varrho : C\,.\,E$. The binding time β is calculated from the constraint set C and fixed for the rest of the reduction.

We write $E[x \mapsto E']$ for the capture-avoiding substitution of all free occurrences of x in E by the $\lambda 2^{\text{region}}$-expression E'.

The terms in $\lambda 2^{\text{region}}$ contain expressions, which never occur in the *result* of a specialization. We make the notion of result more accurate by defining two sub-sets of $\mathsf{TwolevelTerm}$, produced by the following grammar:

$$\begin{aligned}\mathsf{StaticAnswer} \ni \mathrm{Q}^{\mathsf{s}} ::= {} & V \mid \Lambda^{\mathsf{d}}\,\varrho.\,\mathrm{Q}^{\mathsf{s}} \mid \texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}} \mid \\ & \texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1,\ldots,\varrho_n.\,\lambda x.\,R\ \texttt{at}\ \rho\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}}\end{aligned}$$

$$\mathsf{Answer} \ni\ Q ::= \mathrm{Q}^{\mathsf{s}} \mid R$$

A specialization *answer*, Q, cannot be further reduced by the transition semantics. It is either a residual term R or a *static answer*, Q^{s}, the latter being a value V embedded in certain dynamic contexts: the empty context, a dynamic region abstraction, a $\texttt{let}^{\mathsf{d}}$-expression or a $\texttt{letrec}^{\mathsf{d}}$-expression. A static answer that is not a value admits some well-known binding-time improvements as we will explain below.

We extend the notion of erasure for $\lambda_C^{\text{region}}$-expressions to $\lambda 2^{\text{region}}$-expressions: apart from stripping the constraint sets, the erasure of a $\lambda 2^{\text{region}}$-term E deletes underlines, removes the binding-time annotations from region binders and the `let` and `letrec` construct, and maps region placeholders that are not region variables onto $\bullet$. For a two-level term E, we have that $|E|$ is a computational term for λ^{region} as in [28], except for the addition of the `let`-construct. Henceforth, we will refer to the computational extension of λ^{region} as λ^{cregion}.

Not all region annotations δ over placeholders in $\lambda 2^{\text{region}}$ are useful for specialization. For example, region deallocation only takes place at specialization time and we expect the static region $\boldsymbol{s}$ to be mapped on the static binding-time s and similarly, the dynamic region $\boldsymbol{d}$ should be of binding-time d too. Therefore assume an initial region annotation δ_{init} such that $\delta_{\text{init}} = \{\varrho^{\bullet} \mapsto \mathsf{s} \mid \varrho \in \mathsf{RegionVar}\} + \{\boldsymbol{s} \mapsto \mathsf{s}, \boldsymbol{d} \mapsto \mathsf{d}\}$.

3.2 Small-step Semantics

The small-step semantics is defined as a relation $\leadsto\, \in \mathcal{P}(\mathsf{BtAnn} \times \mathsf{TwolevelTerm} \times \mathsf{BtAnn} \times \mathsf{TwolevelTerm})$. It defines one reduction step from a region annotation and a computational $\lambda 2^{\text{region}}$–term into a new region annotation and $\lambda 2^{\text{region}}$–term. The region annotation δ records the binding times for regions as they

$$\delta, c \texttt{ at } \varrho \rightsquigarrow \delta, \langle c \rangle_\varrho \quad \textit{if } \delta(\varrho) = \mathsf{s} \tag{1}$$

$$\delta, \lambda x.\, E \texttt{ at } \varrho \rightsquigarrow \delta, \langle \lambda x.\, E \rangle_\varrho \quad \textit{if } \delta(\varrho) = \mathsf{s} \tag{2}$$

$$\begin{array}{c} \delta, \texttt{letrec } f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \texttt{ at } \varrho \texttt{ in } E_2 \rightsquigarrow \\ \delta, \texttt{letrec } f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \rangle_\varrho \texttt{ in } E_2 \end{array} \quad \textit{if } \delta(\varrho) = \mathsf{s} \tag{3}$$

$$\delta, \Lambda^{\mathsf{s}}\, \varrho.\, Q \rightsquigarrow \delta, Q[\varrho \mapsto \varrho^\bullet] \tag{4}$$

$$\delta, \texttt{lift}\, [\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \rightsquigarrow \delta, \langle c \rangle_{\varrho_2} \quad \textit{if } \delta(\varrho_2) = \mathsf{s} \tag{5}$$

$$\delta, \langle \lambda x.\, E \rangle_\varrho \,@\, Q \rightsquigarrow \delta, \texttt{let } x = Q \texttt{ in } E \tag{6}$$

$$\delta, \texttt{let } x = V \texttt{ in } E \rightsquigarrow \delta, E[x \mapsto V] \tag{7}$$

$$\delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E \rangle_\varrho\, [\rho'_1, \ldots, \rho'_n] \texttt{ at } \varrho_0 \rightsquigarrow \delta, \langle \lambda x.\, E[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n] \rangle_{\varrho_0} \tag{8}$$

$$\begin{array}{c} \delta, \texttt{letrec } f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \rangle_\rho \texttt{ in } E_2 \rightsquigarrow \\ \delta, E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x. \texttt{ letrec } f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \rangle_\rho \texttt{ in } E_1 \rangle_\rho] \end{array} \tag{9}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, E \,@\, E'' \rightsquigarrow \delta', E' \,@\, E''} \tag{10}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, Q \,@\, E \rightsquigarrow \delta', Q \,@\, E'} \tag{11}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{lift}\, [\rho_1, \rho_2]\, E \rightsquigarrow \delta', \texttt{lift}\, [\rho_1, \rho_2]\, E'} \tag{12}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let } x = E \texttt{ in } E'' \rightsquigarrow \delta', \texttt{let } x = E' \texttt{ in } E''} \tag{13}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \Lambda^{\mathsf{s}}\, \varrho.\, E \rightsquigarrow \delta', \Lambda^{\mathsf{s}}\, \varrho.\, E'} \tag{14}$$

Fig. 3. Specialization Semantics - I

are calculated during specialization. Figures 3, 4 and 5 specify the individual transition rules for $\rightsquigarrow$.

Rules (1), (2) and (3) allocate static constants, lambdas and region-lambda abstractions respectively. Rule (4) deallocates a static region after specializing the body. The `lift` in rule (5) moves a constant between two static regions and rule (6) specifies beta value reduction for static lambdas. Static region application is defined in rule (8) and rule (9) unfolds `letrec`-bound functions. Finally, rules (10) to (14) specify a left-to-right call-by-value semantics.

The binding-time of a region is calculated by rule (15) in Figure 4. The rule extends the region annotation, δ, such that the constraint C is solvable. The extra condition $\varrho \notin \mathit{Dom}(\delta)$ can trivially be satisfied by α-renaming ϱ in C and E if required.

Rule (16) implements a first-order *binding-time coercion*: it transforms a specialization-time constant into a dynamic constant by moving it from a static region into a dynamic region. Such binding-time coercions are necessary to obtain good results [29, 33]. Embedding every expression of type `int` in a `lift`-

$$\delta, \Lambda\,\varrho : C\,.\,E \rightsquigarrow \delta + \{\varrho \mapsto \beta\}, \Lambda^{\beta}\,\varrho.\,E \quad \text{if } \varrho \notin \mathit{Dom}(\delta) \text{ and } \delta + \{\varrho \mapsto \beta\} \models C \tag{15}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} \rightsquigarrow \delta, c \;\texttt{at}\; \varrho_2 \quad \text{if } \delta(\varrho_2) = \mathsf{d} \tag{16}$$

$$\delta, c \;\texttt{at}\; \varrho \rightsquigarrow \delta, c \;\underline{\texttt{at}}\; \varrho \quad \text{if } \delta(\varrho) = \mathsf{d} \tag{17}$$

$$\delta, \lambda\,x.\,R \;\texttt{at}\; \varrho \rightsquigarrow \delta, \underline{\lambda}x.\,R \;\underline{\texttt{at}}\; \varrho \quad \text{if } \delta(\varrho) = \mathsf{d} \tag{18}$$

$$\begin{array}{c}\delta, \texttt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,R_1 \;\texttt{at}\; \varrho \;\texttt{in}\; R_2 \rightsquigarrow \\ \delta, \underline{\texttt{letrec}}\; f = \underline{\Lambda}\,\varrho_1, \ldots, \varrho_n.\,\underline{\lambda}x.\,R_1 \;\underline{\texttt{at}}\; \varrho \;\underline{\texttt{in}}\; R_2\end{array} \tag{19}$$

$$\delta, \Lambda^{\mathsf{d}}\,\varrho.\,R \rightsquigarrow \delta, \underline{\Lambda}\,\varrho.\,R \tag{20}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,R \rightsquigarrow \delta, \underline{\texttt{lift}}\,[\varrho_1, \varrho_2]\,R \tag{21}$$

$$\delta, R_1 \,@\, R_2 \rightsquigarrow \delta, R_1 \,\underline{@}\, R_2 \tag{22}$$

$$\delta, \texttt{let}^{\mathsf{d}}\; x = R_1 \;\texttt{in}\; R_2 \rightsquigarrow \delta, \underline{\texttt{let}}\; x = R_1 \;\underline{\texttt{in}}\; R_2 \tag{23}$$

$$\delta, f\;[\varrho_1, \ldots, \varrho_n] \;\texttt{at}\; \varrho \rightsquigarrow \delta, f\;[\varrho_1, \ldots, \varrho_n] \;\underline{\texttt{at}}\; \varrho \tag{24}$$

$$\delta, \texttt{let}\; x = R \;\texttt{in}\; E \rightsquigarrow \delta, \texttt{let}^{\mathsf{d}}\; x = R \;\texttt{in}\; E \tag{25}$$

$$\begin{array}{c}\delta, \texttt{letrec}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,R \;\texttt{at}\; \varrho \;\texttt{in}\; E \rightsquigarrow \\ \delta, \texttt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,R \;\texttt{at}\; \varrho \;\texttt{in}\; E\end{array} \quad \text{if } \delta(\varrho) = \mathsf{d} \tag{26}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \lambda\,x.\,E \;\texttt{at}\; \varrho \rightsquigarrow \delta', \lambda\,x.\,E' \;\texttt{at}\; \varrho} \quad \text{if } \delta(\varrho) = \mathsf{d} \tag{27}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\begin{array}{l}\delta, \texttt{letrec}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,E \;\texttt{at}\; \varrho \;\texttt{in}\; E'' \rightsquigarrow \\ \delta', \texttt{letrec}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,E' \;\texttt{at}\; \varrho \;\texttt{in}\; E''\end{array}} \quad \text{if } \delta(\varrho) = \mathsf{d} \tag{28}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \Lambda^{\mathsf{d}}\,\varrho.\,E \rightsquigarrow \delta', \Lambda^{\mathsf{d}}\,\varrho.\,E'} \tag{29}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let}^{\mathsf{d}}\; x = R \;\texttt{in}\; E \rightsquigarrow \delta', \texttt{let}^{\mathsf{d}}\; x = R \;\texttt{in}\; E'} \tag{30}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\begin{array}{c}\delta, \texttt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,R \;\texttt{at}\; \varrho \;\texttt{in}\; E \rightsquigarrow \\ \delta', \texttt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda\,x.\,R \;\texttt{at}\; \varrho \;\texttt{in}\; E'\end{array}} \tag{31}$$

Fig. 4. Specialization Semantics - II

$$\delta, \texttt{lift}\,[\rho_1, \rho_2]\,(\Lambda^{\mathsf{d}}\,\varrho.\,\mathrm{Q}^{\mathsf{s}}) \leadsto \delta, \Lambda^{\mathsf{d}}\,\varrho.\,(\texttt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}}) \quad \textit{if } \varrho \notin \{\rho_1, \rho_2\} \tag{32}$$

$$\begin{array}{c}\delta, \texttt{let}\ x = (\Lambda^{\mathsf{d}}\,\varrho.\,\mathrm{Q}^{\mathsf{s}})\ \texttt{in}\ E^{\mu} \leadsto \\ \delta, \Lambda^{\mathsf{d}}\,\varrho.\,(\texttt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \texttt{in}\ E^{\mu})\end{array} \quad \textit{if } \varrho \notin \textit{frv}\,(\mu) \tag{33}$$

$$\delta, (\Lambda^{\mathsf{d}}\,\varrho.\,\mathrm{Q}^{\mathsf{s}}_1)\,@\,Q^{\mu}_2 \leadsto \delta, \Lambda^{\mathsf{d}}\,\varrho.\,(\mathrm{Q}^{\mathsf{s}}_1\,@\,Q^{\mu}_2) \quad \textit{if } \varrho \notin \textit{frv}\,(\mu) \tag{34}$$

$$\delta, \texttt{lift}\,[\rho_1, \rho_2]\,(\texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}}) \leadsto \delta, \texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ (\texttt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}}) \tag{35}$$

$$\begin{array}{c}\delta, \texttt{let}\ y = (\texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}})\ \texttt{in}\ E \leadsto \\ \delta, \texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ (\texttt{let}\ y = \mathrm{Q}^{\mathsf{s}}\ \texttt{in}\ E)\end{array} \quad \textit{if } x \notin \textit{fv}\,(E) \tag{36}$$

$$\delta, (\texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}}_1)\,@\,Q_2 \leadsto \delta, \texttt{let}^{\mathsf{d}}\ x = R\ \texttt{in}\ (\mathrm{Q}^{\mathsf{s}}_1\,@\,Q_2) \quad \textit{if } x \notin \textit{fv}\,(Q_2) \tag{37}$$

$$\begin{array}{c}\delta, \texttt{lift}\,[\rho_1, \rho_2]\,(\texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}}) \leadsto \\ \delta, \texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ (\texttt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}})\end{array} \tag{38}$$

$$\begin{array}{c}\delta, \texttt{let}\ x = (\texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}})\ \texttt{in}\ E \leadsto \\ \delta, \texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ (\texttt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \texttt{in}\ E)\end{array} \quad \textit{if } f \notin \textit{fv}\,(E) \tag{39}$$

$$\begin{array}{c}\delta, (\texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ \mathrm{Q}^{\mathsf{s}}_1)\,@\,Q_2 \leadsto \\ \delta, \texttt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\,\lambda x.\,R\ \texttt{at}\ \varrho\ \texttt{in}\ (\mathrm{Q}^{\mathsf{s}}_1\,@\,Q_2)\end{array} \quad \textit{if } f \notin \textit{fv}\,(Q_2) \tag{40}$$

Fig. 5. Specialization Semantics - III

expression yields optimal results [29]. This is the additional functionality of `lift` alluded to in section 1.2.

The rules (17) to (24) generate residual code. This is only allowed if all subterms are already residual. The rules (27) and (28) are context rules that specialize under dynamic lambdas. These rules are the main deviation from a standard semantics and increase the potential for non-termination. Mechanisms to avoid this undesirable behavior (e.g. program point specialization or memoization [51]) are not considered in this paper and left for future work. Rules (25) and (26) create the $\texttt{let}^{\mathsf{d}}$ and $\texttt{letrec}^{\mathsf{d}}$ intermediate expressions and the context rules (29), (30) and (31) specialize inside static answers.

Recall that static answers Q^{s} are static values embedded in a dynamic context. If a static answer occurs in a static context, a specializer can bring the static context closer to the embedded static value, allowing further specialization. These are well-known *binding-time improvements* in partial evaluation for call-by-value calculi [22, 37, 38] and are implemented by the specialization rules in Figure 5.

In $\lambda 2^{\text{region}}$ there are five such static contexts, which can be read from the context rules (10) to (14). However, as can be seen from reduction rule (6), we recast the static context induced by rule (11) to that of (13). Moreover, the static context from rule (14) does not need to be transported since rule (4) deallocates a static region for any answer Q. Hence, the following three static contexts need to be transported to the static value: a lift-expression, which is handled by the rules (32), (35) and (38); a let-expression, handled by the rules (33), (36) and (39); and function application, covered by the rules (34), (37) and (40).

The side constraints on rules (32), (33) and (34) are necessary to make the specialization step sound: they guarantee the condition $\varrho \notin \mathit{frv}\,(\mathit{TE}, \mu)$ in type rule *(regabst-d)*. These constraints can always be satisfied by α-renaming ϱ in the static answer $\Lambda^{\mathsf{d}}\,\varrho.\,\mathrm{Q}^{\mathsf{s}}$. On an expression-variable level, α-renaming x and f in the rules (36) and (37), and (39) and (40) respectively, can always satisfy the conditions.

Specialization generalizes the standard semantics of λ^{cregion}-terms to $\lambda 2^{\text{region}}$-terms. It is easy to extract an operational semantics for λ^{cregion} from the rules in Figure 3. A value v in λ^{cregion} has the form $\langle c\rangle_{|\rho|}$ or $\langle \lambda\, x.\, e\rangle_{|\rho|}$ for some λ^{cregion}-expression e. Each rule $\delta, E \rightsquigarrow \delta', E'$ in Figure 3 gives now rise to a standard semantics rule $|E| \rightarrow |E'|$, where we replace all occurrences of $|Q|$ in $|E|$ and $|E'|$ with v. So, $\rightarrow$ is a binary relation on λ^{cregion}-terms.

The reflexive transitive closure of the specialization relation $\rightsquigarrow$ is denoted by $\overset{*}{\rightsquigarrow}$ and similarly, we write $\overset{*}{\rightarrow}$ for the reflexive transitive closure of the reduction relation $\rightarrow$ for λ^{cregion}.

To specialize a λ^{region}-program e, we need a constraint completion $\mathbf{e}$ (see section 2.4) with associated region annotation $\delta_c = \{\varrho_1 \mapsto \mathsf{d}, \ldots, \varrho_n \mapsto \mathsf{d}\}$. The actual input to the specializer is the region annotation $\delta_{\text{init}} + \delta_c$ with the $\lambda_C^{\text{region}}$-program $\mathbf{e}$. As we shall prove in section 4, our operational specialization semantics either does not terminate or produces a residual program R for such a completion, *i.e.* $\delta_{\text{init}} + \delta_c, \mathbf{e} \overset{*}{\rightsquigarrow} \delta', R$. In section 5 we also show that $|R| \in \lambda^{\text{region}}$ is contextually equivalent to e.

3.3 Examples

We will illustrate specialization by means of two small examples in an ML-typeable applied lambda calculus, extended with an `if`-construct. The semantics of `if` is standard, *i.e.* we branch to `else` when its condition evaluates to 0 and to `then` otherwise.

Example The first example illustrates a typical specialization process. Consider the following expression, where we assume that d is some dynamic input. Additionally, we have inserted `lift`-expressions aroung some expressions of type `int`, where we remind the reader that in a semantics without regions, the `lift`-construct coincides with the identity function:

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

Recall that, to obtain optimal specialization [29], a preprocessor has to insert `lift` operations around *all* expressions of type `int`. However, to maintain readability, we omit most of these `lift`s and insert just a few for exemplification. As only two `lift`-expressions turn out to be actual coercions. As a first step, we do region inference[4]:

$$
\begin{array}{l}
e = \lambda d.\, (\Lambda\, \varrho_4, \varrho_5, \varrho_{18}.\\
\qquad\qquad \texttt{letrec}\ \mathit{apply} = \Lambda\, \varrho_6, \varrho_7, \varrho_8, \varrho_9.\, \lambda f.\, (\lambda x.\, (f \mathbin{@} x)\ \texttt{at}\ \varrho_6)\ \texttt{at}\ \varrho_5\ \texttt{in}\\
\qquad\qquad\quad \texttt{let}\ y = \Lambda\, \varrho_{10}, \varrho_{11}, \varrho_{12}.\, (\Lambda\, \varrho_{13}.\, (\mathit{apply}\, [\varrho_{10}, \varrho_{11}, \varrho_4, \varrho_{12}]\ \texttt{at}\ \varrho_{13})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbin{@} (\lambda n.\, (5\ \texttt{at}\ \varrho_4)\ \texttt{at}\ \varrho_{11}))\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbin{@} (\Lambda\, \varrho_{14}.\, \texttt{lift}\, [\varrho_{14}, \varrho_{12}]\, (3\ \texttt{at}\ \varrho_{14}))\\
\qquad\qquad\quad \texttt{in let}\ z = \Lambda\, \varrho_{15}, \varrho_{16}.\, (\Lambda\, \varrho_{17}.\, (\mathit{apply}\, [\varrho_{15}, \varrho_{16}, \varrho_{18}, \varrho_{18}]\ \texttt{at}\ \varrho_{17})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbin{@} (\lambda m.\, m\ \texttt{at}\ \varrho_{16}))\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbin{@} (\texttt{lift}\, [\varrho_2, \varrho_{18}]\, d)\\
\qquad\qquad\qquad \texttt{in if}\ z\ \texttt{then lift}\, [\varrho_4, \varrho_3]\, y\\
\qquad\qquad\qquad\qquad\quad \texttt{else lift}\, [\varrho_4, \varrho_3]\, (0\ \texttt{at}\ \varrho_4))\ \texttt{at}\ \varrho_1
\end{array}
$$

The type with place of e is $((\texttt{int}, \varrho_2) \xrightarrow{\epsilon.\varphi} (\texttt{int}, \varrho_3), \varrho_1)$ where the arrow effect φ is $\{\varrho_2, \varrho_3\}$ and e's effect $\{\varrho_1\} \cup \varphi$. Next, we apply a constraint analysis on the region-annotated version of our program, which returns:

$$
\begin{array}{l}
\mathbf{e} = \lambda d.\, (\Lambda\, \varrho_4 : \{\varrho_4 \leq \varrho_3\}\, .\\
\qquad\quad \Lambda\, \varrho_5 : \{\varrho_5 \leq \varrho_4, \varrho_5 \leq \varrho_2, \varrho_5 \leq \varrho_3\}\, .\\
\qquad\quad \Lambda\, \varrho_{18} : \{\varrho_2 \leq \varrho_{18}, \varrho_5 \leq \varrho_{18}\}\, .\\
\qquad\quad \texttt{letrec}\ \mathit{apply} = \Lambda\, \varrho_6, \varrho_7, \varrho_8, \varrho_9.\, \lambda f.\, (\lambda x.\, (f \mathbin{@} x)\ \texttt{at}\ \varrho_6)\ \texttt{at}\ \varrho_5\ \texttt{in}\\
\qquad\qquad \texttt{let}\ y = \Lambda\, \varrho_{10} : \{\varrho_5 \leq \varrho_{10}, \varrho_{10} \leq \varrho_4\}\, .\\
\qquad\qquad\qquad\quad \Lambda\, \varrho_{11} : \{\varrho_5 \leq \varrho_{11}, \varrho_{11} \leq \varrho_4\}\, .\\
\qquad\qquad\qquad\quad \Lambda\, \varrho_{12} : \{\varrho_5 \leq \varrho_{12}, \varrho_{11} \leq \varrho_{12}, \varrho_{10} \leq \varrho_{12}\}\, .\\
\qquad\qquad\qquad\quad (\Lambda\, \varrho_{13} : \left\{\begin{array}{l} \varrho_{13} \leq \varrho_5, \varrho_5 \leq \varrho_{13}, \varrho_{13} \leq \varrho_{10},\\ \varrho_{13} \leq \varrho_{11}, \varrho_{13} \leq \varrho_4, \varrho_{13} \leq \varrho_{12} \end{array}\right\}.\\
\qquad\qquad\qquad\qquad (\mathit{apply}\, [\varrho_{10}, \varrho_{11}, \varrho_4, \varrho_{12}]\ \texttt{at}\ \varrho_{13})\\
\qquad\qquad\qquad\qquad \mathbin{@} (\lambda n.\, (5\ \texttt{at}\ \varrho_4)\ \texttt{at}\ \varrho_{11}))\\
\qquad\qquad\qquad\quad \mathbin{@} (\Lambda\, \varrho_{14} : \{\varrho_{14} \leq \varrho_{12}\}\ .\, \texttt{lift}\, [\varrho_{14}, \varrho_{12}]\, (3\ \texttt{at}\ \varrho_{14}))\\
\qquad\qquad \texttt{in let}\ z = \Lambda\, \varrho_{15} : \{\varrho_5 \leq \varrho_{15}\}\, .\\
\qquad\qquad\qquad\quad \Lambda\, \varrho_{16} : \{\varrho_5 \leq \varrho_{16}, \varrho_{16} \leq \varrho_2, \varrho_{16} \leq \varrho_{18}\}\, .\\
\qquad\qquad\qquad\quad (\Lambda\, \varrho_{17} : \left\{\begin{array}{l} \varrho_{17} \leq \varrho_5, \varrho_5 \leq \varrho_{17}, \varrho_{17} \leq \varrho_{18},\\ \varrho_{17} \leq \varrho_{15}, \varrho_{17} \leq \varrho_{16} \end{array}\right\}.\\
\qquad\qquad\qquad\qquad (\mathit{apply}\, [\varrho_{15}, \varrho_{16}, \varrho_{18}, \varrho_{18}]\ \texttt{at}\ \varrho_{17})\\
\qquad\qquad\qquad\qquad \mathbin{@} (\lambda m.\, m\ \texttt{at}\ \varrho_{16}))\\
\qquad\qquad\qquad\quad \mathbin{@} (\texttt{lift}\, [\varrho_2, \varrho_{18}]\, d)\\
\qquad\qquad\quad \texttt{in if}\ z\ \texttt{then lift}\, [\varrho_4, \varrho_3]\, y\\
\qquad\qquad\qquad\qquad \texttt{else lift}\, [\varrho_4, \varrho_3]\, (0\ \texttt{at}\ \varrho_4))\ \texttt{at}\ \varrho_1
\end{array}
$$

The initial constraint set is $C = \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \varrho_3\}$, the effect is $\varphi = \{\varrho_2, \varrho_3\}$ and the region annotation associated with the constraint completion is $\delta_c =$

[4] The annotations are obtained by simplifying and renaming intermediate code produced by the ML-kit [59]. Some abstracted regions do not appear in the body of the expression because region inference also abstracts regions that are only present in the type.

$\{\varrho_1 \mapsto \mathsf{d}, \varrho_2 \mapsto \mathsf{d}, \varrho_3 \mapsto \mathsf{d}\}$, since we have $\mathit{frv}\,(C, ((\mathtt{int}, \varrho_2) \xrightarrow{\epsilon.\varphi} (\mathtt{int}, \varrho_3), \varrho_1)) \cup \{\varrho_1\} \cup \varphi = \{\varrho_1, \varrho_2, \varrho_3\}$. Obviously we have that $\delta_{\text{init}} + \delta_c \models C$ and we can specialize $\mathbf{e}$ with $\delta_{\text{init}} + \delta_c$ to obtain the following residual program:

$$\lambda\, d. (\underline{\Lambda}\, \varrho_{18}.\, \mathtt{let}\ z = \underline{\mathtt{lift}}\,[\varrho_2, \varrho_{18}]\, d\ \mathtt{in}\ \underline{\mathtt{if}}\ z\ \underline{\mathtt{then}}\ (5\ \underline{\mathtt{at}}\ \varrho_3)\ \underline{\mathtt{else}}\ (0\ \underline{\mathtt{at}}\ \varrho_3))\ \underline{\mathtt{at}}\ \varrho_1$$

Note the binding-time polymorphism in the argument and result of the applications of *apply*.

Example In the second example we show that the conservative treatment of the effect in the $\lambda_C^{\text{region}}$-rule *(Abstr)* is necessary to obtain sound specialization. Consider the program:

$$\begin{array}{l}(\mathtt{let}\ g = \lambda\, z.\, z\\ \ \mathtt{in}\ \lambda\, y.\, \mathtt{let}\ f = \lambda\, x.\, (g\, @\, (\lambda\, y.\, y))\\ \qquad\qquad \mathtt{in}\ (\lambda\, f.\, 4)\, @\, f)\, @\, 2\end{array}$$

The following region-annotated version is typeable in the standard region type system such as the one described in [61].

$$\begin{array}{l}\Lambda\, \varrho_1.\, ((\Lambda\, \varrho_2.\, \mathtt{let}\ g = \lambda\, z.\, z\ \mathtt{at}\ \varrho_2\\ \qquad\qquad \mathtt{in}\ \lambda\, y.\, (\mathtt{let}\ f = \lambda\, x.\, (g\, @\, (\lambda\, y.\, y\ \mathtt{at}\ \varrho_3))\ \mathtt{at}\ \varrho_3\\ \qquad\qquad\qquad \mathtt{in}\ (\lambda\, f.\, (4\ \mathtt{at}\ \varrho_3)\ \mathtt{at}\ \varrho_1)\, @\, f)\ \mathtt{at}\ \varrho_1)\, @\, (2\ \mathtt{at}\ \varrho_1))\end{array}$$

The problem in this example is the function bound to f. It is effectively *dead* code, which means that it is only allocated but never used. Standard region inference in the Tofte-Talpin system is clever enough to see this, in particular, the region bound to ϱ_2 is deallocated before the closure for the function f is allocated. This behavior is sound in $\to$, but is not in $\rightsquigarrow$ if the region ϱ_3 is dynamic. In the latter case, the closure bound to g will be deallocated before specialization tries to reduce under the dynamic function f, causing a dereference of a dangling pointer. The adapted type-system avoids this undesirable behavior, by not typing the above example. It imposes a slightly more conservative annotation:

$$\begin{array}{l}\Lambda\, \varrho_1.\, \Lambda\, \varrho_2.\, ((\mathtt{let}\ g = \lambda\, z.\, z\ \mathtt{at}\ \varrho_2\\ \qquad\qquad \mathtt{in}\ \lambda\, y.\, (\mathtt{let}\ f = \lambda\, x.\, (g\, @\, (\lambda\, y.\, y\ \mathtt{at}\ \varrho_3))\ \mathtt{at}\ \varrho_3\\ \qquad\qquad\qquad \mathtt{in}\ (\lambda\, f.\, (4\ \mathtt{at}\ \varrho_3)\ \mathtt{at}\ \varrho_1)\, @\, f)\ \mathtt{at}\ \varrho_1)\, @\, (2\ \mathtt{at}\ \varrho_1))\end{array}$$

This λ^{region}-program specializes fine when ϱ_1 and ϱ_2 are static and ϱ_3 is dynamic. We omit the details.

3.4 Extending the Static Semantics

To prove soundness of the constraint analysis and specialization, we have to extend the static semantics of section 2 to all terms of $\lambda 2^{\text{region}}$. All semantic objects retain their definitions, even though region placeholders are extended to deallocated and binding-time regions.

$$\textit{(Const-val)}\quad \frac{C \vdash_{\mathit{wft}} (\mathtt{int}, \rho) \quad C \rhd \{\rho \le \mathsf{s}\}}{\mathit{TE}, C \vdash_c \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset}$$

$$\textit{(Abstr-val)}\quad \frac{\begin{array}{c}\mathit{TE} + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi \\ C \vdash_{\mathit{wft}} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho) \quad \varphi \subseteq \varphi' \quad C \rhd \{\rho \le \mathsf{s}\}\end{array}}{\mathit{TE}, C \vdash_c \langle \lambda x.\, E \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset}$$

$$\textit{(Letrec-val)}\quad \frac{\begin{array}{c}P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\epsilon\}) \cap (\mathit{fv}\,(\mathit{TE}) \cup \{\rho\}) = \emptyset \quad \sigma = \forall\, \boldsymbol{\alpha} . \hat{\sigma} \\ \{\boldsymbol{\alpha}\} \cap \mathit{ftv}\,(\mathit{TE}) = \emptyset \quad \mathit{TE} + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset \\ \hat{\sigma} = \forall\, \boldsymbol{\epsilon} . \forall\, \varrho_1, \ldots, \varrho_n . C^{\uparrow P} \Rightarrow \tau \quad \mathit{TE} + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi\end{array}}{\mathit{TE}, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_2 : \mu, \varphi}$$

$$\textit{(Recvar-val)}\quad \frac{\begin{array}{c}\mathit{TE}, C' \vdash_c \langle \lambda x.\, E \rangle_{\rho'} : (\tau, \rho'), \emptyset \quad S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \\ \{\varrho_1, \ldots, \varrho_n\} \cap (\mathit{frv}\,(\mathit{TE}) \cup \{\rho, \rho'\}) = \emptyset \quad \tau' = S_r(\tau) \\ C \rhd S_r(C') \cup \{\rho' \le \rho, \rho \le \rho', \rho' \le \rho'_1, \ldots, \rho' \le \rho'_n\}\end{array}}{\mathit{TE}, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E \rangle_\rho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

$$\textit{(Constr-elim)}\quad \frac{\mathit{TE}, C \vdash_c E : \mu, \varphi \quad C \rhd \{\varrho^\bullet \le \mathsf{s}\}}{\mathit{TE}, C_{\downarrow \varrho^\bullet} \vdash_c E : \mu, \varphi \setminus \{\varrho^\bullet\}}$$

$$\textit{(Regabst-s)}\quad \frac{\mathit{TE}, C \vdash_c E : \mu, \varphi \quad \varrho \notin \mathit{frv}\,(\mathit{TE}, \mu) \quad C \rhd \{\varrho \le \mathsf{s}\}}{\mathit{TE}, C_{\downarrow \varrho} \vdash_c \Lambda^{\mathsf{s}}\, \varrho.\, E : \mu, \varphi \setminus \{\varrho\}}$$

Fig. 6. Extra Constraint Rules - I

The extension of placeholders makes the type rules of Figure 1 to work over deallocated regions as well. Additionally, we assume that all occurrences of **e** in these type rules can be read as $\lambda 2^{\text{region}}$-terms E. The terms unaccounted for are listed in the Figures 6 and 7.

The rules *(Const-val)*, *(Abstr-val)* and *(Letrec-val)* are the pointer counterparts of rules *(Const)*, *(Abstr)* and *(Letrec)* in Figure 1 respectively. In contrast to the allocating expressions, pointers have no effect. Moreover, the rules *(Const-val)* and *(Abstr-val)* include binding-time constraints which guarantee that the top-level region of the value is static for any solution of C. This is motivated by the fact that pointers are computational and only occur during specialization. The rule *(Recvar-val)* types the application of a region closure and is quite similar to rule *(Recvar)*. For more elaboration, we refer to [28].

The rule *(Constr-elim)* is an auxiliary rule, which removes constraints involving deallocated regions. However, this is only allowed if the constraint $\varrho^\bullet \le \mathsf{s}$ is in C. This is necessary to guarantee that regions that previously had to be static because they were *younger* than $\varrho^\bullet$, still have static binding-time for any solution of $C_{\downarrow \varrho^\bullet}$. A static region abstraction is typed by rule *(Regabst-s)* and is almost identical to rule *(Regabst)*, except for the additional binding-time constraint which guarantees that the abstracted region is statically annotated.

$$\textit{(Regabst-d)} \quad \frac{TE, C \vdash_c E : \mu, \varphi \quad \varrho \notin \mathit{frv}\,(TE, \mu) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow \varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.\, E : \mu, \varphi \setminus \{\varrho\}}$$

$$\textit{(Let-d)} \quad \frac{TE, C \vdash_c R : \mu', \varphi' \quad TE + \{x \mapsto \mu'\}, C \vdash_c E : \mu, \varphi}{TE, C \vdash_c \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'}$$

$$\textit{(Letrec-d)} \quad \frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\boldsymbol{\epsilon}\}) \cap (\mathit{fv}\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \boldsymbol{\epsilon}\,.\forall\, \varrho_1, \ldots, \varrho_n\,.C^{\uparrow P} \Rightarrow \tau \quad \{\boldsymbol{\alpha}\} \cap \mathit{ftv}\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \lambda x.\, R\ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \quad C \rhd \{\mathtt{d} \leq \rho\} \\ \sigma = \forall\, \boldsymbol{\alpha}\,.\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, R\ \mathtt{at}\ \rho\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'}$$

$$\textit{(Const-r)} \quad \frac{C \vdash_{\mathit{wft}} (\mathtt{int}, \varrho) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C \vdash_c c\ \underline{\mathtt{at}}\ \varrho : (\mathtt{int}, \varrho), \{\varrho\}}$$

$$\textit{(Abstr-r)} \quad \frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c R : \mu_2, \varphi \\ C \vdash_{\mathit{wft}} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho) \quad \varphi \subseteq \varphi' \quad C \rhd \{\mathtt{d} \leq \varrho\} \end{array}}{TE, C \vdash_c \underline{\lambda} x.\, R\ \underline{\mathtt{at}}\ \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi \cup \{\varrho\}}$$

$$\textit{(Recvar-r)} \quad \frac{\begin{array}{c} TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \boldsymbol{\alpha}\,.\forall\, \boldsymbol{\epsilon}\,.\forall\, \varrho_1, \ldots, \varrho_n\,.C'' \Rightarrow \tau \\ (C', \tau') \prec \sigma\ \mathit{via}\ (S_t, S_e, S_r) \quad S_r = \{\varrho_1 \mapsto \varrho'_1, \ldots, \varrho_n \mapsto \varrho'_n\} \\ C \rhd C' \cup \{\varrho' \leq \rho, \rho \leq \varrho', \varrho' \leq \varrho'_1, \ldots, \varrho' \leq \varrho'_n, \mathtt{d} \leq \varrho'\} \end{array}}{TE, C \vdash_c f\ [\varrho'_1, \ldots, \varrho'_n]\ \underline{\mathtt{at}}\ \varrho' : (\tau', \varrho'), \{\rho, \varrho'\}}$$

$$\textit{(Lift-r)} \quad \frac{TE, C \vdash_c R : (\mathtt{int}, \varrho_1), \varphi \quad C \rhd \{\varrho_1 \leq \varrho_2, \mathtt{d} \leq \varrho_1\}}{TE, C \vdash_c \underline{\mathtt{lift}}\, [\varrho_1, \varrho_2]\, R : (\mathtt{int}, \varrho_2), \varphi \cup \{\varrho_1, \varrho_2\}}$$

$$\textit{(Regabst-r)} \quad \frac{TE, C \vdash_c R : \mu, \varphi \quad \varrho \notin \mathit{frv}\,(TE, \mu) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow \varrho} \vdash_c \underline{\Lambda}\, \varrho.\, R : \mu, \varphi \setminus \{\varrho\}}$$

Fig. 7. Extra Constraint Rules - II

Similarly, rule *(Regabst-d)* in Figure 7 types a dynamic region abstraction. The additional constraint guarantees that ϱ has dynamic binding time. The rule *(Let-d)* is identical to rule *(Let)* and, with respect to rule *(Letrec)* in Figure 1, rule *(Letrec-d)* only adds a binding-time constraint, imposing a dynamic `letrec`-bound function.

The type rules for residual expressions are also very similar to their non-residual counterparts from Figure 1. We only list those rules that slightly deviate from their non-residual counterpart. The rules *(Const-r)* and *(Abstr-r)* add the constraint $\mathbf{d} \leq \varrho$ to their constraint sets. Similarly, the rules *(Recvar-r)*, *(Lift-r)* and *(Regabst-r)* add a constraint to guarantee that every solution for C maps the top-level region onto d. At this point, it may be tempting to empty effects originating from residual syntax since these are not affected during specialization. However, in order to prove the specializer sound with respect to a standard semantics, we have to guarantee that the erasures of the above rules conservatively approximate standard type rules.

Under the extended type system for $\lambda 2^{\text{region}}$, it is easy to verify that the lemmas of section 2.3 remain valid. It is also trivial to verify that whenever $TE, C \vdash_c E : \mu, \varphi$ holds, then $|TE| \vdash_t |E| : |\mu|, |\varphi|$ holds too, where we assume similar extended type rules for λ^{cregion}.

4 Soundness of the Constraint Analysis

The correctness proof of the constraint-analysis closely follows the proof structure in [28], extended to $\lambda 2^{\text{region}}$. It relies crucially on two substitution lemmas. They show that a syntactic substitution in an expression does not affect the expression's type.

We observe that values have no effect.

Lemma 11. *If $TE, C \vdash_c V : \mu, \varphi$ then $\varphi = \emptyset$.*

Under certain conditions, we have that residual code has a top-level region variable which can only be dynamic.

Lemma 12. *Suppose $TE, C \vdash_c R : (\tau, \varrho), \varphi$; $C \vdash_{wft} TE$; $\delta \models C$ and for all $(\sigma, \rho') \in Ran(TE)$, it holds that $\delta(\rho') = \mathsf{d}$. Then, $\delta(\varrho) = \mathsf{d}$.*

Proof. Induction on $TE, C \vdash_c R : (\tau, \varrho), \varphi$.

The substitution lemma for `let`-bound variables is mostly standard (see for instance Lemma 4.4 in [62]).

Lemma 13 (First Substitution Lemma). *Suppose $TE + \{x \mapsto \mu\}, C \vdash_c E : \mu', \varphi'$ and $TE, C \vdash_c V : \mu, \emptyset$, then $TE, C \vdash_c E[x \mapsto V] : \mu', \varphi'$.*

Proof. By induction on the derivation of $TE + \{x \mapsto \mu\}, C \vdash_c E : \mu', \varphi'$. The only interesting case is the application of *(Var)* to (free occurrences of) x. Suppose $TE' = TE + \{x \mapsto \mu\}$, then because $TE'(x) = TE + \{x \mapsto \mu\}(x) = \mu$ and rule *(Var)* we have $TE', C \vdash_c x : \mu, \emptyset$. On the other hand, $x[x \mapsto V] = V$ and, by the second assumption, $TE, C \vdash_c V : \mu, \emptyset$ since $\varphi' = \emptyset$, by Lemma 11.

All other cases are simple appeals to the induction hypothesis.

Substitution of variables bound to region polymorphic expressions is more technical.

Lemma 14 (Second Substitution Lemma). *Suppose*

1. $P = \{\varrho_1 \ldots, \varrho_n\}$ *and* $(P \cup \{\boldsymbol{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset$ *and* $\{\boldsymbol{\alpha}\} \cap ftv\,(TE) = \emptyset$ *and* $\sigma = \forall\, \boldsymbol{\alpha}\,.\hat{\sigma}$ *and* $\hat{\sigma} = \forall\, \boldsymbol{\epsilon}\,.\forall\, \varrho_1, \ldots, \varrho_n\,.C^{\uparrow P} \Rightarrow \tau$
2. $TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$
3. $TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset$

then $TE, C_{\downarrow P} \vdash_c E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho] : \mu, \varphi$ *where* $A = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E_1 \rangle_\rho$.

Proof. The proof is an adaptation of the proof for Lemma 5.6 in [28], extended to handle constraint sets. It can be found in Appendix B

Type preservation states that an expression's type is unaffected by specialization, given a region annotation that includes the initial region annotation δ_{init} and solves the constraint set associated with the expression. In addition, we show that the new effect is smaller, the new constraint set and region annotation larger, and the latter still solves the former.

Proposition 1 (Type preservation). *Suppose* $TE, C \vdash_c E^\mu : \mu, \varphi$ *and* $\delta \models C$ *for* $\delta_{\text{init}} \subseteq \delta$. *If* $\delta, E^\mu \rightsquigarrow \delta', E'$ *then there exists a* C' *such that* $TE, C' \vdash_c E' : \mu, \varphi'$ *where* $\varphi' \subseteq \varphi$; $\delta' \models C'$; $C \subseteq C'$ *and* $\delta \subseteq \delta'$.

The fact that both the constraint set C and region annotation δ grow is a consequence of binding-time calculations during specialization. The new constraint set C' adds at most constraints of the form $\rho \leq \boldsymbol{s}$ or $\boldsymbol{d} \leq \rho$ to reflect binding-time knowledge into the type system: in particular one could prove that $C_{\downarrow\{\boldsymbol{s}, \boldsymbol{d}\}} = C'_{\downarrow\{\boldsymbol{s}, \boldsymbol{d}\}}$.

Proof. See Appendix C

The *canonical forms* lemma specifies the form of an answer if its type is known.

Lemma 15 (Canonical Forms). *Suppose* $TE, C \vdash_c Q^\mu : \mu, \varphi$; $\mu = (\tau, \rho)$; $C \vdash_{wft} TE$; $\delta \models C$ *and for all* $(\sigma, \rho') \in Ran(TE)$, *it holds that* $\delta(\rho') = \mathsf{d}$, *then*

1. *if* $\delta(\rho) = \mathsf{d}$, *then* Q^μ *is residual code of the form* R
2. *if* $\delta(\rho) = \mathsf{s}$, *then* Q^μ *is a static answer* Q^{s}. *Additionally, if* Q^{s} *is a value* V, *then we also have one of the following:*
   (a) *if* $\tau = \texttt{int}$, *then* V *has the form* $\langle c \rangle_\rho$ *for some constant* c
   (b) *if* $\tau = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$, *then* V *has the form* $\langle \lambda\, x.\, E \rangle_\rho$ *for some* x *and* E.

Proof. The proof is by induction on the derivation of $TE, C \vdash_c Q^\mu : \mu, \varphi$. The assumption that $\delta \models C$ and the binding-time constraints in the rules of Figure 6 are critically used to identify canonical forms.

Given a well-typed expression E, and some additional conditions, it is always possible to make a specialization step unless E is an answer.

Proposition 2 (Progress). *Suppose for a $\lambda 2^{\text{region}}$-term E^μ that $TE, C \vdash_c E^\mu : \mu, \varphi$. Additionally, assume that $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$; $Clean(\varphi)$; $C \vdash_{wft} TE$; and for all $(\sigma, \rho') \in Ran(TE)$, it holds that $\delta(\rho') = \mathsf{d}$. Then we either have that*

1. *there exists an E' and δ' such that $\delta, E^\mu \rightsquigarrow \delta', E'$; or*
2. *E is an answer,* i.e. *of the form Q*

The requirement that the environment only contains expressions living in dynamic regions reflects the fact that during specialization we immediately reduce static expression binders such as `let` and lambda. However, since we reduce inside dynamic lambda and `let` expressions, we cannot assume that the type environment is empty as it is usually the case in progress proofs. The condition $Clean(\varphi)$ avoids that deallocated regions or the global static or dynamic region are touched during reduction.

Proof. See Appendix D

Theorem 2 (Constraint Analysis Soundness). *Suppose E^μ is a $\lambda 2^{\text{region}}$-program for which $\{\ \}, C \vdash_c E^\mu : \mu, \varphi$. Additionally, assume that $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$ and that $Clean(\varphi)$, then either*

1. *there exists an answer Q, a region annotation δ', constraint set C' and effect φ' such that $\delta, E \overset{*}{\rightsquigarrow} \delta', Q$ and $\{\ \}, C' \vdash_c Q : \mu, \emptyset$ for which $\delta' \models C'$; $\varphi' \subseteq \varphi$; and $\delta \subseteq \delta'$; or*
2. *for each E' and δ' where $\delta, E \overset{*}{\rightsquigarrow} \delta', E'$, there exist E'' and δ'' with $\delta', E' \rightsquigarrow \delta'', E''$.*

Proof. Assume $\delta, E \overset{*}{\rightsquigarrow} \delta', E'$. We proceed by induction on the length of the reduction. For the base case, assume that $E = E'$. Then, by proposition 2 (progress) either item 1 or 2 applies. Alternatively, we have $\delta, E \rightsquigarrow \delta'', E''$ and $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', E'$. By proposition 1 (type preservation), we can apply the induction hypothesis on $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', E'$ by which item 1 or 2 applies, proving the theorem.

As a corollary of the soundness theorem and the canonical forms lemma, we can derive that specialization of a constraint completion always yields a residual program, provided partial evaluation terminates.

Corollary 1 (Residual Code). *Suppose e is a constraint completion with region annotation δ_c,* i.e. *$\{\ \}, C \vdash_c \mathsf{e}^\mu : \mu, \varphi$; $\delta_c \models C$ and for all $\rho' \in (frv\,(C, \mu) \cup \varphi)$ holds $\delta_c(\rho') = \mathsf{d}$.*

Then, either we have non-termination of specialization or we obtain a residual program R with region annotation δ' such that $\delta_{\text{init}} + \delta_c, E \overset{*}{\rightsquigarrow} \delta', R$.

Proof. Take $\delta = \delta_{\text{init}} + \delta_c$ and $\mu = (\tau, \rho)$. The condition that for all $\rho' \in (frv\,(C, \mu) \cup \varphi)$ holds $\delta_c(\rho') = \mathsf{d}$, makes $Clean(\varphi)$, $\delta \models C$ and $\delta(\rho) = \mathsf{d}$. Because of constraint-analysis soundness, we either have non-termination or $\delta, \mathsf{e} \overset{*}{\rightsquigarrow} \delta', Q$, where $\{\,\}, C' \vdash_c Q : (\tau, \varrho), \varphi'$ holds with $\delta' \models C'$; $\varphi' \subseteq \varphi$; and $\delta \subseteq \delta'$. Since $\delta'(\varrho) = \mathsf{d}$ (as a result of $\delta \subseteq \delta'$), we can apply the canonical forms lemma (item 1), guaranteeing that Q is of the form R, *i.e.* residual code.

A residual program resulting from the specialization of a constraint completion is again amenable to specialization since we have that $\{\,\}, C \vdash_c R : \mu, \varphi$ implies $\{\,\} \vdash_t |R| : |\mu|, |\varphi|$.

5 Soundness of Specialization

The previous section shows that the constraint analysis inserts annotations such that the specializer does not go *wrong* in the sense of Milner [39]. Full specialization correctness requires the residual program to be *semantically equivalent* to the source program, *i.e.* we have to show that the reductions of the specializer are contextual equivalences in the region calculus. Elsewhere [26], we have developed an equational theory for λ^{cregion} and proved it sound with respect to Morris-style contextual equivalence [42]. The soundness proof of the specializer crucially relies on these results.

5.1 An Equational Theory for $\boldsymbol{\lambda}^{\text{cregion}}$

Before formulating the soundness theorem, we summarize the equational theory [26] for λ^{cregion} and some properties.

The equational theory is formulated for a slightly stronger language than λ^{cregion} since it treats recursive functions as first class objects. However, `letrec`-terms and `letrec`-bound variables are an instance of this general formulation (see section 2.4. in [26]). Moreover, for brevity, we will restrict ourselves to the λ^{cregion}-calculus without polymorphism and constants.

As mentioned in section 2.2, the type system used by the constraint analysis is slightly more conservative than the usual region type system given in the literature [26,28,61]. Therefore, we define a type system for the judgment $TE \vdash^t_t e : \mu, \varphi$ with identical rules as for $TE \vdash_t e : \mu, \varphi$, except that rule *(Abstr)* is replaced by

$$(\textit{Abstr-TT}) \qquad \frac{TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash^t_t \lambda x.\, e \;\texttt{at}\; \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}}$$

The only difference is that the effect of the lambda does not contain φ. We have the following straightforward property:

Proposition 3. *Suppose $TE \vdash_t e : \mu, \varphi$. Then $TE \vdash^t_t e : \mu, \varphi'$, where $\varphi' \subseteq \varphi$.*

Proof. All cases are straightforward, except for rule *(Abstr)*. So, suppose $TE \vdash_t \lambda x.\, e \;\texttt{at}\; \rho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \rho), \varphi \cup \{\rho\}$ and hence, we have that $TE + \{x \mapsto \mu_1\} \vdash_t e : \mu_2, \varphi$ with $\varphi \subseteq \varphi''$. Using the induction hypothesis, we have that $TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2, \varphi'$ where $\varphi' \subseteq \varphi$. Therefore, we also have $\varphi' \subseteq \varphi''$ and

$$\textit{(Comp-var)} \quad \frac{TE(x) = (\mu, \rho) \quad Clean(\varphi)}{TE \rhd x \,\widehat{\mathcal{R}}\, x : (\mu, \rho), \varphi}$$

$$\textit{(Comp-abstr)} \quad \frac{TE + \{x \mapsto \mu_1\} \rhd e \,\mathcal{R}\, e' : \mu_2, \varphi' \quad Clean(\varphi)}{TE \rhd \lambda\, x.\, e \texttt{ at } \varrho \,\widehat{\mathcal{R}}\, \lambda\, x.\, e' \texttt{ at } \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \{\varrho\} \cup \varphi}$$

$$\textit{(Comp-abstr-val)} \quad \frac{TE + \{x \mapsto \mu_1\} \rhd e \,\mathcal{R}\, e' : \mu_2, \varphi' \quad Clean(\varphi)}{TE \rhd \langle \lambda\, x.\, e \rangle_\rho \,\widehat{\mathcal{R}}\, \langle \lambda\, x.\, e' \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \varphi}$$

$$\textit{(Comp-regabst)} \quad \frac{TE \rhd e \,\mathcal{R}\, e' : \mu, \varphi' \quad \varrho \notin frv\,(TE, \mu) \quad Clean(\varphi)}{TE \rhd \Lambda\, \varrho.\, e \,\widehat{\mathcal{R}}\, \Lambda\, \varrho.\, e' : \mu, (\varphi' \setminus \{\varrho\}) \cup \varphi}$$

$$\textit{(Comp-app)} \quad \frac{TE \rhd e_1 \,\mathcal{R}\, e_1' : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi_1 \quad Clean(\varphi) \quad TE \rhd e_2 \,\mathcal{R}\, e_2' : \mu_1, \varphi_2}{TE \rhd e_1 \,@\, e_2 \,\widehat{\mathcal{R}}\, e_1' \,@\, e_2' : \mu_2, \varphi' \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \varrho\} \cup \varphi}$$

$$\textit{(Comp-let)} \quad \frac{TE \rhd e_1 \,\mathcal{R}\, e_1' : \mu', \varphi_1 \quad Clean(\varphi) \quad TE + \{x \mapsto \mu'\} \rhd e_2 \,\mathcal{R}\, e_2' : \mu, \varphi_2}{TE \rhd \texttt{let } x = e_1 \texttt{ in } e_2 \,\widehat{\mathcal{R}}\, \texttt{let } x = e_1' \texttt{ in } e_2' : \mu, \varphi_1 \cup \varphi_2 \cup \varphi}$$

Fig. 8. Compatible Refinement

we can apply rule *(Abstr-TT)* to obtain $TE \vdash_t^t \lambda\, x.\, e \texttt{ at } \rho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \rho), \{\rho\}$. Obviously it holds that $\{\rho\} \subseteq \{\rho\} \cup \varphi$, proving the case.

Define a *typed relation* $\mathcal{R} \in \mathsf{TypedRel}$ as a relation on quintuples $(TE, e_1, e_2, \mu, \varphi)$ where $\mathsf{TypedRel} = \mathcal{P}(\mathsf{TypeEnv} \times \lambda^{\text{cregion}} \times \lambda^{\text{cregion}} \times \mathsf{TypeWPlace} \times \mathsf{Effect})$. We write $TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi$ to say that $(TE, e_1, e_2, \mu, \varphi) \in \mathcal{R}$.

Definition 2. *A typed relation $\mathcal{R}$ is* well-formed *if the following two conditions hold simultaneously:*

1. *if $TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi$ then $TE \vdash_t^t e_1 : \mu, \varphi_1$ and $TE \vdash_t^t e_2 : \mu, \varphi_2$ such that $\varphi_1 \cup \varphi_2 \subseteq \varphi$ with $Clean(\varphi)$ and $TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi_1 \cup \varphi_2$.*
2. *if $TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi$ then for all sets of effects φ' with $Clean(\varphi')$, also $TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi \cup \varphi'$ holds.*

A typed relation $\mathcal{R}$ is reflexive if $TE \rhd e \,\mathcal{R}\, e : \mu, \varphi$, whenever $TE \vdash_t^t e : \mu, \varphi'$ and $\varphi' \subseteq \varphi$ with $Clean(\varphi)$.

Given a typed relation $\mathcal{R}$, we define its *compatible refinement* $\widehat{\mathcal{R}} \in \mathsf{TypedRel}$ as the least relation closed under the rules of Figure 8. The refinement rules are pretty much binary extensions of the type rules in Figure 1 without rule *(Effect)*.

We say that a typed relation $\mathcal{R}$ is *compatible* iff $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. A compatible relation is a typed relation which has been extended to all typed contexts. We use this notion below to specify an equational theory and contextual equivalence for λ^{cregion}. It is easy to verify that a typed relation $\mathcal{R}$ is reflexive if it is compatible. Moreover, $\widehat{\mathcal{R}}$ is well-formed whenever $\mathcal{R}$ is.

$$\textit{(Eq-comp)}\ \frac{TE \rhd e \mathrel{\widehat{\doteq}} e' : \mu, \varphi}{TE \rhd e \doteq e' : \mu, \varphi} \qquad \textit{(Eq-symm)}\ \frac{TE \rhd e' \doteq e : \mu, \varphi}{TE \rhd e \doteq e' : \mu, \varphi}$$

$$\textit{(Eq-trans)}\ \frac{TE \rhd e \doteq e' : \mu, \varphi \quad TE \rhd e' \doteq e'' : \mu, \varphi}{TE \rhd e \doteq e'' : \mu, \varphi}$$

$$\textit{(Eq-abstr)}\ \frac{TE + \{x \mapsto \mu_1\} \vdash_t^t e : \mu_2, \varphi'' \quad \varphi'' \subseteq \varphi' \quad \varrho \in \varphi \quad \mathit{Clean}(\varphi)}{TE \rhd \lambda x.\, e \ \mathtt{at}\ \varrho \doteq \langle \lambda x.\, e \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi}$$

$$\textit{(Eq-app-cbv)}\ \frac{\begin{array}{c} TE \vdash_t^t \langle \lambda x.\, e_1 \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu, \varrho), \emptyset \\ TE \vdash_t^t e_2 : \mu_1, \varphi' \quad \varphi' \cup \varphi'' \cup \{\varrho, \epsilon\} \subseteq \varphi \quad \mathit{Clean}(\varphi) \end{array}}{TE \rhd \langle \lambda x.\, e_1 \rangle_\varrho \mathbin{@} e_2 \doteq \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : \mu, \varphi}$$

$$\textit{(Eq-let-beta)}\ \frac{TE \vdash_t^t v : \mu', \emptyset \quad TE + \{x \mapsto \mu'\} \vdash_t^t e : \mu, \varphi' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}(\varphi)}{TE \rhd \mathtt{let}\ x = v\ \mathtt{in}\ e \doteq e[x \mapsto v] : \mu, \varphi}$$

$$\textit{(Eq-app-let)}\ \frac{\begin{array}{c} TE \vdash_t^t e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu, \varrho), \varphi_1 \quad TE \vdash_t^t e_2 : \mu_1, \varphi_2 \\ f \notin \mathit{fv}\,(e_2) \quad \varphi' \cup \varphi_1 \cup \varphi_2 \cup \{\varrho, \epsilon\} \subseteq \varphi \quad \mathit{Clean}(\varphi) \end{array}}{TE \rhd e_1 \mathbin{@} e_2 \doteq \mathtt{let}\ f = e_1\ \mathtt{in}\ f \mathbin{@} e_2 : \mu, \varphi}$$

$$\textit{(Eq-let-let)}\ \frac{\begin{array}{c} TE \vdash_t^t e_1 : \mu_1, \varphi_1 \quad x \notin \mathit{frv}\,(e_3) \quad \varphi_1 \cup \varphi_2 \cup \varphi_3 \subseteq \varphi \quad \mathit{Clean}(\varphi) \\ TE + \{x \mapsto \mu_1\} \vdash_t^t e_2 : \mu_2, \varphi_2 \quad TE + \{y \mapsto \mu_2\} \vdash_t^t e_3 : \mu, \varphi_3 \end{array}}{\begin{array}{l} TE \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ (\mathtt{let}\ y = e_2\ \mathtt{in}\ e_3) \doteq \\ \qquad \mathtt{let}\ y = (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ \mathtt{in}\ e_3 : \mu, \varphi \end{array}}$$

$$\textit{(Eq-drop)}\ \frac{TE \vdash_t^t e : \mu, \varphi' \quad \varrho \notin \mathit{frv}\,(TE, \mu) \quad \varphi' \subseteq \varphi \quad \mathit{Clean}(\varphi)}{TE \rhd e \doteq \Lambda \varrho.\, e : \mu, \varphi}$$

$$\textit{(Eq-dealloc)}\ \frac{TE \vdash_t^t e : \mu, \varphi' \quad \varrho \notin \mathit{frv}\,(\mu) \cup \varphi' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}(\varphi)}{TE \rhd e \doteq e[\varrho \mapsto \bullet] : \mu, \varphi}$$

Fig. 9. Equational Theory

We now have sufficient machinery to define a typed equational theory on λ^{cregion}-terms. Define the relation $\doteq$ as the least typed relation closed under the rules in Figure 9. The rules *(Eq-comp)*, *(Eq-symm)* and *(Eq-trans)* build the equivalent and compatible closure. Rules *(Eq-abstr)*, *(Eq-app-cbv)* and *(Eq-let-beta)* are the call-by-value beta-rules from the operational semantics, interpreted as equalities. Rule *(Eq-app-let)* models the fact in λ^{cregion} function application is left-to-right, whereas *(Eq-let-let)* specifies standard let-flattening. Rule *(Eq-drop)* states that a region binder does not influence observable program execution and can be dropped, whereas rule *(Eq-dealloc)* says that a region can be deallocated whenever it is provable dead. It is easy to verify that the equational theory $\doteq$ is well-formed.

Intuitively, two λ^{cregion}-terms are contextual equivalent whenever arbitrary (well-typed and closed) contexts exhibit identical termination behavior. We formalize this with a technique due to Lassen [47], where we say that a λ^{cregion}-term e converges ($e\downarrow$) iff there exists a value v such that $e \xrightarrow{*} v$.

Define the *context closure* $\mathcal{R}^c$ of a typed relation $\mathcal{R}$ as the least relation, closed under the rules

$$\textit{(Closure-1)}\quad \frac{TE \rhd e_1\,\mathcal{R}\,e_2 : \mu,\varphi}{TE \rhd e_1\,\mathcal{R}^c\,e_2 : \mu,\varphi} \qquad\qquad \textit{(Closure-2)}\quad \frac{TE \rhd e_1\,\widehat{\mathcal{R}^c}\,e_2 : \mu,\varphi}{TE \rhd e_1\,\mathcal{R}^c\,e_2 : \mu,\varphi}$$

A context closure closes the elements of a typed relation under all typed contexts.

Definition 3 (Contextual Equivalence). *Contextual equivalence, $\simeq$, is a well-formed typed relation on λ^{cregion}-terms such that $TE \rhd e_1 \simeq e_2 : \mu,\varphi$ iff for all e_1', e_2' μ' and φ' such that $\{\,\} \rhd e_1' \left\{(TE, e_1, e_2, \mu)\right\}^c e_2' : \mu',\varphi'$ it holds that $e_1'\downarrow \Leftrightarrow e_2'\downarrow$.*

The equational theory is sound with respect to contextual equivalence, meaning that terms proven equal in the syntactic theory are also contextual equivalent:

Theorem 3. *If $TE \rhd e_1 \doteq e_2 : \mu,\varphi$, then also $TE \rhd e_1 \simeq e_2 : \mu,\varphi$.*

Proof. The proof uses a theory of bisimilarity and can be found elsewhere [26].

5.2 Specialization Soundness

We now prove that the specialization rules from the Figures 3, 4, and 5 are sound. We need one auxiliary lemma, which says something about the effect of an answer.

Lemma 16. *Suppose $TE, C \vdash_c Q^\mu : \mu,\varphi$ such that $C \vdash_{\mathit{wft}} TE$ and $\delta \models C$ and for all $(\sigma,\rho') \in \mathit{Ran}(TE)$ it holds that $\delta(\rho') = \mathsf{d}$. Then, $TE, C \vdash_c Q^\mu : \mu,\varphi_0$ with $\varphi_0 \subseteq \varphi$ and for all $\rho' \in \varphi_0$ it holds that $\delta(\rho') = \mathsf{d}$.*

Proof. By induction on $TE, C \vdash_c Q^\mu : \mu,\varphi$ using lemma 11.

The following proposition states that every erased specialization step is an equality in the syntactic theory.

Proposition 4. *Suppose $TE, C \vdash_c E : \mu,\varphi$ where $\delta \models C$ and $C \vdash_{\mathit{wft}} TE$ such that for all $(\sigma,\rho') \in \mathit{Ran}(TE)$ it holds that $\delta(\rho') = \mathsf{d}$ and also $\mathit{Clean}(\varphi)$. Then, $\delta, E \rightsquigarrow \delta', E'$ implies $TE \rhd |E| \doteq |E'| : \mu,\varphi$.*

Proof. By proposition 1 (type preservation) we have that $TE, C' \vdash_c E' : \mu,\varphi'$ and $\varphi' \subseteq \varphi$. Therefore, we also have $\mathit{Clean}(\varphi')$. We prove by rule induction on $\rightsquigarrow$ and consider each reduction rule from the Figures 3, 4 and 5, that does not reduce a polymorphic or constant term.

Rule (2). Immediate by *(Eq-abstr)*.

Rule (4). By type rule *(Regabst-s)* we have $TE, C \vdash_c \Lambda^{s}\,\varrho.\,Q : \mu, \varphi_0 \setminus \{\varrho\}$, and hence $TE, C \vdash_c Q : \mu, \varphi_0$ such that $\varrho \notin frv\,(TE, \mu)$ and $C \rhd \{s \leq \varrho\}$. From lemma 16 and the assumptions, we derive that $TE, C \vdash_c \Lambda^{s}\,\varrho.Q : \mu, \varphi_1$ where $\varrho \notin frv\,(TE, \mu) \cup \varphi_1$ and $\varphi_1 \subseteq \varphi_0$. Via proposition 3, this implies $|TE| \vdash_t^t |\Lambda^{s}\,\varrho.Q| : |\mu|, \varphi'_1$ with $\varphi'_1 \subseteq |\varphi_1| \subseteq |\varphi_0|$ and hence also $\varrho \notin frv\,(|TE|, |\mu|) \cup \varphi'_1$. The equality now easily follows from the rules *(Eq-drop)* and *(Eq-dealloc)*.

Rule (6). Immediate by equational rule *(Eq-app-cbv)*.

Rule (7). Follows from rule *(Eq-let-beta)*.

Rules (10), (11), (13) and (14). Immediate applications of the induction hypothesis.

Rules (15), (18), (20), (22), (23) and (25). Here both the left-hand and right-hand side yield identical (erased) expressions and reflexivity of $\doteq$ proves these cases.

Rules (27), (29) and (30). Immediate applications of the induction hypothesis.

Rule (33). Within *(Comp-let)*, apply *(Eq-drop)* via *(Eq-comp)* on the left-hand side. Using *(Eq-drop)* on the right-hand side and combining with *(Eq-trans)* proves the case.

Rule (34). Analogous the previous case but with *(Comp-app)* instead.

Rule (36). Immediate from rule *(Eq-let-let)*.

Rule (37). Use rule *(Eq-app-let)* on the left-hand side, followed by *(Eq-let-let)*. Application of *(Eq-app-let)* then proves the case.

We conclude with the soundness theorem for specialization:

Theorem 4. *Suppose* $\{\,\}, C \vdash_c E : \mu, \varphi$; $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$; *and* $Clean(\varphi)$. *Additionally, we have* $\delta, E \overset{*}{\leadsto} \delta', Q$. *Then, it holds that* $\{\,\} \rhd |E| \simeq |Q| : \mu, \varphi$

Proof. By induction the length of the specialization. The base case is trivial by reflexivity of $\doteq$ and Theorem 3. Otherwise we have that $\delta, E \leadsto \delta'', E''$ and $\delta'', E'' \overset{*}{\leadsto} \delta', Q$. We apply Proposition 4 to obtain $\{\,\} \rhd |E| \doteq |E''| : \mu, \varphi$. Using Proposition 1 (type preservation), we also have a C'' and φ'' such that $\{\,\}, C'' \vdash_c E'' : \mu, \varphi''$ where $\delta \subseteq \delta''$; $C \subseteq C''$; $\varphi'' \subseteq \varphi$ and $\delta'' \models C''$. The induction hypothesis now yields $\{\,\} \rhd |E''| \doteq |Q| : \mu, \varphi$ and we prove the claim by rule *(Eq-trans)* and Theorem 3.

6 Conclusion and Future Work

Building upon the region analysis of Tofte and Talpin, we have presented a simple and general approach for polymorphic specialization of functional languages with an ML-style type discipline. The framework modularizes the problem of binding-time analysis [53] by separating the flow analysis (region inference) from the binding-time annotation phase (constraint analysis). The polymorphic binding-time analysis comprises polymorphic recursion and is proven correct with respect to the specializer. Additionally, we have proven that the specializer preserves the call-by-value semantics of the region calculus.

Some of the attractiveness of using region inference for a binding-time analysis are the coinciding goals of region separation on the one hand and binding-time separation on the other hand. This makes us hope that optimizations that are good for region inference are also good for partial evaluation and the other way around. While we have currently no counter-evidence for the former, the latter turns out to be false, at least from a memory management perspective. It is well-known that a CPS-transformation allows for binding-time improvements [12], but it destroys the memory management goals for the region calculus completely. The problem is that in a CPS-transformed program, values never return and hence, regions are only allocated, not deallocated. However, the actual cause is not the CPS transformation itself, but the stack-based nature of region inference. We conjecture that one could develop a continuation-based region calculus which acts particularly well for a CPS-transformed program. Moreover, when region inference is used solely for a binding-time analysis (as is the case in this paper), it does not really matter when a region is deallocated as long as region separation is still good. More importantly, our specializer implements context shifts that make a CPS-transformation redundant [37], so the impact of this observation on our framework is insignificant.

This paper provides a first step towards a usable program specializer for a real programming language like ML. Whereas the constraint analysis can be readily built on top of region inference (section 2.4 suggests an algorithm), we still have to address the development of a compiler generator or *cogen* [5, 50] The cogen-approach to offline partial evaluation is essential for the effective specialization of ML-like programming languages [4,35]. A cogen for the specialization technique presented here is particular challenging since it may have to resolve statically unknown binding times. Since it is desirable to code a cogen for ML in ML itself, binding times at runtime may cause tagging. This problem may be addressed using a richer type language, for instance with dependent types or even first-class modules.

Although our specializer avoids some code duplication by `let`-insertion – necessary to guarantee sound specialization – code duplication may still occur due to recursively defined functions. In particular, static calls under dynamic control are known to cause infinite specialization. A standard solution to this problem is to *memoize* the function's binding-time description [33, 51] and insert so-called dynamic program points. The current formalization does not cover memoization: It either unrolls recursive functions or keeps them with a specialized body. It should be investigated how to extend our specialization mechanism with memoization.

Regions and effects have been used previously to specialize side-effects on reference cells [52] with destructive updates. Recently we have extended the storeless small-step semantics for λ^{region} from this paper with an explicit store [7]. This operational extension caters for ML-style reference cells and could provide a basis for specialization-time reductions of store operations. However, such additions are not straightforward in practice [55] and may complicate the soundness proof of the specializer considerably.

A Proof of Constraint Completion Existence

Suppose $\{\ \} \vdash_t e : \mu, \varphi$. Then there exists a constraint completion $\mathbf{e}^\mu$ with region annotation δ_c.

Proof. To prove the theorem, we prove a slightly more general claim. If $TE' \vdash_t e : \mu, \varphi$, then for any TE where $|TE| = TE'$, there exist a constraint set C, and $\lambda_C^{\text{region}}$-term $\mathbf{e}$ such that $TE, C \vdash_c \mathbf{e} : \mu, \varphi$ where $|\mathbf{e}| = e$. This is proven by rule induction over $TE' \vdash_t e : \mu, \varphi$.

Case x**.** Take an environment TE with $|TE| = TE'$, constraint set $C = \emptyset$ and $\mathbf{e} = x$.

Case $c \texttt{ at } \rho$**.** For any TE with $|TE| = TE'$, take $C = \emptyset$ and $\mathbf{e} = c \texttt{ at } \rho$ to prove the case.

Case $\lambda x.\, e \texttt{ at } \rho$**.** By rule *(Abstr)* for λ^{region}, we have $TE' + \{x \mapsto \mu_1\} \vdash_t e : \mu_2, \varphi$ where $\varphi \subseteq \varphi'$. Take a TE such that $|TE| = TE'$, so obviously $|TE + \{x \mapsto \mu_1\}| = TE' + \{x \mapsto \mu_1\}$. By induction, there exist a constraint set C and $\mathbf{e}$ such that $TE + \{x \mapsto \mu_1\}, C \vdash_c \mathbf{e} : \mu_2, \varphi$ where $|\mathbf{e}| = e$.

Now, take C_0 such that $C \subseteq C_0$ and $C_0 \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho)$ by adding the well-formedness constraints to C. From Lemma 2 and 4 we now have $TE + \{x \mapsto \mu_1\}, C_0 \vdash_c \mathbf{e} : \mu_2, \varphi$. So, from rule *(Abstr)* for $\lambda_C^{\text{region}}$, we obtain $TE, C_0 \vdash_c \lambda x.\, \mathbf{e} \texttt{ at } \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\} \cup \varphi$.

Case $e_1 @ e_2$**.** By rule *(App)* for λ^{region} we have $TE' \vdash_t e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $TE' \vdash_t e_2 : \mu_1, \varphi_2$. So, take any TE with $|TE| = TE'$ and then by induction constraint sets C_1 and C_2 and $\lambda_C^{\text{region}}$-terms $\mathbf{e}_1$ and $\mathbf{e}_2$ such that $TE', C_1 \vdash_c \mathbf{e}_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $TE', C_2 \vdash_c \mathbf{e}_2 : \mu_1, \varphi_2$. Choose $C = \widetilde{C_0}$, where $C_0 = C_1 \cup C_2$, and so by Lemma 4 we have $TE', C \vdash_c \mathbf{e}_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $TE', C \vdash_c \mathbf{e}_2 : \mu_1, \varphi_2$ from which we prove the case with rule *(App)*.

Case $\texttt{let } x = e_1 \texttt{ in } e_2$**.** Similar inductive reasoning using Lemma 4.

Case $\texttt{lift}[\rho_1, \rho_2]\, e$**.** Similar inductive reasoning using Lemma 4.

Case $\Lambda\, \varrho.\, e$**.** By rule *(Regabst)* for λ^{region}, it must be that $TE' \vdash_t e : \mu, \varphi$ where $\varrho \notin \mathit{frv}\,(TE', \mu)$. Given TE with $|TE| = TE'$, we apply the induction hypothesis to obtain a constraint set C and term $\mathbf{e}$ such that $TE, C \vdash_t \mathbf{e} : \mu, \varphi$ and $|\mathbf{e}| = e$. Obviously $\varrho \notin \mathit{frv}\,(TE, \mu)$ and so by *(Regabst)*, we have $TE, C_{\downarrow\varrho} \vdash_c \Lambda\, \varrho : C^{\uparrow\varrho}.\mathbf{e} : \mu, \varphi \setminus \{\varrho\}$. The claim is proven since $|\Lambda\, \varrho : C^{\uparrow\varrho}.\mathbf{e}| = \Lambda\, \varrho.\, e$.

Case $\texttt{letrec } f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, e_1 \texttt{ at } \rho \texttt{ in } e_2$**.** Rule *(Letrec)* for λ^{region} gives that for $P = \{\varrho_1, \ldots, \varrho_n\}$ we have $(P \cup \{\epsilon\}) \cap (\mathit{fv}\,(TE') \cup \{\rho\}) = \emptyset$; $\hat{\sigma}' = \forall \boldsymbol{\epsilon}. \forall \varrho_1, \ldots, \varrho_n. \tau$; $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}\,(TE') = \emptyset$; $\sigma' = \forall \boldsymbol{\alpha}. \hat{\sigma}'$; $TE' + \{f \mapsto (\hat{\sigma}', \rho)\} \vdash_t \lambda x.\, e_1 \texttt{ at } \rho : (\tau, \rho), \varphi'$; and $TE' + \{f \mapsto (\sigma', \rho)\} \vdash_t e_2 : \mu, \varphi \cup \varphi'$. By induction, for any TE where $|TE| = TE'$ and some $\hat{\sigma}$ such that $|\hat{\sigma}| = \hat{\sigma}'$ we have that there exists a C_1 and $\mathbf{e}_1$ such that $TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C_1 \vdash_c \lambda x.\, \mathbf{e}_1 \texttt{ at } \rho : (\tau, \rho), \varphi'$ and $|\mathbf{e}_1| = e_1$. This holds in particular for $\hat{\sigma} = \hat{\sigma}_1 =$

$\forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.{C_1^0}^{\uparrow P} \Rightarrow \tau$ where $C_1^0 = \emptyset$. This provides the base case for a fixpoint iteration: in each step, repeat the constraint analysis (with fixed region typing) starting from rule *(Letrec)* where $\hat{\sigma}_{i+1} = \forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.{C_1^i}^{\uparrow P} \Rightarrow \tau$ to obtain C_1^{i+1} and $\mathbf{e}_1^{i+1}$.

Since there are only finitely many constraints over finitely many region variables (*i.e.* those that occur in $TE' + \{f \mapsto (\hat{\sigma}', \rho)\} \vdash_t \lambda x.\, e_1 \texttt{ at } \rho : (\tau, \rho), \varphi')$ and C_1^{i+1} grows monotonously with respect to C_1^i, the sequence of C_1^i and $\mathbf{e}_1^i$ reaches a fixpoint C_f and $\mathbf{e}_f$ in finitely many steps. Monotonicity is guaranteed since a constraint set in a typing judgment only becomes smaller if we change the underlying type in λ^{region}: in rule *(Letrec)*, the set P had to be changed and in rule *(Regabst)*, either the type μ or the type environment TE'. This is impossible since we assume that the underlying types are fixed by region inference.

Using constraint set C_f, define $\hat{\sigma} = \forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.C_f^{\uparrow P} \Rightarrow \tau$ and $\sigma_f = \forall \boldsymbol{\alpha}.\hat{\sigma}_f$, where obviously $|\hat{\sigma}_f| = \hat{\sigma}$ and $|\sigma_f| = \sigma$. With this we apply the induction hypothesis again to obtain $TE' + \{f \mapsto (\sigma_f, \rho)\}, C_2 \vdash_c \mathbf{e}_2 : \mu, \varphi \cup \varphi'$. Without loss of generality, we can assume by α-renaming that $P \cap \mathit{frv}\,(C_2) = \emptyset$. For $C_0 = C_f \cup C_2$, take $C = \widetilde{C_0}$ where $C^{\uparrow P} = C_f^{\uparrow P}$ and by Lemma 4, we have $TE + \{f \mapsto (\hat{\sigma}_f, \rho)\}, C \vdash_c \lambda x.\, \mathbf{e}_f \texttt{ at } \rho : (\tau, \rho), \varphi'$.

Since $C^{\uparrow P} = C_f^{\uparrow P}$, we obviously have $C_2 \subseteq C_{\downarrow P}$ and therefore by Lemma 1 and 4 that $TE + \{f \mapsto (\sigma_f, \rho)\}, C_{\downarrow P} \vdash_c \mathbf{e}_2 : \mu, \varphi$. So rule *(Letrec)* is applicable and yields $TE, C_{\downarrow P} \vdash_c \texttt{letrec } f = \Lambda\, \varrho_1,\dots,\varrho_n.\lambda x.\, \mathbf{e}_f \texttt{ at } \rho \texttt{ in } \mathbf{e}_2 : \mu, \varphi \cup \varphi'$, proving the case.

Case $f\,[\rho_1',\dots,\rho_n'] \texttt{ at } \rho$**.** By rule *(Recvar)* for λ^{region} we have that $TE'(f) = (\sigma', \rho)$; $\sigma' = \forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.\tau$; $S_r = \{\varrho_1 \mapsto \rho_1',\dots,\varrho_n \mapsto \rho_n'\}$; and the instance $\tau_0 \prec \sigma'$ *via* (S_t, S_e, S_r). Take any TE such that $|TE| = TE'$. So, $TE(f) = (\sigma, \rho)$ where $\sigma = \forall \boldsymbol{\alpha}.\forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.C_0' \Rightarrow \tau$ for some C_0'. So, we have $(C_0, \tau_0) \prec \sigma$ *via* (S_t, S_e, S_r). Now, take the constraint set C to be $C_0 \cup \{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho_1', \dots, \rho' \leq \rho_n'\}$ and apply *(Recvar)* to prove the case: $TE, \widetilde{C} \vdash_c f\,[\rho_1',\dots,\rho_n'] \texttt{ at } \rho : (\tau_0, \rho'), \{\rho, \rho'\}$.

From this generalized claim, we obtain that for each $\{\,\} \vdash_t e : \mu, \varphi$, there exists a constraint set C and $\lambda_C^{\text{region}}$-term $\mathbf{e}$ such that $\{\,\}, C \vdash_c \mathbf{e} : \mu, \varphi$ and $|\mathbf{e}| = e$. Moreover, take the region annotation δ_c such that for all $\rho \in \mathit{frv}\,(C, \mu) \cup \varphi$ it holds that $\delta_c(\rho) = \mathsf{d}$. Then, we trivially have that $\delta_c \models C$, which concludes the proof.

B Proof of Second Substitution Lemma

Suppose

1. $P = \{\varrho_1 \dots, \varrho_n\}$ and $(P \cup \{\boldsymbol{\epsilon}\}) \cap (\mathit{fv}\,(TE) \cup \{\rho\}) = \emptyset$ and $\{\boldsymbol{\alpha}\} \cap \mathit{ftv}\,(TE) = \emptyset$ and $\sigma = \forall \boldsymbol{\alpha}.\hat{\sigma}$ and $\hat{\sigma} = \forall \boldsymbol{\epsilon}.\forall \varrho_1,\dots,\varrho_n.C^{\uparrow P} \Rightarrow \tau$
2. $TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$
3. $TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset$

then $TE, C_{\downarrow P} \vdash_c E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1\rangle_\rho] : \mu, \varphi$ where $A = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E_1\rangle_\rho$.

Proof. By induction on the derivation of $TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$. All cases are straightforward applications of the induction hypothesis, except for free occurrences of f in E_2. Consider the final derivation step for this case:

$$\frac{\begin{array}{c} TE(f) = (\underline{\sigma}, \rho) \quad \underline{\sigma} = \forall\, \boldsymbol{\alpha} .\forall\, \boldsymbol{\epsilon} .\forall\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n .\chi(C)^{\uparrow \underline{P}} \Rightarrow \chi(\tau) \\ (C', \tau') \prec \underline{\sigma}\ \mathit{via}\ S_s \quad S_r = \{\underline{\varrho}_1 \mapsto \rho'_1, \ldots, \underline{\varrho}_n \mapsto \rho'_n\} \quad S_s = (S_t, S_e, S_r) \\ C_0 = \left\{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho'_1, \ldots, \rho' \leq \rho'_n\right\} \quad C_{\downarrow P} \rhd C' \cup C_0 \end{array}}{TE + \{f \mapsto (\underline{\sigma}, \rho)\}, C_{\downarrow P} \vdash_c f\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' : (\tau', \rho'), \left\{\rho, \rho'\right\}}$$

where we have α-renamed the variables $P = \left\{\varrho_1, \ldots, \varrho_n\right\}$ into $\underline{P} = \left\{\underline{\varrho}_1, \ldots, \underline{\varrho}_n\right\}$ via the *renaming* region substitution $\chi = \{\varrho_1 \mapsto \underline{\varrho}_1, \ldots, \varrho_n \mapsto \underline{\varrho}_n\}$. In addition, we assume that

$$\underline{P} \cap (\mathit{frv}\,(TE, C) \cup \left\{\rho, \rho'\right\} \cup P \cup \mathit{Img}(S_s)) = \emptyset \tag{41}$$

where all $\underline{\varrho}_i$ are distinct. Also define χ^{-1} as the inverse substitution of χ. Observe that the type schemes $\underline{\sigma}$ and σ are equal up to α-renaming and similarly for $\underline{\hat{\sigma}}$ and $\hat{\sigma}$.

We must show that

$$\begin{array}{l} TE, C_{\downarrow P} \vdash_c \langle \Lambda\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n .\, \lambda\, x.\, \texttt{letrec}\ f = \chi(A)\ \texttt{in}\ \chi(E_1)\rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' \\ \qquad\qquad : (\tau', \rho'), \left\{\rho, \rho'\right\} \end{array}$$

That is, by rule *(Recvar-val)*, there must exist τ'' and C'' such that

- $TE, C'' \vdash_c \langle \lambda\, x.\, \texttt{letrec}\ f = \chi(A)\ \texttt{in}\ \chi(E_1)\rangle_{\rho'} : (\tau'', \rho'), \emptyset$;
- $\underline{P} \cap (\mathit{frv}\,(TE) \cup \left\{\rho, \rho'\right\})$ (immediate by construction);
- $\tau' = S_r(\tau'')$;
- $C_{\downarrow P} \rhd S_r(C'') \cup C_0$

A suitable choice for τ'' is $S'_s(\chi(\tau))$ where $S'_s = (S_t, S_e, I_r)$ and we take $\chi(C)$ for C''. Construction of $\underline{P}$ makes that $\mathit{Supp}(S_r) \cap \mathit{Img}(S'_s) = \emptyset$, hence we have $S_s = S_r S'_s$. Since $(C', \tau') \prec \underline{\sigma}\ \mathit{via}\ S_s$, the type τ'' satisfies the third requirement.

To show the last requirement, note that also via $(C', \tau') \prec \underline{\sigma}\ \mathit{via}\ S_s$ we have that $S_r(\chi(C)^{\uparrow \underline{P}}) = C' \subseteq C_{\downarrow P}$ and surely $S_r(\chi(C)^{\uparrow \underline{P}}) = S_r(\chi(C^{\uparrow P}))$. We also have by construction of χ that $S_r(\chi(C_{\downarrow P})) = C_{\downarrow P} \subseteq C_{\downarrow P}$. Putting this together, yields $S_r(\chi(C_{\downarrow P})) \cup S_r(\chi(C^{\uparrow P})) \subseteq C_{\downarrow P}$ or also that $S_r(\chi(C)) \subseteq C_{\downarrow P}$. The requirement is now proven since $C_0 \subseteq C_{\downarrow P}$ is given by one of the premises above.

It remains to show the first requirement which says that $TE, \chi(C) \vdash_c \langle \lambda\, x.\, \texttt{letrec}\ f = \chi(A)\ \texttt{in}\ \chi(E_1)\rangle_{\rho'} : (S'_s(\chi(\tau)), \rho'), \emptyset$. Using χ^{-1} as a region substitution and Lemma 5, this boils down to showing the following judgment:

$$TE, C \vdash_c \langle \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1\rangle_{\rho'} : (S'_s(\tau), \rho'), \emptyset \tag{42}$$

To see this, it is first shown that

$$TE, C \vdash_c \langle \lambda x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{43}$$

From item 3 we have trivially

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\underline{\hat{\sigma}}, \rho)\}, C \vdash_c \langle \lambda x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{44}$$

for any μ_1. Because of rule *(Abstr-val)*, we can take $\tau = \mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2$ and for some $\varphi'' \subseteq \varphi'$ have

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\underline{\hat{\sigma}}, \rho)\} + \{x \mapsto \mu_1\}, C \vdash_c E_1 : \mu_2, \varphi'' \tag{45}$$

where

$$C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2, \rho) \tag{46}$$

Applying Lemma 6 to (45) for $\underline{\hat{\sigma}} \prec \underline{\sigma}$ and observing that $TE + \{x \mapsto \mu_1\} + \{f \mapsto (\underline{\hat{\sigma}}, \rho)\} + \{x \mapsto \mu_1\} = TE + \{x \mapsto \mu_1\} + \{f \mapsto (\underline{\hat{\sigma}}, \rho)\}$ because $x \neq f$, yields

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\underline{\sigma}, \rho)\}, C \vdash_c E_1 : \mu_2, \varphi'' \tag{47}$$

Observe now that $C = C_{\downarrow \underline{P}}$ since $\mathit{frv}\,(C) \cap \underline{P} = \emptyset$. Hence, from rule *(Letrec-val)*, item 1 and equations (41), (44) and (47), we obtain

$$TE + \{x \mapsto \mu_1\}, C \vdash_c \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1 \rangle_\rho\ \texttt{in}\ E_1 : \mu_2, \varphi'' \tag{48}$$

Using rule *(Abstr-val)*, equations (46) and (48), we can derive equation (43). Now, applying Lemma 5 with S'_s on equation (43) yields

$$S'_s(TE), S'_s(C) \vdash_c S'_s(\langle \lambda x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho) : S'_s(\tau, \rho), S'_s(\emptyset) \tag{49}$$

The support of $S'_s = (S_t, S_e, I_r)$ is a subset of $\{\boldsymbol{\epsilon}, \boldsymbol{\alpha}\}$ since $(C', \tau') \prec \sigma$ *via* (S_t, S_e, S_r). By item 1, $\{\boldsymbol{\epsilon}, \boldsymbol{\alpha}\}$ is disjoint from $\mathit{fv}\,(TE)$. Therefore,

- $S'_s(TE) = TE$;
- $S'_s(\langle \lambda x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho) = \langle \lambda x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho$ due to I_r;
- $S'_s(C) = C$ due to I_r; and
- $S'_s(\tau, \rho) = (S'_s(\tau), \rho)$.

So, we have $TE, C \vdash_c \langle \lambda x.\, \texttt{letrec}\ f = A\ \texttt{in}\ E_1 \rangle_\rho : (S'_s(\tau), \rho), \emptyset$.

Note that given (46), we also have $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2, \rho')$ since $\{\rho \leq \rho', \rho' \leq \rho\} \subseteq C_0 \subseteq C$. Equation (42) can now be derived by replacing ρ for ρ' in rule *(Abstr-val)*. This concludes the proof.

C Proof of Type Preservation

Suppose $TE, C \vdash_c E^\mu : \mu, \varphi$ and $\delta \models C$ for $\delta_{\text{init}} \subseteq \delta$. If $\delta, E^\mu \rightsquigarrow \delta', E'$ then there exists a C' such that $TE, C' \vdash_c E' : \mu, \varphi'$ where $\varphi' \subseteq \varphi$; $\delta' \models C'$; $C \subseteq C'$ and $\delta \subseteq \delta'$.

Proof. By induction on the number of subsequent uses of *(Effect)* or *(Constr-elim)* at the end of E's type derivation. The base case for this induction is in turn proven by rule induction on the definition of $\rightsquigarrow$

If $TE, C \vdash_c E : \mu, \varphi$ derives from $TE, C \vdash_c E : \mu, \varphi \cup \{\epsilon\}$ by rule *(Effect)*, for some $\epsilon \notin \mathit{fev}\,(TE, \mu, \varphi)$, then induction yields that $TE, C \vdash_c E' : \mu, \varphi'$ where $\varphi' \subseteq \varphi \cup \{\epsilon\}$. If $\epsilon \notin \varphi'$ then $\varphi' \subseteq \varphi$ and the claim holds. Otherwise, use *(Effect)* to get $TE, C \vdash_c E' : \mu, \varphi' \setminus \{\epsilon\}$ where $\varphi' \setminus \{\epsilon\} \subseteq \varphi \cup \{\epsilon\} \setminus \{\epsilon\} = \varphi$, as required.

If $TE, C_{\downarrow\varrho^\bullet} \vdash_c E : \mu, \varphi$ derives from $TE, C \vdash_c E : \mu, \varphi$ by rule *(Constr-elim)*, then we also have $C \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. We are given that $\delta \models C_{\downarrow\varrho^\bullet}$ and $\delta_{\text{init}} \subseteq \delta$. This together with the fact that $C \setminus C_{\downarrow\varrho^\bullet} = C^{\uparrow\varrho^\bullet}$ by Lemma 1.1, makes that $\delta \models C^{\uparrow\varrho^\bullet}$ and therefore $\delta \models C$. Hence, applying the induction hypothesis gives that $TE, C' \vdash_c E' : \mu, \varphi'$ where $C \subseteq C'$, $\varphi' \subseteq \varphi$, $\delta' \models C'$ and $\delta \subseteq \delta'$. Since $C \subseteq C'$, we trivially have $C' \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. Applying rule *(Constr-elim)* proves the case.

If the last rule in the proof of $TE, C \vdash_c E : \mu, \varphi$ is not *(Effect)* or *(Constr-elim)*, we proceed by rule induction on $\rightsquigarrow$. When one of the accompanying facts $\varphi' \subseteq \varphi$, $\delta \subseteq \delta'$, $C \subseteq C'$ or $\delta \models C$ is immediate (for example, because the objects are identical or an immediate result by induction) we ignore them in the proof of the particular case. Also, recall that $\widetilde{C}$ builds the least transitive closure of C.

Case $\delta, c\ \mathtt{at}\ \varrho \rightsquigarrow \delta, \langle c \rangle_\varrho$ *if* $\delta(\varrho) = \mathtt{s}$**.**

By rule *(Const)*, we have that $C \vdash_{\mathit{wft}} (\mathtt{int}, \varrho)$. So, take $C'' = C \cup \{\varrho \leq \mathtt{s}\}$ and $C' = \widetilde{C''}$. Surely, $C \subseteq C'$ and since $\delta \models C$ and $\delta \models \{\varrho \leq \mathtt{s}\}$, we also have $\delta \models C'$ by Lemma 8. Of course we have $C' \vdash_{\mathit{wft}} (\mathtt{int}, \varrho)$ and hence by *(Const-val)* that $TE, C' \vdash_c \langle c \rangle_\varrho : (\mathtt{int}, \varrho), \emptyset$. Since $\emptyset \subseteq \varphi$, the case is proven.

Cases for the rules (2) and (3). Analogous.

Case $\delta, \Lambda^{\mathtt{s}}\, \varrho.\, Q \rightsquigarrow \delta, Q[\varrho \mapsto \varrho^\bullet]$**.**

We have by assumption that $TE, C_{\downarrow\varrho} \vdash_c \Lambda\, \varrho.\, Q : \mu, \varphi \setminus \{\varrho\}$ and $\delta \models C_{\downarrow\varrho}$. Hence, from rule *(Regabst-s)*, this implies that $TE, C \vdash_c Q : \mu, \varphi$ with $\varrho \notin \mathit{frv}\,(TE, \mu)$ and $C \rhd \{\varrho \leq \mathtt{s}\}$. Using Lemma 5, we obtain that $TE, C[\varrho \mapsto \varrho^\bullet] \vdash_c Q[\varrho \mapsto \varrho^\bullet] : \mu, \varphi[\varrho \mapsto \varrho^\bullet]$. The fact that $C \rhd \{\varrho \leq \mathtt{s}\}$ implies that $C[\varrho \mapsto \varrho^\bullet] \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. So from rule *(Constr-elim)*, we obtain $TE, C[\varrho \mapsto \varrho^\bullet]_{\downarrow\varrho^\bullet} \vdash_c Q[\varrho \mapsto \varrho^\bullet] : \mu, (\varphi[\varrho \mapsto \varrho^\bullet]) \setminus \{\varrho^\bullet\}$. The claim is proven by observing that $C[\varrho \mapsto \varrho^\bullet]_{\downarrow\varrho^\bullet} = C_{\downarrow\varrho}$ and $(\varphi[\varrho \mapsto \varrho^\bullet]) \setminus \{\varrho^\bullet\} = \varphi \setminus \{\varrho\}$.

Case $\delta, \mathtt{lift}\,[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \rightsquigarrow \delta, \langle c \rangle_{\varrho_2}$ *if* $\delta(\varrho_2) = \mathtt{s}$**.**

So, $TE, C \vdash_c \mathtt{lift}\,[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} : (\mathtt{int}, \varrho_2), \varphi \cup \{\varrho_1, \varrho_2\}$. Because of rule *(Lift)* and rule *(Const-val)*, we have that $\varphi = \emptyset$; $C \rhd \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \mathtt{s}\}$ and $C \vdash_{\mathit{wft}} (\mathtt{int}, \varrho_1)$. Take C' such that for $C'' = C \cup \{\varrho_1 \leq \mathtt{s}\}$, $C' = \widetilde{C''}$. Now, we obviously have $C' \vdash_{\mathit{wft}} (\mathtt{int}, \varrho_1)$ and also $\delta \models C'$ because of $\delta \models C$;

$\delta \models \{\varrho_2 \leq s\}$ and Lemma 8. Type rule *(Const-val)* makes that $TE, C \vdash_c \langle c \rangle_{\varrho_2} : (\mathtt{int}, \varrho_2), \emptyset$ and since $\emptyset \subseteq \{\varrho_1, \varrho_2\}$, the case is proven.

Case $\delta, \langle \lambda\, x.\, E \rangle_\varrho \,@\, Q \leadsto \delta, \mathtt{let}\ x = Q\ \mathtt{in}\ E$**.**

By assumption and Lemma 11 we have $TE, C \vdash_c \langle \lambda\, x.\, E \rangle_\varrho \,@\, Q : \mu_2, \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$ via *(App)*, where $TE, C \vdash_c \langle \lambda\, x.\, E \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \varrho), \emptyset$ and $TE, C \vdash_c Q : \mu_1, \varphi'$. So, by rule *(Abstr-val)* this makes $TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi''$ too, with $\varphi'' \subseteq \varphi'$. Using *(Let)* or the latter two judgments Q and E, we obtain $TE, C \vdash_c \mathtt{let}\ x = Q\ \mathtt{in}\ E : \mu_2, \varphi'' \cup \varphi'$, where $\varphi'' \cup \varphi' \subseteq \varphi \cup \varphi' \subseteq \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$.

Case $\delta, \mathtt{let}\ x = V\ \mathtt{in}\ E \leadsto \delta, E[x \mapsto V]$**.**

Immediate by rule *(Let)* and the first substitution lemma.

Case $\begin{array}{c} \delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E \rangle_\varrho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho_0 \leadsto \\ \delta, \langle \lambda\, x.\, E[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n] \rangle_{\varrho_0} \end{array}$.

So, rule *(Recvar-val)* applies:

$$\frac{\begin{array}{c} TE, C' \vdash_c \langle \lambda\, x.\, E \rangle_{\varrho_0} : (\tau, \varrho_0), \emptyset \quad \{\varrho_1, \ldots, \varrho_n\} \cap (frv\,(TE) \cup \{\varrho, \rho_0\}) = \emptyset \\ S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \quad \tau' = S_r(\tau) \\ C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \rho'_1, \ldots, \varrho_0 \leq \rho'_n\} \end{array}}{TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E \rangle_\varrho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho_0 : (\tau', \varrho_0), \{\varrho, \varrho_0\}}$$

By Lemma 5, we can apply S_r on the derivation for the pointer to the lambda abstraction. This yields: $TE, S_r(C') \vdash_c \langle \lambda\, x.\, S_r(E) \rangle_{\varrho_0} : (S_r(\tau), \varrho_0), \emptyset$. Since $\emptyset \subseteq \{\varrho, \varrho_0\}$; $S_r(\tau) = \tau'$; and $C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \rho'_1, \ldots, \varrho_0 \leq \rho'_n\}$, the case is proven by Lemma 4.

Case $\begin{array}{c} \delta, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_2 \leadsto \\ \delta, E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_1 \rangle_\rho] \end{array}$.

The left-hand side can only be typed by rule *(Letrec-val)*. This, however, gives exactly the requirements for the second substitution lemma, which on its turn proves the case.

Case $\dfrac{\delta, E \leadsto \delta', E'}{\delta, E \,@\, E'' \leadsto \delta', E' \,@\, E''}$.

The last step in the type derivation for $E \,@\, E''$ must be rule *(App)*, so we have $TE, C \vdash_c E \,@\, E'' : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. This means that $TE, C \vdash_c E : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $TE, C \vdash_c E'' : \mu_1, \varphi_2$ holds too. By induction, we have that $TE, C \vdash_c E' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi'_1$ for $\varphi'_1 \subseteq \varphi_1$, so the typing is immediately proven using rule *(App)*.

Cases for the rules (11) through (14).

Analogous to the previous case.

Case $\delta, \Lambda\, \varrho : C_0\, .\, E \leadsto \delta + \{\varrho \mapsto \mathsf{s}\}, \Lambda^{\mathsf{s}}\, \varrho.\, E$ *if* $\varrho \notin Dom(\delta)$ *and* $\delta + \{\varrho \mapsto \mathsf{s}\} \models C_0$**.**

First, observe that $\delta \subseteq \delta + \{\varrho \mapsto \mathsf{s}\}$. By assumption there exists a C such that $C_0 = C^{\uparrow\varrho}$ and $TE, C_{\downarrow\varrho} \vdash_c \Lambda\, \varrho : C_0.\, E : \mu, \varphi \setminus \{\varrho\}$ via *(Regabst)*, and $\delta \models C_{\downarrow\varrho}$. Hence, we have $TE, C \vdash_c E : \mu, \varphi$ where $\varrho \notin frv\,(\mu, TE)$. Take $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq s\}$. So, by Lemma 4, we have that $TE, C' \vdash_c E : \mu, \varphi$. As $C' \rhd \{\varrho \leq s\}$, we can apply rule *(Regabst-s)* to obtain $TE, C'_{\downarrow\varrho} \vdash_c \Lambda\, \varrho.\, E : \mu, \varphi \setminus \{\varrho\}$. Trivially we have $C \subseteq C'$ and hence also $C_{\downarrow\varrho} \subseteq C'_{\downarrow\varrho}$. Moreover, since we have that $\delta \models C_{\downarrow\varrho}$ and $\delta + \{\varrho \mapsto \mathsf{s}\} \models C_0 (= C^{\uparrow\varrho})$, applying Lemma 10 yields that $\delta + \{\varrho \mapsto \mathsf{s}\} \models C$. Also, we have $\delta + \{\varrho \mapsto \mathsf{s}\} \models \{\varrho \leq s\}$, so Lemma 8 makes that $\delta + \{\varrho \mapsto \mathsf{s}\} \models C'$. The case is now proven as we obviously have $\delta + \{\varrho \mapsto \mathsf{s}\} \models C'_{\downarrow\varrho}$ too.

Case $\delta, \Lambda\, \varrho : C \,.\, E \rightsquigarrow \delta + \{\varrho \mapsto \mathsf{d}\}, \Lambda^{\mathsf{d}}\, \varrho.\, E$ *if* $\varrho \notin Dom(\delta)$ *and* $\delta + \{\varrho \mapsto \mathsf{d}\} \models C$.

Analogous to the previous case.

Case $\delta, \mathtt{lift}\,[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \rightsquigarrow \delta, c \ \mathtt{at}\ \varrho_2$ *if* $\delta(\varrho_2) = \mathsf{d}$.

Analogous to the case for `lift` with $\delta(\varrho_2) = \mathsf{s}$.

Cases for the rules (17) through (24).

These rules are all fairly straightforward by the design of the type rules for residual expressions. Reductions (19), (20) and (23) rely on the design of the type rules *(Letrec-d)*, *(Regabst-d)* and *(Let-d)* respectively. Reduction rule (21) needs Lemma 12 and reduction (24) relies on the assumption about the type environment.

Cases for the rules (25) and (26) Trivial by the type rules *(Let)*, *(Let-d)*, *(Letrec)* and *(Letrec-d)*.

Case $\dfrac{\delta, E \rightsquigarrow \delta', E'}{\delta, \lambda\, x.\, E \ \mathtt{at}\ \varrho \rightsquigarrow \delta', \lambda\, x.\, E' \ \mathtt{at}\ \varrho}$ *if* $\delta(\varrho) = \mathsf{d}$.

We have to type the expressions with rule *(Abstr)*, which gives us that $TE, C \vdash_c \lambda\, x.\, E \ \mathtt{at}\ \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi$, where we have that $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho)$; $\varphi \subseteq \varphi'$; and $TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi$. The induction hypothesis, the premise of (27) and the latter judgment give us that $TE + \{x \mapsto \mu_1\}, C \vdash_c E' : \mu_2, \varphi''$ with $\varphi'' \subseteq \varphi$. So we can apply *(Abstr)* to prove the case.

Cases for the rules (28) through (31).

All similarly simple appeals to the induction hypothesis.

Case $\delta, \mathtt{lift}\,[\rho_1, \rho_2]\, (\Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \Lambda^{\mathsf{d}}\, \varrho.\, (\mathtt{lift}\,[\rho_1, \rho_2]\, \mathrm{Q}^{\mathsf{s}})$ *if* $\varrho \notin \{\rho_1, \rho_2\}$.

To type the left-hand side, we first use *(Lift)* and then *(Regabst-d)*, giving us the following derivation:

$$\dfrac{\dfrac{\begin{array}{c} TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi \\ \varrho \notin frv\,(TE) \cup \{\rho_1\} \quad C \rhd \{d \leq \varrho\} \end{array}}{TE, C_{\downarrow\varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi \setminus \{\varrho\}} \qquad C_{\downarrow\varrho} \rhd \{\rho_1 \leq \rho_2\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\, (\Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}}) : (\mathtt{int}, \rho_2), (\varphi \setminus \{\varrho\}) \cup \{\rho_1, \rho_2\}}$$

The case is now proven by re-ordering: first, we have $TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi$. Now, obviously we have $\{\rho_1 \leq \rho_2\} \subseteq C$ and by Lemma 3 therefore also that $C \rhd \{\rho_1 \leq \rho_2\}$. By rule *(Lift)*, this makes $TE, C \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\, \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}$. Using the fact that $\varrho \notin \{\rho_1, \rho_2\}$, we have that $\varrho \notin frv\,(TE) \cup \{\rho_2\}$. Since $C \rhd \{d \leq \varrho\}$, we can apply *(Regabst-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.\, (\mathtt{lift}\,[\rho_1, \rho_2]\, \mathrm{Q}^{\mathsf{s}}) : (\mathtt{int}, \rho_2), (\varphi \cup \{\rho_1, \rho_2\}) \setminus \{\varrho\}$. And obviously we have $(\varphi \cup \{\rho_1, \rho_2\}) \setminus \{\varrho\} \subseteq (\varphi \setminus \{\varrho\}) \cup \{\rho_1, \rho_2\}$, concluding the case.

Case $\delta, \mathtt{let}\ x = (\Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} \rightsquigarrow \delta, \Lambda^{\mathsf{d}}\, \varrho.\, (\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E^{\mu})$ *if* $\varrho \notin frv\,(\mu)$.

Construct the following (part of a) type derivation for the left-hand side, using *(Let)* and *(Regabst-d)*:

$$\dfrac{\dfrac{\begin{array}{c} TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : \mu', \varphi' \\ \varrho \notin \mathit{frv}\,(\mu', TE) \quad C \rhd \{\mathtt{d} \leq \varrho\} \end{array}}{TE, C_{\downarrow\varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}} : \mu', \varphi' \setminus \{\varrho\}} \qquad TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \vdash_c E^{\mu} : \mu, \varphi}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{let}\ x = (\Lambda^{\mathsf{d}}\, \varrho.\, \mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} : \mu, \varphi \cup (\varphi' \setminus \{\varrho\})}$$

First, observe by Lemma 1.1, 3 and 4, that given $TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \vdash_c E : \mu, \varphi$, we also have $TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \cup C^{\uparrow\varrho} \vdash_c E : \mu, \varphi$. This, together with $TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : \mu', \varphi'$ and rule *(Let)* makes $TE, C \vdash_c \mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'$. Since we have that $\varrho \notin \mathit{frv}\,(TE)$; $C \rhd \{\mathtt{d} \leq \varrho\}$ and also that $\varrho \notin \mathit{frv}\,(\mu)$, we can apply rule *(Regabst-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E) : \mu, (\varphi \cup \varphi') \setminus \{\varrho\}$. The claim now follows since $(\varphi \cup \varphi') \setminus \{\varrho\} = (\varphi \setminus \{\varrho\}) \cup (\varphi' \setminus \{\varrho\}) \subseteq \varphi \cup (\varphi' \setminus \{\varrho\})$.

Cases for the rules (34) through (40).
Proven analogous to the two previous cases: the derivation tree of the original expression is reordered to obtain a proof for the result expression.

D Proof of Progress

Suppose for a $\lambda 2^{\text{region}}$-term E^{μ} that $TE, C \vdash_c E^{\mu} : \mu, \varphi$. Additionally, assume that $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$; $\mathit{Clean}(\varphi)$; $C \vdash_{\mathit{wft}} TE$; and for all $(\sigma, \rho') \in \mathit{Ran}(TE)$, it holds that $\delta(\rho') = \mathsf{d}$. Then we either have that

1. there exists an E' and δ' such that $\delta, E^{\mu} \rightsquigarrow \delta', E'$; or
2. E is an answer, *i.e.* of the form Q

Proof. By rule induction on $TE, C \vdash_c E^{\mu} : \mu, \varphi$. As in Proposition 1, we can assume that the derivation does not end with *(Effect)* or *(Constr-elim)*. Otherwise, we can use exactly the same arguments to apply the induction hypothesis, using Proposition 1 to conclude. The other judgments are proven by a case distinction.

Case $TE, C \vdash_c x : \mu, \emptyset$. A variable x is always residual code, *i.e.* an answer.
Case $TE, C \vdash_c c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$. Note that $\mathit{Clean}(\{\rho\})$, so we have that $\rho \in \mathsf{RegionVar}$. Either $\delta(\rho) = \mathsf{s}$ in which case reduction rule (1) applies or, if $\delta(\varrho) = \mathsf{d}$, we can use rule (17).
Case $TE, C \vdash_c \lambda x.\, E'\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\} \cup \varphi$. Whenever $\delta(\rho) = \mathsf{s}$ we can reduce with rule (2), otherwise, we have that $\delta(\rho) = \mathsf{d}$. From type rule *(Abstr)* we also have that

$$TE + \{x \mapsto \mu_1\}, C \vdash_c E' : \mu_2, \varphi \tag{50}$$

where $\varphi \subseteq \varphi'$ and $C \vdash_{\mathit{wft}} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho)$. Since $\delta(\varrho) = \mathsf{d}$, we have for $\mu_1 = (\tau_1, \varrho_1)$ and $\mu_2 = (\tau_2, \varrho_2)$ that $\delta(\varrho_1) = \mathsf{d}$ and $\delta(\varrho_2) = \mathsf{d}$ due to well-formedness. Moreover, it now also holds that $C \vdash_{\mathit{wft}} TE + \{x \mapsto \mu_1\}$ with for all $(\sigma, \rho') \in \mathit{Ran}(TE)$ that $\delta(\rho') = \mathsf{d}$.

Applying the induction hypothesis, we either have $\delta, E' \rightsquigarrow \delta', E''$, in which case we can reduce with the context rule (27). Alternatively, we have that E' is answer of the form Q. However, we can apply the canonical forms Lemma 15 to see that Q must be of the form R. Hence, the expression is reducible with rule (18).

Case $TE, C_{\downarrow P} \vdash_c \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, E_1\ \texttt{at}\ \rho\ \texttt{in}\ E_2 : \mu, \varphi \cup \varphi'$. Either we have that $\delta(\rho) = \mathsf{s}$ and we apply reduction rule (3). Alternatively, we have $\delta(\rho) = \mathsf{d}$. By an analogous argument as in the previous case (using well-formedness of C) we can either reduce the expression with the context rule (28) or with rule (26).

Case $TE, C \vdash_c E_1 \mathbin{@} E_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. So, by rule *(App)*, we have

$$TE, C \vdash_c E_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \tag{51}$$

and

$$TE, C \vdash_c E_2 : \mu_1, \varphi_2 \tag{52}$$

Now, apply the induction hypothesis on equation (51). Either $\delta, E_1 \rightsquigarrow \delta', E_1'$ which means we can use reduction rule (10) to reduce $E_1 \mathbin{@} E_2$ to $E_1' \mathbin{@} E_2$. Alternatively $E_1 = Q_1$, so then use the induction hypothesis on equation (52). If we have that $\delta, E_2 \rightsquigarrow \delta', E_2'$, we can apply reduction rule (11) on $Q_1 \mathbin{@} E_2$. If $E_2 = Q_2$, we make a case distinction for Q_1.

So, suppose first $\delta(\rho) = \mathsf{d}$, then because of equation (51) and the canonical forms lemma, it must be that Q_1 is residual code R_1. However, by Lemma 7, it also holds that $C \vdash_{\mathit{wft}} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)$, and since $\delta \models C$, we have that for $\mu_1 = (\tau_1, \rho_1)$ it must be that $\delta(\rho_1) = \mathsf{d}$. So, by the canonical forms lemma and judgment (52), it must be that Q_2 is also residual code R_2. But then we have that $E_1 \mathbin{@} E_2$ is actually $R_1 \mathbin{@} R_2$, allowing us to apply rule (22).

Alternatively, suppose $\delta(\rho) = \mathsf{s}$. Then by judgment (51) and the canonical forms lemma, it must be that Q_1 is a static answer, *i.e.* of the form Q_1^{s}. If Q_1^{s} is value, then again by the canonical forms lemma, Q_1^{s} is of the form $\langle \lambda x.\, E' \rangle_\varrho$ and reduction rule (6) applies. If Q_1^{s} is not a value, we can shift the static expression context with the rules (34), (37) or (40).

Case $TE, C \vdash_c \texttt{let}\ x = E_1\ \texttt{in}\ E_2 : \mu, \varphi \cup \varphi'$. By type rule *(Let)* we have that $TE, C \vdash_c E_1 : \mu', \varphi'$, so we can immediately appeal to the induction hypothesis. Suppose first $\delta, E_1 \rightsquigarrow \delta', E_1'$. Then we can use reduction rule (13) and item 1 applies. If E_1 is of the form Q_1, then we have the following case distinction: if Q_1 is a value V, then we can reduce with rule (7). If Q_1 is a static answer Q_1^{s}, but not a value, then either one of the rules (33), (36) or (39) applies. If Q_1 is residual code R, then we use reduction (25).

Case $TE, C \vdash_c f\, [\rho_1', \ldots, \rho_n']\ \texttt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}$. Immediately reducible with rule (24).

Case $TE, C_{\downarrow\varrho} \vdash_c \Lambda\, \varrho : C^{\uparrow\varrho} .\, E : \mu, \varphi \setminus \{\varrho\}$. By Lemma 9, we can satisfy the constraint of rule (15).

Case $TE, C \vdash_c \texttt{lift}\,[\rho_1, \rho_2]\, E : (\texttt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}$. By type rule *(Lift)* we have that $TE, C \vdash_c E : (\texttt{int}, \rho_1), \varphi$ and $C \rhd \{\rho_1 \leq \rho_2\}$. Since $\mathit{Clean}(\varphi \cup$

$\{\rho_1, \rho_2\}$), we have that $\{\rho_1, \rho_2\} \subseteq \mathsf{RegionVar}$. By application of the induction hypothesis we either have that $\delta, E \rightsquigarrow \delta', E'$, in which case we can reduce with rule (12), or we have that E is an answer Q. First, consider the case that $\delta(\rho_1) = \mathsf{d}$. Then obviously also $\delta(\rho_2) = \mathsf{d}$, implying by the canonical forms lemma that Q is residual code R. In this case, we can reduce with rule (21).

However, if $\delta(\rho_1) = \mathsf{s}$, we have by the canonical forms lemma that Q is a static answer Q^{s}. If Q^{s} is not a value, then the context rules (32), (35) or (38) are applicable. On the other hand, if Q^{s} is a value V, then by the canonical forms lemma it must be that $V = \langle c \rangle_{\rho_1}$. If now $\delta(\rho_2) = \mathsf{s}$, we reduce with rule (5), otherwise, if $\delta(\rho_2) = \mathsf{d}$, we reduce with rule (16).

Case $TE, C \vdash_c \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset$**.** Constant pointers are answers.

Case $TE, C \vdash_c \langle \lambda\, x.\, E \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset$**.** Function pointers are answers.

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_2 : \mu, \varphi$**.** We reduce this term with reduction rule (9).

Case $TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda\, x.\, E \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}$**.** Since *Clean*($\{\rho, \rho'\}$) we have that $\{\rho, \rho'\} \subseteq \mathsf{RegionVar}$. Hence, we reduce the expression with rule (8).

Case $TE, C \vdash_c \Lambda^{\mathsf{s}}\, \varrho.\, E : \mu, \varphi \setminus \{\varrho\}$**.** Since *Clean*($\varphi \setminus \{\varrho\}$), we obviously have *Clean*(φ). Take a region annotation $\delta' = \delta + \{\varrho \mapsto \mathsf{s}\}$ where $\varrho \notin (\mathit{Dom}(\delta) \cup \mathit{frv}\,(C))$ and $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq s\}$. It is given that $\delta \models C$, hence also $\delta' \models C$. Obviously we have $\delta' \models \{\varrho \leq s\}$ too, so by Lemma 8 this implies that $\delta' \models C'$. Since $C = C'_{\downarrow\varrho}$, type rule *(Regabst-s)* gives us that $TE, C' \vdash_c E : \mu, \varphi$ with $\varrho \notin \mathit{frv}\,(TE, \mu)$. Now, all conditions are satisfied to apply the induction hypothesis. Either $\delta', E \rightsquigarrow \delta'', E'$ and we can reduce the left-hand side with rule (14) or if E is an answer, we can use rule (4).

Case $TE, C_{\downarrow\varrho} \vdash_c \Lambda^{\mathsf{d}}\, \varrho.\, E : \mu, \varphi \setminus \{\varrho\}$**.** With almost identical arguments (using the constraint $d \leq \varrho$ instead of $\varrho \leq s$) as in the previous case we can apply the induction hypothesis. If E is a static answer Q^{s}, then the whole term is a static answer. If E is residual code R, then we reduce the expression with (20). Finally, if E can be further reduced, we can use transition rule (29) to further reduce the expressions.

Case $TE, C \vdash_c \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ E' : \mu, \varphi \cup \varphi'$**.** So, by type rule *(Let-d)*, we have $TE, C \vdash_c R : (\tau', \rho'), \varphi'$ and $TE + \{x \mapsto \mu'\}, C \vdash_c E' : \mu, \varphi$. By lemma 12 we have that $\delta(\rho') = \mathsf{d}$ and hence, we can apply the induction hypothesis on E'. So, either we have $\delta, E' \rightsquigarrow \delta', E''$ and reduction rule (30) applies. Whenever E' is an answer Q, we consider two cases. If Q is residual code R', we can reduce with rule (23). Otherwise, if Q is a static answer Q^{s}, then the whole expression is a static answer as well.

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\, \lambda x.\, R\ \mathtt{at}\ \rho\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'$**.** Analogous to the previous case using rules (19) and (31).

Case $TE, C \vdash_c c\ \underline{\mathtt{at}}\ \varrho : (\mathtt{int}, \varrho), \{\varrho\}$**.** Residual terms are immediately answers.

Remaining cases for residual terms. All immediately, since residual terms are answers.

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*, pages 88–97, Trento, Italy, July 1999. IEEE Computer Society Press.
3. Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 58:299–392, 2001.
4. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.
5. Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 1–10, Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.
6. Mikhail A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In Schmidt [48], pages 59–65.
7. Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.
8. Charles Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université de Paris VI, Paris, France, June 1989.
9. Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990. ACM Press.
10. Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [48], pages 66–77.
11. Charles Consel. A tour of Schism. In Schmidt [48], pages 134–154.
12. Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [32], pages 496–519.
13. 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.
14. Charles Consel, Pierre Jouvelot, and Peter Ørbæk. Separate polyvariant binding-time reconstruction. Technical Report CRI-A/261, Ecole des Mines, Paris, October 1994.
15. Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 206–214, Las Vegas, NV, USA, June 1997. ACM Press.
16. 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*, number 983 in Lecture Notes in Computer Science, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.

17. Marc Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In *Proceedings of the 2nd Workshop on Static Analysis*, volume 81–82 of *Bigre Journal*, pages 322–330, Rennes, France, 1992. IRISA.
18. David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.
19. Kevin Glynn, Peter Stuckey, Martin Sulzmann, and Harald Søndergaard. Boolean constraints for binding-time analysis. In *Programs as Data Objects II*, number 2053 in Lecture Notes in Computer Science, pages 39–62, Aarhus, Denmark, May 2001. Springer-Verlag.
20. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [21].
21. Robert Harper. A note on: "A simplified account of polymorphic references". *Information Processing Letters*, 57(1):15–16, January 1996. See also [20].
22. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.
23. Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator*. PhD thesis, Chalmers University of Technology, April 2001.
24. Rogardt Heldal and John Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science*, 248:99–145, 2000.
25. Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. In *PSI-01: Andrei Ershov Fourth International Conference, Perspectives of System Informatics*, number 2244 in Lecture Notes in Computer Science, pages 191–204, Novosibirsk, Russia, July 2001. Springer-Verlag.
26. Simon Helsen. An equational theory for a region calculus. Technical Report 172, Institut für Informatik, University of Freiburg, Germany, April 2002. Available from ftp://ftp.informatik.uni-freiburg.de/documents/reports/report172/report00172.ps.gz.
27. Simon Helsen and Peter Thiemann. Fragmental specialization. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation (SAIG'00)*, number 1927 in Lecture Notes in Computer Science, pages 51–71, Montreal, Canada, September 2000. Springer-Verlag.
28. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Montreal, Canada, September 2000. Elsevier Science.
29. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [32], pages 448–472.
30. 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.
31. 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.
32. John Hughes, editor. *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, MA, 1991. Springer-Verlag.
33. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

34. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.
35. John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
36. 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.
37. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report Technical Report CS-95-178, Brandeis University, Waltham, Massachusetts, 1995. Extended version of [36] from ftp://ftp.brics.dk/pub/danvy/Papers/lawall-danvy-lfp94-extended.ps.gz.
38. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, number 1281 in Lecture Notes in Computer Science, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.
39. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
40. Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
41. Torben Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, number 351,352 in Lecture Notes in Computer Science, pages II, 298–312, Barcelona, Spain, March 1989. Springer-Verlag.
42. James H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT Press, December 1968.
43. Flemming Nielson and Hanne Riis Nielson. Automatic binding-time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.
44. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.
45. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.
46. Bernhard Rytz and Marc Gengler. A polyvariant binding time analysis. In Charles Consel, editor, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '92*, pages 21–28, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.
47. S.B.Lassen. Relational reasoning about contexts. In Andrew Gordon and Andrew Pitts, editors, *HOOTS '98 Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 91–136. Cambridge University Press, 1998.
48. David Schmidt, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, Copenhagen, Denmark, June 1993. ACM Press.
49. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.
50. Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.

51. Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, number 1140 in Lecture Notes in Computer Science, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.
52. 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.
53. Peter Thiemann. A unified framework for binding-time analysis. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 742–756, Lille, France, April 1997. Springer-Verlag.
54. Peter Thiemann. Enforcing safety properties using type specialization. In David Sands, editor, *Proc. 10th European Symposium on Programming*, Lecture Notes in Computer Science, Genova, Italy, April 2001. Springer-Verlag.
55. 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.
56. Peter Thiemann and Michael Sperber. Polyvariant expansion and compiler generators. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, number 1181 in Lecture Notes in Computer Science, pages 285–296, Novosibirsk, Russia, June 1996. Springer-Verlag.
57. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
58. Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
59. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olensen, and Peter Sestoft. *Programming with Regions in the ML Kit, Version 4.0.0*, September 2001. Available from http://www.it.edu/research/mlkit/dist/mlkit-4.0.0.pdf.
60. 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, OR, January 1994. ACM Press.
61. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
62. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.