Polymorphic Specialization for ML

SIMON HELSEN[1]
University of Waterloo, Canada
and
PETER THIEMANN
Universität Freiburg, Germany

We present a framework for offline partial evaluation for call-by-value functional programming languages with an ML-style typing discipline. This includes a binding-time analysis which is

(1) polymorphic with respect to binding times;
(2) allows the use of polymorphic recursion with respect to binding times;
(3) is applicable to a polymorphically typed term; and
(4) is proven correct with respect to a novel small-step specialization semantics.

The main innovation is to build the analysis on top of the region calculus of Tofte and Talpin, thus leveraging the tools and techniques developed for it. Our approach factorizes the binding-time analysis into region inference and a subsequent constraint analysis. The key insight underlying our framework is to consider binding times as properties of regions.

Specialization is specified as a small-step semantics, building on previous work on syntactic type soundness results for the region calculus. 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.

Categories and Subject Descriptors: D.1.1 [**PROGRAMMING TECHNIQUES**]: Applicative (Functional) Programming; D.3.1 [**PROGRAMMING LANGUAGES**]: Formal Definitions and Theory; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—*Operational semantics*; I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming—*Program Transformation*; *Program Synthesis*

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Binding-time Analysis, Program Specialization, Regions

[1]Work performed while at the Universität Freiburg, Germany.

Authors' addresses: S. Helsen, University of Waterloo, Department of Electrical and Computer Engineering, 200 University Av. West, Waterloo, ON, N2L 3G1, Canada; email: shelsen@computer.org; P. Thiemann, Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079, D-79110 Freiburg im Breisgau, Germany; email: thiemann@informatik.uni-freiburg.de.

1. INTRODUCTION

1.1 Background

Partial evaluation [Jones et al. 1993; Consel and Danvy 1993] is a well-explored program specialization technique, which factorizes the evaluation of a program in two stages. The *static* or *specialization* stage performs symbolic evaluation of the program. Starting from some known part of the input and the constants in the programs, the specialization executes all operations depending only on known data. The result of that stage is a specialized (or *residual*) program. The *dynamic* or *runtime* stage executes the specialized program given the rest of the input. Running the specialized program in this way is often faster than running the original program on the whole input, thereby amortizing the cost of partial evaluation if the specialized program is executed several times.

Offline partial evaluation is a variant of program specialization where the specialization is preceded by a *binding-time analysis* (BTA) [Henglein 1991; Thiemann 1997b; Nielson and Nielson 1988]. A BTA is a program analysis that determines which constructs in a program can be executed at specialization time. It yields an annotation of each construct with a binding time indicating in which stage it is to be executed. A binding-time annotated expression is often called a *two-level term* [Nielson and Nielson 1992] and a binding-time analysis is part of the static semantics for offline partial evaluation[2]. A BTA can be expressed in terms of an annotated type system, with binding-time annotated *two-level types*.

A *monovariant* BTA assigns each construct a fixed binding time. Many existing partial evaluators rely on a monovariant BTA [Nielson and Nielson 1988; Consel 1990; Henglein 1991; Bondorf and Jørgensen 1993; Birkedal and Welinder 1994; Thiemann 2000] and many applications have been studied in the context of these systems. However, such a BTA is unnecessarily conservative and often requires program transformations (binding-time improvements) like duplicating function definitions [Jones et al. 1993]. In contrast, a *polyvariant* analysis computes for each function a binding-time annotation that depends on the context of each particular call of a function [Consel 1993a]. Several applications benefit from binding-time polyvariance, for instance modular program specialization [Dussart et al. 1997], separate compilation via specialization [Helsen and Thiemann 2000a; Heldal and Hughes 2000], and enforcing security properties by specialization [Thiemann 2001]. 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 the finiteness of the analysis.

An appealing 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 [Henglein and Mossin 1994; Dussart et al. 1995] yield such an analysis using a simply typed lambda calculus as a base language. Their analysis is binding-time polymorphic and admits polymorphic recursion on the level of binding times, that is, the binding times of recursive calls of a function may be different at each call. Dependencies between as yet unknown

[2]The term "static semantics" is not to be confused with "static stage". In fact, specialization (static evaluation) is the *dynamic semantics* of offline partial evaluation.

binding times are expressed as constraints in qualified type schemes. The higher-order version of the polymorphic analysis [Dussart et al. 1995] includes a notion of annotation subtyping where the subtyping relation only affects the binding-time annotations but the structure of the underlying simple type remains unaffected.

1.2 Our Work

From this starting point, we further develop the idea of polymorphic binding-time analysis and polymorphic specialization. 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 on types and binding times and polymorphic recursion on the level of binding times.

The key idea of our approach is to start from a rich core calculus that already includes information about data flow and dependency required to support binding-time analysis. We have chosen the region calculus [Tofte and Talpin 1997] as the basis because it exhibits these properties and because there is a well-established typed translation from the ML-core calculus to the region calculus. The regions in this calculus stand for disjoint sets of memory locations and the calculus contains explicit constructs for introducing regions and for attaching allocations and data accesses to regions. The rules of the calculus make sure that the management of regions is consistent with the data flow and data dependencies.

These observations have led us to consider a binding time as a property of a region. This perhaps unexpected use of the region calculus offers a number of advantages. First, it separates the concerns of the flow analysis that is implicit in the region analysis from the well-formedness (dependency) requirements on two-level types [Thiemann 1997b; 2002]. The latter are captured in a *constraint analysis* subsequent to region inference. Second, an implementation of the BTA can take advantage of existing polynomial-time region reconstruction algorithms [Tofte and Birkedal 1998; Birkedal and Tofte 2001]. Third, *any* type-correct term in the region calculus can be converted into a soundly annotated term by the constraint analysis. This enables the BTA to take advantage of further progress in region inference algorithms and also of manually region-annotated terms. In particular, the more precise the region analysis is, the better is the binding-time annotation resulting from its constraint analysis. Fourth, the region calculus is built on top of a type-polymorphic language and incorporates region polymorphism and polymorphic recursion for regions, both of which are desirable for a binding-time analysis.

Extracting results from the constraint analysis amounts to simplifying the generated constraints (see [Dussart et al. 1995] for some techniques). Since the actual binding times only become known at specialization time, a polymorphic analysis cannot resolve all constraints at analysis time. Rather, the binding time of each region is computed during specialization when the region is created. The analysis simplifies the constraints to the point where this computation amounts to taking the least upper bound of a set of known binding times.

Specialization is described with a Plotkin-style structural operational semantics [Plotkin 1981]. Soundness of the constraint analysis and correctness of the specializer are proven by operational methods. Building on previous work [Calcagno 2001; Helsen and Thiemann 2000b; Calcagno et al. 2002], we use syntactic type soundness techniques [Wright and Felleisen 1994; Harper 1994] to prove the sound-

ness 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. This proof relies on an equational theory for the region calculus established in a companion paper [Helsen 2004].

In summary, the key contributions of this work are

—the BTA is specified for a polymorphic base language;

—the BTA is factored into a region analysis and a subsequent constraint analysis, thus leveraging algorithms for region inference [Tofte and Birkedal 1998; Birkedal and Tofte 2001] as well as existing constraint simplification techniques [Dussart et al. 1995];

—specialization is formalized using a small-step operational semantics derived from a semantics for the region calculus [Helsen and Thiemann 2000b];

—the BTA is proven correct with respect to the specialization semantics using standard techniques for proving type soundness; and

—the specialization steps are proven correct with respect to an adequate equational theory for the region calculus (see the companion paper [Helsen 2004]).

1.3 Related Work

Dussart, Henglein and Mossin [Dussart et al. 1995; Henglein and Mossin 1994] specify a polymorphic binding-time analysis with polymorphic recursion in terms of an annotated type system. Their framework builds on top of a simply-typed lambda calculus. It includes a notion of annotation subtyping that only affects binding times, but not the underlying structure of the type. Their work presents constraint simplification rules and gives a polynomial-time fix-point algorithm for constraint inference. The first-order version [Henglein and Mossin 1994] includes a soundness proof based on a denotational model which does not deal correctly with name generation. In comparison, our base language is polymorphically typed, we provide a syntactic type soundness proof for specialization, and we prove the soundness of each specialization step. However, our system does not include annotation subtyping. Annotation subtyping implies the (automatic) insertion of higher-order *binding-time coercions* into the program. Such a coercion can transform a static, specialization-time value to a runtime value (code) and is implemented using two-level eta expansion [Danvy et al. 1995; 1996]. Uncontrolled, automatic use of this feature can lead to code duplication if, for example, a static function is coerced to code in several different places. Hence, we prefer to have the user of the BTA selectively enforce the effect of annotation subtyping by eta-expanding terms at suitable program points [Thiemann 1997b]. Of course, our system includes first-order binding-time coercions (also known as lifts) which are indispensable.

Heldal and Hughes [Heldal and Hughes 2001; Heldal 2001] extend the work of Dussart, Henglein, and Mossin by considering a polymorphic base language and by representing the coercions required by annotation subtyping in their type language. While our work does not address annotation subtyping and coercions, as discussed in the previous paragraph, it provides a comprehensive correctness proof of the BTA and specialization.

Glynn et al. [2001] generalize a polymorphic binding-time analysis to polymorphic programs, by encoding binding-time subtyping constraints as boolean formulae within the HM(X)-framework [Odersky et al. 1999]. Constraint simplification relies on existing fast boolean constraint solvers. They do not specify a specialization semantics corresponding to their analysis and do not consider the formal correctness.

Thiemann considers another variant of the region calculus as the basis of a binding-time analysis for a simply-typed call-by-value lambda calculus with references [Thiemann 1997a]. In that work, the emphasis is 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.

Abadi et al. [1999] show that their Dependency Core Calculus can provide a semantic basis for a monomorphic binding-time analysis in a simply-typed calculus. Continuing that work, Banerjee et al. [1999] define a polymorphic lambda calculus that provides a denotational proof for the soundness of the region calculus. 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 [Calcagno 2001; Helsen and Thiemann 2000b; Calcagno et al. 2002]. These papers discuss and relate other syntactic approaches to the type soundness result. The present work adds the handling of constraints and binding times to the type system and to the operational semantics.

Mogensen [1989] 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 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 et al. [1994]. 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 hampers their goal of modularity.

Hornof and Noyé [2000] define a binding-time analysis for imperative languages that is flow, context, and return sensitive. Their context sensitivity corresponds to a 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 [Consel 1989; 1993a; Bulyonkov 1993; Gengler and Rytz 1992; Rytz and Gengler 1992; Consel 1993b]. Some of them do not formally relate the analysis with the specialization phase and some only deal with first-order functional languages.

1.4 Notation

Let A and B be sets. We write $A \hookrightarrow B$ for the set of partial functions from A to B. The *domain* of a function $f \in A \hookrightarrow B$ is $Dom\,(f) = \{x \in A \mid \exists y(y \in B \wedge f(x) = y)\}$ and its *range* is $Ran\,(f) = \{y \in B \mid \exists x(x \in A \wedge f(x) = y)\}$. We write $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ for a finite function f with $Dom\,(f) = \{x_1, \ldots, x_n\}$ and $\{\,\}$ for the empty function. A partial function $f \in A \hookrightarrow B$ is *total* whenever $Dom\,(f) = A$ and we write $A \to B$ for the set of total functions from A to B.

The extension $f + \{x \mapsto a\}$ of $f \in A \hookrightarrow B$ is a partial function where $(f + \{x \mapsto a\})(x') = f(x')$ if $x' \in Dom\,(f) \setminus \{x\}$ and $(f + \{x \mapsto a\})(x) = a$. The application $(f + \{x \mapsto a\})(x')$ is undefined whenever $x' \notin Dom\,(f) \cup \{x\}$. The *co-restriction* $f - X$ of a partial function f is defined by $(f - X)(x) = f(x)$ if $x \in Dom\,(f) \setminus X$ and undefined otherwise. A finite sequence of symbols a_i is written $\vec{a}$ and the set of the elements in a sequence $\vec{a}$ is denoted by $\{\vec{a}\}$. The symbol $=$ denotes equality of terms modulo α-conversion for its bound variables.

1.5 Overview

Section 2 contains a brief introduction to the region calculus. In Section 3, we describe a constraint analysis on top of a type derivation in the region calculus: after defining the extended calculus, we introduce semantic objects (Section 3.2) which are required to describe the constraint rules (Section 3.3). We then formulate some basic properties (Section 3.4), and define the notion of a constraint completion (Section 3.5). Section 4 introduces the specialization semantics: we define a two-level extension of the region calculus (Section 4.1) and give a small-step operational semantics (Section 4.2). Next, we extend the calculus to accommodate a small-step rewrite semantics (Section 4.3). We then illustrate specialization with some examples (Section 4.4) and add the missing type rules for the two-level extension of the region calculus (Section 4.5). The correctness proof has two parts: Section 5 handles the soundness of the constraint analysis. Two substitution lemmas are needed to prove type preservation. Then, we formulate progress and present a soundness result for the constraint analysis. Soundness of specialization is proven in Section 6. First, we recall a sound equational theory for the region calculus from a companion paper [Helsen 2004] (Section 6.1). Then, we use the equational theory to prove soundness of each specialization step (Section 6.2). Section 7 contains a detailed assessment of our work and discusses avenues for future work. Finally, in Section 8, we conclude.

2. THE REGION CALCULUS

A region calculus is an annotated polymorphically typed lambda calculus that makes memory management explicit using regions [Tofte and Talpin 1994; Talpin and Jouvelot 1992; Gifford and Lucassen 1986; Jouvelot and Gifford 1991]. A prominent example is λ^{region}, the region calculus by Tofte and Talpin [Tofte and Talpin 1997]. The main use of λ^{region} is the tracking of memory allocation and deallocation. It has been successfully applied to implement a runtime system for Standard ML [Milner et al. 1997] that does not require garbage collection [Tofte et al. 2001]. It also forms the basis for the offline partial evaluation framework developed in the present work.

A region is a set of potentially infinitely many memory locations. Two regions are either disjoint or equal. In λ^{region}, regions are allocated and deallocated as a whole in a stack-like manner. Region inference [Birkedal and Tofte 2001; Tofte and Birkedal 1998; 2000] translates an ML program into a λ^{region}-term as defined by the following grammar:

$$\begin{aligned}\lambda^{region}\text{-Term} \ni e \ ::= \ & x \mid c\ \mathtt{at}\ \varrho \mid \lambda x.\, e\ \mathtt{at}\ \varrho \mid e \mathbin{@} e \mid \mathtt{lift}\,[\varrho_1, \varrho_2]\, e \mid \\ & \mathtt{new}\,\varrho.\, e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid f\ [\varrho_1, \ldots, \varrho_n]\ \mathtt{at}\ \varrho \mid \\ & \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, e\ \mathtt{at}\ \varrho\ \mathtt{in}\ e\end{aligned}$$

The main syntactic novelty is the use of region variables, ϱ, that range over regions of memory. The annotation $\mathtt{at}\,\varrho$ on constants, lambda expressions, and `letrec` definitions of recursive functions indicates that the respective object is to be allocated in the region bound to ϱ. Evaluation of a region abstraction, $\mathtt{new}\,\varrho.\, e$, first allocates a new region and binds it to ϱ. Then, evaluation determines the value of the expression e, deallocates the region, and finally returns the value. The `letrec` expression defines a recursive function f that also abstracts over the region variables $\varrho_1, \ldots, \varrho_n$. The expression $f\ [\varrho_1, \ldots, \varrho_n]\ \mathtt{at}\ \varrho$ instantiates such a region abstraction and stores the resulting function object in the region bound to ϱ.

Free and bound expression variables are defined as usual. Analogously, a region abstraction binds a region variable within its lexical scope. An expression without free expression variables is called a *program*. Programs may have free region variables, for instance, an initial region in which the program stores its result.

Compared to λ^{region}, our calculus contains two additional constructs. They affect neither the formal properties of the underlying system nor the results of region reconstruction. The `lift` expression models a canonical primitive operation. It is the identity function copying its argument (of base type) from the region bound to ϱ_1 to the region bound to ϱ_2. We will extend its meaning in Section 4 where we discuss the specialization semantics. The presence of the `let` expression is required by our equational theory. It does *not* introduce polymorphism. Operationally, $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ behaves like $(\lambda x.\, e_2\ \mathtt{at}\ \varrho) \mathbin{@} e_1$ but without allocating a closure. Intuitively, the `let` just puts the value of e_1 on the stack.

Let us look at an example of a λ^{region}-Term.

Example 2.1.

$$\begin{aligned}e_0 \equiv\ & \mathtt{letrec}\ f = \Lambda\, \varrho_1, \varrho_2 . \lambda x.\, (\mathtt{lift}\,[\varrho_1, \varrho_2]\, x)\ \mathtt{at}\ \varrho_0\ \mathtt{in} \\ & (f\ [\varrho_3, \varrho_4]\ \mathtt{at}\ \varrho_5) \mathbin{@} (42\ \mathtt{at}\ \varrho_3)\end{aligned}$$

The function f is bound to the identity function on a base type. It is parameterized over regions ϱ_1 and ϱ_2. The `lift` in f copies the value of x from region ϱ_1 to region ϱ_2. In the body of the `letrec`, the function f is applied to the regions bound to ϱ_3 and ϱ_4 respectively. The resulting lambda is then stored in the region bound to ϱ_5. Finally, the lambda is applied to the constant 42 which lives in the region bound to ϱ_3. The result of this term is 42 stored in region ϱ_4.

3. CONSTRAINT ANALYSIS

The static semantics of specialization extends Tofte and Talpin's type system for λ^{region} by a constraint analysis. Since the formal properties of the λ^{region}-type

$$\varrho \in \mathsf{RegionVar} \qquad x, f \in \mathsf{Var} \qquad c \in \mathsf{Constant} \qquad \mathsf{p} \in \mathsf{Placeholder} \supseteq \mathsf{RegionVar}$$

$$C \in \mathsf{ConstraintSet} = \mathcal{P}(\{\mathsf{p}_1 \leq \mathsf{p}_2 \mid \mathsf{p}_1, \mathsf{p}_2 \in \mathsf{Placeholder}\})$$

$$\begin{aligned} \lambda_C^{region}\text{-}\mathsf{Term} \ni e ::= {} & x \mid c\ \mathtt{at}\ \mathsf{p} \mid \lambda x.\, e\ \mathtt{at}\ \mathsf{p} \mid e\,@\,e \mid \mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,e \mid \\ & \mathtt{new}\,\varrho : C\,.e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid f\ [\mathsf{p}_1, \ldots, \mathsf{p}_n]\ \mathtt{at}\ \mathsf{p} \mid \\ & \mathtt{letrec}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda x.\, e\ \mathtt{at}\ \mathsf{p}\ \mathtt{in}\ e \end{aligned}$$

Fig. 1. Syntax of λ_C^{region}

system are well established [Tofte and Talpin 1997; Calcagno 2001; Helsen and Thiemann 2000b; Calcagno et al. 2002], the present work concentrates on the constraint aspects.

The analysis starts from a type derivation for a λ^{region}-term and generates constraints according to our extended rules. As explained above, the result of the analysis is a term annotated with (simplified) sets of constraints between the formal binding times of the regions required for evaluating the term. At specialization time, when the actual binding times are known, the binding time of each region is computed from these constraint sets.

3.1 Syntax of λ_C^{region}

The result of the constraint analysis is a term in the constraint-annotated calculus, λ_C^{region}, generated by the grammar in Figure 1. Except for the use of region placeholders and a constraint set in region abstractions, λ_C^{region}-Terms are identical to λ^{region}-Terms[3]. The sets RegionVar, Var, and Constant are infinite denumerable disjoint sets of region variables, expression variables, and constant symbols. A region *placeholder* p is a meta variable which serves as a notational convenience to reuse type and reduction rules. In λ_C^{region}, a placeholder can only be a region variable. The definition of specialization later on extends the set of placeholders to include other objects. Finally, a constraint set C contains ordered pairs of placeholders, written $\mathsf{p}_1 \leq \mathsf{p}_2$. They impose *binding-time constraints* as we shall see below. When specializing the region abstraction $\mathtt{new}\,\varrho : C\,.e$, the constraint set C must be solved to determine the binding time of the region bound to ϱ. We will make this mechanism more precise when we discuss specialization.

Define the *erasure* as a total function $|\cdot| \in \lambda_C^{region}\text{-}\mathsf{Term} \to \lambda^{region}\text{-}\mathsf{Term}$. It transforms a λ_C^{region}-term into a λ^{region}-term by stripping the constraint sets C from all region abstractions.

3.2 Semantic Objects of λ_C^{region}

Most semantic objects required in the constraint analysis are taken from the type system of λ^{region} [Tofte and Talpin 1997]. In this section, we briefly summarize them and provide definitions to handle constraints.

[3]We will omit the annotations λ^{region} and λ_C^{region} from categories if no confusion is possible.

3.2.1 *Types.* The following grammar generates the type language of λ_C^{region}, where EffectVar and TypeVar are disjoint denumerable sets:

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

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

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

An *effect* φ is a finite set of placeholders, p, 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 to simplify type reconstruction for λ^{region} [Tofte and Birkedal 2000]. They play no active role during the constraint analysis. Each effect variable identifies a group of functions that are connected by shared applications. The use of effect variables amounts to a simple control-flow analysis. The latent effect of an arrow effect contains the regions affected by the body of the function. Hence, these regions appear in the effect of each application point of the function. A *type with place*, μ, is a pair of a type, τ, and a placeholder, p. The placeholder 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 is qualified by a constraint set which transports constraints from the function definition point to the function application point.

As an example, consider the qualified type scheme of the `letrec`-bound variable f in the expression e_0 of Example 2.1.

$$\sigma = \forall\, \epsilon\,.\forall\, \varrho_1, \varrho_2\,.\{\varrho_1 \leq \varrho_2, \varrho_0 \leq \varrho_1, \varrho_0 \leq \varrho_2\} \Rightarrow (\mathtt{int}, \varrho_1) \xrightarrow{\epsilon.\{\varrho_1,\varrho_2\}} (\mathtt{int}, \varrho_2) \quad (1)$$

Disregarding the constraint set, it means that the function takes an integer in a arbitrary region ϱ_1, returns an integer in region ϱ_2, and may affect regions ϱ_1 and ϱ_2. Later on, in Section 3.3, we discuss the constraint part of this type scheme.

A type environment, $TE \in \mathsf{TypeEnv}$, is a partial function that maps lambda-bound variables to pairs of the form (τ, p) and `letrec`-bound variables to pairs of the form (σ, p).

The *erasure* function $|\cdot|$ also applies to type schemes and type environments by stripping off the qualifying constraint sets. Hence, $|\sigma| \in \lambda^{region}$-TypeScheme and $|TE| \in \lambda^{region}$-TypeEnv.

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

3.2.2 *Substitutions.* A substitution, $S_s = (S_t, S_e, S_r)$, is a triple of functions: a type substitution $S_t \in \mathsf{TypeVar} \to \mathsf{Type}$, an effect substitution $S_e \in \mathsf{EffectVar} \to \mathsf{EffectVar} \times \mathsf{Effect}$, and a region substitution $S_r \in \mathsf{RegionVar} \to \mathsf{Placeholder}$. The substitution S_s distributes operations component-wise over its three substitutions. The definition of S_s on types, types with place, placeholders and effects extends the usual definition [Tofte and Talpin 1997] as follows (regarding $\bullet$ as a special region

variable for the moment):

$$S_s(\varphi) = \{S_s(\mathsf{p}) \mid \mathsf{p} \in \varphi\} \cup \{\eta \mid \epsilon \in \varphi \wedge S_e(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\} \quad S_s(\bullet) = \bullet$$

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

$$S_s(\alpha) = S_t(\alpha) \quad S_s(\mathtt{int}) = \mathtt{int} \quad S_s(\varrho) = S_r(\varrho) \quad S_s(\tau, \mathsf{p}) = (S_s(\tau), S_s(\mathsf{p}))$$

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

$$S_s(C) = \{S_s(\mathsf{p}_1) \leq S_s(\mathsf{p}_2) \mid \mathsf{p}_1 \leq \mathsf{p}_2 \in C\}$$

The substitutions I_t and I_r are the identities on type and region variables, respectively, and 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, $\mathit{Supp}\,(S_t)$, is the set $\{\alpha \mid S_t(\alpha) \neq \alpha\}$. By abuse of notation, we sometimes write $\{\alpha_1 \mapsto S_t(\alpha_1), \ldots, \alpha_n \mapsto S_t(\alpha_n)\}$ for the substitution S_t with $\mathit{Supp}\,(S_t) = \{\alpha_1, \ldots, \alpha_n\}$. The sets $\mathit{Supp}\,(S_r)$, $\mathit{Supp}\,(S_e)$ and $\mathit{Supp}\,(S_s)$ are defined analogously. Similarly, we define the *image* of a type substitution S_t by $\mathit{Img}\,(S_t) = \{S_t(\alpha) \mid \alpha \in \mathit{Supp}\,(S_t)\}$ and analogously for S_e, S_r and S_s.

Since region placeholders may occur in expression syntax, we define the substitution $S_s(e) = S_r(e)$, where the region substitution propagates homomorphically in the expression, respecting locally bound region variables as usual.

Substitutions extend to type environments, and expressions in the usual capture avoiding way.

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

As an example of instantiation, let us take an instance of the type scheme σ from (1). An appropriate substitution could be $S_s = (I_t, \{\epsilon \mapsto \epsilon'.\{\}\}, \{\varrho_1 \mapsto \varrho_3, \varrho_2 \mapsto \varrho_4\})$. With this substitution, we would have that

$$(\{\varrho_3 \leq \varrho_4, \varrho_0 \leq \varrho_3, \varrho_0 \leq \varrho_4\}, (\mathtt{int}, \varrho_3) \xrightarrow{\epsilon'.\{\varrho_3, \varrho_4\}} (\mathtt{int}, \varrho_4)) \prec \sigma \textit{ via } S_s$$

3.2.3 *Constraints.* Finally, we establish a few definitions and properties on constraint sets.

Definition 3.1. *Suppose a constraint set C.*

(1) *C is* transitive closed *iff $\mathsf{p}_1 \leq \mathsf{p}_2 \in C$ and $\mathsf{p}_2 \leq \mathsf{p}_3 \in C$ implies $\mathsf{p}_1 \leq \mathsf{p}_3 \in C$.*

(2) *A constraint set C' is* consistent *with respect to C (written $C \rhd C'$) 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. We always have $\mathit{fps}\,(C) = \mathit{fps}\,(\widetilde{C})$. Constraint sets have upper and lower projections with respect to a set of placeholders:

$$C^{\uparrow\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}} = \{\mathsf{p}_0 \leq \mathsf{p}'_0 \in C \mid \exists\, i \in \{1, \ldots, n\} : \mathsf{p}_0 = \mathsf{p}_i \vee \mathsf{p}'_0 = \mathsf{p}_i\}$$

$$C_{\downarrow\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}} = C \setminus C^{\uparrow\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}} \qquad C_{\downarrow\mathsf{p}} = C_{\downarrow\{\mathsf{p}\}} \qquad C^{\uparrow\mathsf{p}} = C^{\uparrow\{\mathsf{p}\}}$$

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

A region annotation δ is a *solution* for a constraint set C (written $\delta \models C$) if $\mathit{fps}\,(C) \subseteq \mathit{Dom}\,(\delta)$ and for all $\mathsf{p}_1 \leq \mathsf{p}_2 \in C$ it holds that $\delta(\mathsf{p}_1) \leq \delta(\mathsf{p}_2)$. A constraint set C is *solvable* if it has a solution.

As an example, consider the constraint set $C = \{\varrho_3 \leq \varrho_4, \varrho_0 \leq \varrho_3\}$. The least transitive closure of this constraint set is $\widetilde{C} = \{\varrho_3 \leq \varrho_4, \varrho_0 \leq \varrho_3, \varrho_0 \leq \varrho_4\}$. Its upper projection on ϱ_0 is as follows: $(\widetilde{C})^{\uparrow \varrho_0} = \{\varrho_0 \leq \varrho_3, \varrho_0 \leq \varrho_4\}$. Finally, the region annotation $\delta = \{\varrho_0 \mapsto \mathsf{s}, \varrho_3 \mapsto \mathsf{d}, \varrho_4 \mapsto \mathsf{d}\}$ is a solution for $\widetilde{C}$, *i.e.* $\delta \models \widetilde{C}$.

3.3 Static Semantics of λ_C^{region}

3.3.1 *Well-formedness.* No specializer allows static values to be embedded in generated code. This property is enforced by the well-formedness of the binding-time annotations and is specified by adding constraints to the underlying type system. A type scheme, $\sigma \in \mathsf{TypeScheme}$, and placeholder, $\mathsf{p} \in \mathsf{Placeholder}$, are well-formed with respect to constraint set C if the judgment $C \vdash_{wft} (\sigma, \mathsf{p})$ can be derived using the following rules:

$$\textit{(Wft-Base)}\quad \frac{C \rhd \emptyset}{C \vdash_{wft} (\mathtt{int}, \mathsf{p})} \qquad \textit{(Wft-Tyvar)}\quad \frac{C \rhd \emptyset}{C \vdash_{wft} (\alpha, \mathsf{p})}$$

$$\textit{(Wft-Fun)}\quad \frac{\begin{array}{c} C \rhd \{\mathsf{p} \leq \mathsf{p}_1, \mathsf{p} \leq \mathsf{p}_2\} \\ C \vdash_{wft} \mu_1 \quad C \vdash_{wft} \mu_2 \quad \mu_1 = (\tau_1, \mathsf{p}_1) \quad \mu_2 = (\tau_2, \mathsf{p}_2) \end{array}}{C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \mathsf{p})}$$

$$\textit{(Wft-Scheme)}\quad \frac{C \vdash_{wft} (\tau, \mathsf{p})}{C_{\downarrow\{\vec{\varrho}\}} \vdash_{wft} (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\forall\, \vec{\varrho}\,.C^{\uparrow\{\vec{\varrho}\}} \Rightarrow \tau, \mathsf{p})}$$

The rules *(Wft-Base)* and *(Wft-Fun)* are standard [Dussart et al. 1995; Thiemann 1997b].

For base types and type variables, rules *(Wft-Base)* and *(Wft-Tyvar)* respectively, require a transitive closed C, but no additional constraints. An arrow type is well-formed by rule *(Wft-Fun)*. 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-Scheme)* generalizes the well-formedness property to a pair of a qualified type scheme and a placeholder. It does this by collecting all the constraints for the un-quantified type τ in the set C. The qualified constraint set only contains the constraints of C involving quantified regions, *i.e.*, $C^{\uparrow\{\vec{\varrho}\}}$. The constraint set of the context only needs to have the remaining constraints, *i.e.*, $C_{\downarrow\{\vec{\varrho}\}}$.

The notion of a *a well-formed type environment* TE with respect to a constraint set C (written $C \vdash_{wft} TE$) is defined by the following two inference rules.

$$\frac{C \rhd \emptyset}{C \vdash_{wft} \{\,\}} \qquad \frac{C \vdash_{wft} (\sigma, \mathsf{p}) \quad C \vdash_{wft} TE}{C \vdash_{wft} TE + \{f \mapsto (\sigma, \mathsf{p})\}}$$

That is, $C \vdash_{wft} TE$ implies that C is transitive closed and, for each $f \in Dom\,(TE)$, it holds that $C \vdash_{wft} TE(f)$.

Let σ be the type scheme from equation (1). We will show that (σ, ϱ_0) is well-formed under constraint set $C_0 = \{\varrho_1 \leq \varrho_2, \varrho_0 \leq \varrho_1, \varrho_0 \leq \varrho_2\}$. By *(Wft-Base)* it holds that $C_0 \vdash_{wft} (\mathtt{int}, \varrho_1)$ and $C_0 \vdash_{wft} (\mathtt{int}, \varrho_2)$. It is also true that $C_0 \rhd \{\varrho_0 \leq \varrho_1, \varrho_0 \leq \varrho_2\}$. So, by *(Wft-Fun)*, we obtain

$$C_0 \vdash_{wft} ((\mathtt{int}, \varrho_1) \xrightarrow{\epsilon.\{\varrho_1,\varrho_2\}} (\mathtt{int}, \varrho_2), \varrho_0)$$

Hence, we conclude with rule *(Wft-Scheme)* that $\{\} \vdash_{wft} (\sigma, \varrho_0)$.

3.3.2 *Typing Rules.* The constraint analysis is based on a region derivation, as produced by a region inference algorithm. Hence, all λ_C^{region}-Terms e have a fixed region type with place, μ, which we indicate by superscripting, e^μ, when needed.

A constrained type judgment has the form $TE, C \vdash_c e : \mu, \varphi$, which is read "given a type environment TE and a constraint set C, the λ_C^{region}-term e has type μ and effect φ". The rules in Figure 2 specify the constraint analysis.

Rules *(Var)*, *(Const)*, *(App)*, *(Let)*, and *(Effect)* are just the standard rules (see [Tofte and Talpin 1997] for a good explanation) extended by threading of the constraint set. The additional premises in *(Var)* and *(Const)* ensure the transitive closedness of the constraint set. Rule *(Lift)* deals with copying a value from placeholder p_1 to p_2. Accordingly, the binding time of p_1 must be less than that of p_2. This allows for first-order binding-time coercions as we shall see below. Rule *(Reglam)* introduces a region variable ϱ. The body, e, of the region abstraction is type checked with the constraint set C. The region abstraction passes only 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 mention ϱ, *i.e.*, $C^{\uparrow\varrho}$. When specialization reaches a region abstraction, all regions in $C^{\uparrow\varrho}$ —except ϱ— will have a known binding time. 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 body of the `letrec` and the context can be type checked with the constraints in C *not* mentioning the region variables in P, *i.e.*, $C_{\downarrow P}$.

Rule *(Regapp)* 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 *(Lam)* 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 guarantees sound specialization. The latent effect φ' is a superset of the effect φ of the body. This potential enlargement is required when two or more different functions flow together and therefore need the same functional type. In the standard type system for λ^{region}, the effect of a lambda expression is $\{\mathsf{p}\}$, indicating that a closure is allocated in the region p. However, for program specialization, the effect $\{\mathsf{p}\}$ is not sufficient because specialization continues "under the lambda" if the lambda turns out to be dynamic (where we stress that the actual binding time is only known at specialization time).

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

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

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

(Letrec)
$$\frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\mathsf{p}\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}. \forall\, \varrho_1, \ldots, \varrho_n\, .C^{\uparrow P} \Rightarrow \tau \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, C \vdash_c \lambda x.\, e_1\ \mathtt{at}\ \mathsf{p} : (\tau, \mathsf{p}), \varphi' \\ \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \mathsf{p})\}, C_{\downarrow P} \vdash_c e_2 : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, e_1\ \mathtt{at}\ \mathsf{p}\ \mathtt{in}\ e_2 : \mu, \varphi \cup \varphi'}$$

(App)
$$\frac{TE, C \vdash_c e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \mathsf{p}), \varphi_1 \quad TE, C \vdash_c e_2 : \mu_1, \varphi_2}{TE, C \vdash_c e_1\ @\ e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathsf{p}\}}$$

(Let)
$$\frac{TE, C \vdash_c e_1 : \mu', \varphi' \quad TE + \{x \mapsto \mu'\}, C \vdash_c e_2 : \mu, \varphi}{TE, C \vdash_c \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mu, \varphi \cup \varphi'}$$

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

(Reglam)
$$\frac{TE, C \vdash_c e : \mu, \varphi \quad \varrho \notin fps\,(TE, \mu)}{TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}\ \varrho : C^{\uparrow \varrho}\, .e : \mu, \varphi \setminus \{\varrho\}}$$

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

(Lift)
$$\frac{TE, C \vdash_c e : (\mathtt{int}, \mathsf{p}_1), \varphi \quad C \rhd \{\mathsf{p}_1 \leq \mathsf{p}_2\}}{TE, C \vdash_c \mathtt{lift}\, [\mathsf{p}_1, \mathsf{p}_2]\, e : (\mathtt{int}, \mathsf{p}_2), \varphi \cup \{\mathsf{p}_1, \mathsf{p}_2\}}$$

Fig. 2. Constraint Analysis

The rule *(Lam)* from Figure 2 addresses this problem by including the effect of the lambda's body in the effect of the lambda. This modification is required to ensure type soundness. It is conservative with respect to the original rule and may result in delayed region deallocation in some pathological cases. For example, consider a dynamic lambda which is never applied and which contains (dead) static code in its body. In λ^{region}, it would be safe to deallocate the objects in the dynamic lambda immediately, because the lambda itself *is* dead and the body is never executed. However, since a specializer reduces the static code of the body of a dynamic lambda – even if the lambda is dead – it is only safe to deallocate the static objects in the body *after* the dynamic lambda is specialized. Since every lambda may be used with both binding times during specialization, the BTA conservatively

assumes that every lambda is reduced at its definition site, *i.e.*, it is used at least once with binding time dynamic. This particular situation is illustrated with an example in Section 4.4.3.

If a dynamic lambda is not dead, then the body's effect will be discharged by rule *(App)*, even in the standard region type system, making the practical impact of the change rather small. Late deallocation is not critical in an program specialization context, early deallocation is still desirable to keep the constraint sets small in rule *(Reglam)*.

The attentive reader might wonder if the change in rule *(Lam)* invalidates known region reconstruction algorithms. Fortunately, only minimal changes that do not impact the complexity are required in the algorithms [Tofte and Birkedal 1998; Birkedal and Tofte 2001][4].

Now, we come back to the example expression e_0 of Example 2.1. In Section 3.2, we already stated that the type scheme of the `letrec`-bound function f is

$$\forall\, \epsilon\, . \forall\, \varrho_1, \varrho_2\, . \{\varrho_1 \leq \varrho_2, \varrho_0 \leq \varrho_1, \varrho_0 \leq \varrho_2\} \Rightarrow (\mathtt{int}, \varrho_1) \xrightarrow{\epsilon . \{\varrho_1, \varrho_2\}} (\mathtt{int}, \varrho_2)$$

The function type maps a value of type with place $(\mathtt{int}, \varrho_1)$ to type with place $(\mathtt{int}, \varrho_2)$. Because of rule *(Lift)*, the effect of the body must be $\{\varrho_1, \varrho_2\}$ and the constraint set C in the type judgment must include $\varrho_1 \leq \varrho_2$. Then, by type rule *(Lam)*, we move the effect set $\{\varrho_1, \varrho_2\}$ into the arrow effect of the function type and make sure that $\varrho_0 \leq \varrho_1$ and $\varrho_0 \leq \varrho_2$ are now also included in C because of the well-formedness requirement. Finally, we can use type rule *(Letrec)* to obtain the qualified type scheme where we observe that $C^{\uparrow\{\varrho_1,\varrho_2\}} = C$ and $C_{\downarrow\{\varrho_1,\varrho_2\}} = \emptyset$.

In the body of the `letrec`, we see that the expression $f\ [\varrho_3, \varrho_4]\ \mathtt{at}\ \varrho_5$ has type with place $((\mathtt{int}, \varrho_3) \xrightarrow{\epsilon . \{\varrho_3, \varrho_4\}} (\mathtt{int}, \varrho_4), \varrho_5)$ because of type rule *(Regapp)*. The resulting constraint set in the type judgment will be $C_0 = \{\varrho_0 \leq \varrho_5, \varrho_5 \leq \varrho_0, \varrho_5 \leq \varrho_3, \varrho_5 \leq \varrho_4, \varrho_0 \leq \varrho_3, \varrho_0 \leq \varrho_4, \varrho_3 \leq \varrho_4\}$. Using type rule *(App)*, it is now easy to see that the type with place of the expression e_0 is $(\mathtt{int}, \varrho_4)$. The total effect of e_0 is $\{\varrho_0, \varrho_3, \varrho_4, \varrho_5\}$. In other words, we have that

$$\{\,\}, C_0 \vdash_c e_0 : (\mathtt{int}, \varrho_4), \{\varrho_0, \varrho_3, \varrho_4, \varrho_5\}$$

is derivable.

3.4 Properties of the Constraint Analysis

The constraint analysis of Figure 2 induces a type system for λ^{region} by *erasing* the pars dealing with constraints. That is, for every judgment $TE, C \vdash_c e : \mu, \varphi$ in the type system for λ^{region}_C, there is a type judgment $|TE| \vdash_t |e| : \mu, \varphi$ for λ^{region}. The rules are identical to those of Figure 2, except that we drop the well-formedness judgments for the constraint sets.

Constraints and the constraint analysis have many useful properties, usually proven by straightforward induction arguments.

LEMMA 3.1. *Let* $P = \{p_1, \ldots, p_n\}$.

(1) $C^{\uparrow P} \cup C_{\downarrow P} = C$ *and* $C^{\uparrow P} \cap C_{\downarrow P} = \emptyset$.

[4]Private communication with Lars Birkedal, December 2001

(2) If C is transitive closed, then so is $C_{\downarrow P}$.

LEMMA 3.2. *Whenever $P_1, P_2 \subseteq \mathsf{Placeholder}$, we have $(C_{\downarrow P_1})_{\downarrow P_2} = (C_{\downarrow P_2})_{\downarrow P_1}$*

LEMMA 3.3. *Suppose $C \vdash_{wft} \mu$, then C is transitive closed.*

LEMMA 3.4. *Suppose $TE, C \vdash_c e : \mu, \varphi$, then C is transitive closed.*

PROOF. By induction on a type derivation, using Lemma 3.1.2 and Lemma 3.3. □

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

LEMMA 3.5. *Suppose $TE, C \vdash_c e : \mu, \varphi$ and $C' \rhd C$ then $TE, C' \vdash_c e : \mu, \varphi$.*

LEMMA 3.6. *Suppose $C \vdash_{wft} (\tau, \mathsf{p})$, then for any S_s we also have that $S_s(C) \vdash_{wft} (S_s(\tau), S_s(\mathsf{p}))$.*

A constraint type judgment exhibits a type substitution lemma:

LEMMA 3.7. *If $TE, C \vdash_c e : \mu, \varphi$ then, for all substitutions S_s, we have that $S_s(TE), S_s(C) \vdash_c S_s(e) : S_s(\mu), S_s(\varphi)$.*

PROOF. By induction on the derivation of $TE, C \vdash_c e : \mu, \varphi$. Standard extension of Lemma 5.4 in [Helsen and Thiemann 2000b] using Lemma 3.6. □

It also obeys a weakening lemma:

LEMMA 3.8. *If $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C \vdash_c e : \mu, \varphi$, and $\sigma \prec \sigma'$, then $TE + \{f \mapsto (\sigma', \mathsf{p})\}, C \vdash_c e : \mu, \varphi$.*

PROOF. A straightforward induction on $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C \vdash_c e : \mu, \varphi$. Simple extension of Lemma 5.5 in [Helsen and Thiemann 2000b]. □

LEMMA 3.9. *If $TE, C \vdash_c e : \mu, \varphi$ and $C \vdash_{wft} TE$, then $C \vdash_{wft} \mu$.*

PROOF. By induction on the type derivation. Except for *(Regapp)* and *(Letrec)*, all cases are straightforward applications of the induction hypothesis.

Case (Regapp). Because $C \vdash_{wft} TE$, we have that $C \vdash_{wft} (\sigma, \mathsf{p})$. By the well-formedness rule *(Wft-Scheme)*, this implies that $C = C_{\downarrow\{\vec{\varrho}\}}$ and by its premise that $C_{\downarrow\{\vec{\varrho}\}} \vdash_{wft} (\tau, \mathsf{p})$. By α-renaming, we can assume without loss of generality that $\mathsf{p} \notin \{\vec{\varrho}\}$. Assume that $S_s = (S_t, S_e, \{\varrho_1 \mapsto \mathsf{p}'_1, \ldots, \varrho_n \mapsto \mathsf{p}'_n\})$, then we have that $S_s(C_{\downarrow\{\vec{\varrho}\}}) \vdash_{wft} (\tau', \mathsf{p})$ making use of Lemma 3.6. But since we also have that $C \rhd \{\mathsf{p} \leq \mathsf{p}', \mathsf{p}' \leq \mathsf{p}\}$, we can conclude that $C \vdash_{wft} (\tau', \mathsf{p}')$.

Case (Letrec). In the premise of rule *(Letrec)*, we have $TE + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, C \vdash_c \lambda x.\, e_1 \ \mathtt{at}\ \mathsf{p} : (\tau, \mathsf{p}), \varphi'$. Hence, by type rule *(Lam)*, we can immediately conclude that $C \vdash_{wft} (\tau, \mathsf{p})$. Using well-formedness rules *(Wft-Scheme)*, this implies that $C_{\downarrow P} \vdash_{wft} (\sigma, \mathsf{p})$. Now, we can apply the induction hypothesis on $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C_{\downarrow P} \vdash_c e_2 : \mu, \varphi$ to conclude that $C_{\downarrow P} \vdash_{wft} \mu$.

□

Closing a constraint set C does not affect its solution.

LEMMA 3.10. *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 sacrificing solvability:

LEMMA 3.11. *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 3.12. *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 3.1.1, we have that $C = C^{\uparrow\varrho} \cup C_{\downarrow\varrho}$, so for any $\mathsf{p}_1 \leq \mathsf{p}_2 \in C$, we either have $\mathsf{p}_1 \leq \mathsf{p}_2 \in C^{\uparrow\varrho}$, which trivially implies $\delta \models C^{\uparrow\varrho}$ by assumption, or $\mathsf{p}_1 \leq \mathsf{p}_2 \in C_{\downarrow\varrho}$. Since $\delta \models C_{\downarrow\varrho}$ and $\varrho \notin fps(C_{\downarrow\varrho})$, by definition, we obviously have that $\delta' \models C_{\downarrow\varrho}$. □

3.5 Constraint Completions

An offline partial evaluator requires a *constraint completion*[5] to specialize successfully. Starting from a region derivation for $\{\} \vdash_t e' : \mu, \varphi$, the constraint completion provides a λ_C^{region}-Term e whose erasure is identical to the λ^{region}-Term e' and which is typeable in the constraint analysis of Figure 2 with a solvable constraint set. The following definition makes this precise:

DEFINITION 3.2. *Suppose $\{\} \vdash_t e' : (\tau, \varrho), \varphi$ for λ^{region}-Term e' with $(\tau, \varrho) \in$ TypeWPlace and $\varphi \in$ Effect. A λ_C^{region}-Term e with initial constraint set $C \in$ ConstraintSet and region annotation δ_c is a* constraint completion *of e' iff $|e^{(\tau,\varrho)}| = e'$, the judgment $\{\}, C \vdash_c e^{(\tau,\varrho)} : (\tau, \varrho), \varphi$ holds, $\delta_c \models C$, $\delta_c(\varrho) = \mathsf{d}$ and for all $\mathsf{p} \in \varphi$ it holds that $\delta_c(\mathsf{p}) = \mathsf{d}$.*

A constraint completion exists for every well-typed λ^{region}-program. That is, the constraint analysis conservatively extends region inference.

THEOREM 3.1. *Suppose $\{\} \vdash_t e' : (\tau, \varrho), \varphi$. Then there exists a constraint completion $e^{(\tau,\varrho)}$ with region annotation δ_c.*

PROOF. The construction is a follows:

—take the set $C = \{\varrho \leq \varrho' \mid$ *for all region variables ϱ and ϱ' in the program*$\}$

—translate every $\texttt{new}\,\varrho.\,e$ in the program into $\texttt{new}\,\varrho : C^{\uparrow\varrho}.e'$, where e' is the translation of e.

—from this, it is easy to see that the program is constraint-typeable with the rules from Figure 2 because we can enlarge the constraint sets in typing judgments arbitrarily with Lemma 3.5.

[5] In partial evaluation literature, this is usually called a *binding-time completion* [Henglein 1991].

—finally, we take δ_c such that it maps all region variables in $\varphi \cup \mathit{fps}\,(C, \mu)$ to d. Then, it obviously holds that $\delta_c \models C$ as well.

□

The expression e_0 of Example 2.1 is already a constraint completion with initial constraint set $C_0 = \{\varrho_0 \leq \varrho_5, \varrho_5 \leq \varrho_0, \varrho_5 \leq \varrho_3, \varrho_5 \leq \varrho_4, \varrho_0 \leq \varrho_3, \varrho_0 \leq \varrho_4, \varrho_3 \leq \varrho_4\}$ and region annotation $\delta_0 = \{\varrho_0 \mapsto \mathsf{d}, \varrho_3 \mapsto \mathsf{d}, \varrho_4 \mapsto \mathsf{d}, \varrho_5 \mapsto \mathsf{d}\}$.

The expression e_0 is not all that interesting for specialization. In program specialization, we usually specialize programs that are functions with some input and output. To this end, consider the following extension of e_0.

EXAMPLE 3.1. $e_1 \equiv \lambda\, z.\, e_0\ \mathtt{at}\ \varrho_6$

This expression is typeable with the following judgment:

$$\{\,\}, C_0 \cup \{\varrho_6 \leq \varrho_4\} \vdash_c \lambda\, z.\, e_0\ \mathtt{at}\ \varrho_6 : ((\alpha, \varrho_6) \xrightarrow{\epsilon'.\varphi'} (\mathtt{int}, \varrho_4), \varrho_6), \{\varrho_6\} \cup \varphi'$$

where $\varphi' = \{\varrho_0, \varrho_3, \varrho_4, \varrho_5\}$. Hence, it is a constraint completion with initial constraint set C_0 and region annotation $\delta_1 = \{\varrho_0 \mapsto \mathsf{s}, \varrho_3 \mapsto \mathsf{s}, \varrho_4 \mapsto \mathsf{d}, \varrho_5 \mapsto \mathsf{s}, \varrho_6 \mapsto \mathsf{d}\}$. In Section 4.4.1, we will specialize e_1.

3.6 Complexity

Theorem 3.1 shows the mere existence of a completion that can be computed in linear time. An implementation would attempt to construct a more interesting completion that keeps the binding times as small as possible. In the polymorphic setting, it means that binding times should only be instantiated if it cannot be avoided.

Such a binding-time analysis runs in time polynomial in the size, n, of the typed input program after ML type inference. First, a simple transformation inserts a `lift` on each subexpression of type `int`. In the worst case, this will double the size of the program. Next, region inference is performed in polynomial time [Tofte and Birkedal 1998; Birkedal and Tofte 2001]. Finally, the constraint analysis decorates the λ^{region} derivation with constraint sets and closes them. To determine the complexity of the latter constraint analysis, we perform a rough worst-case analysis. At most one constraint set is required at each point in the type derivation. The size of each constraint set is clearly bounded by n^2, since the number of different region variables in a program is bounded by n. The rules *(Var)*, *(Const)*, *(App)*, *(Let)*, and *(Effect)* do not contribute to the constraint set at all. The rule *(Lift)* adds one constraint, the rule *(Lam)* adds $O(n)$ constraints due to the well-formedness assumption, and the rule *(Regapp)* adds $O(n^2)$ constraints (since it has to copy C'' in the worst case). Only the projection operations in rules *(Letrec)* and *(Reglam)* require the transitive closure to be computed, which can be implemented in time $O(n^3)$. Each projection operation itself runs in $O(n^2)$ time. Last, at the rule *(Letrec)*, a fixpoint computation is required to determine the resulting constraint set. This requires at most $O(n^2)$ iterations since the set of all constraint sets is a lattice of height $O(n^2)$. Since the constraint sets at individual program points always grow during the computation, it is not necessary to restart nested fixpoint computations every time from scratch (see [Dussart et al. 1995]) and also the computations of

$$
\begin{array}{lll}
\lambda_2^{region}\text{-}\mathsf{Term} \ni E ::= & V \mid R \mid c\ \mathtt{at}\ \mathsf{p} \mid \lambda x.\, E\ \mathtt{at}\ \mathsf{p} \mid E\ @\ E \mid \mathtt{lift}\ [\mathsf{p}_1, \mathsf{p}_2]\ E \mid \mathtt{new}\ \varrho : C\ .E \mid \\
& f\ [\mathsf{p}_1, \ldots, \mathsf{p}_n]\ \mathtt{at}\ \mathsf{p} \mid \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E \rangle_{\mathsf{p}}\ [\mathsf{p}_1, \ldots, \mathsf{p}_n]\ \mathtt{at}\ \mathsf{p} \mid \\
& \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E\ \mathtt{at}\ \mathsf{p}\ \mathtt{in}\ E \mid \mathtt{let}\ x = E\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E \rangle_{\mathsf{p}}\ \mathtt{in}\ E \mid \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, R\ \mathtt{at}\ \mathsf{p}\ \mathtt{in}\ E \mid \mathtt{new}^{\beta}\ \varrho . E
\end{array}
$$

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

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

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

Fig. 3. Syntax of λ_2^{region}

the transitive closures need not be thrown away from one fixpoint iteration to the next. Under the pathological assumption that each expression in the program is a `letrec` expression, a transitive closure must be computed at every subexpression, leading (using the amortization argument of [Dussart et al. 1995]) to a bound of $O(n^4)$ operations.

4. SPECIALIZATION

This section specifies region-based partial evaluation for λ_C^{region} using a small-step structural operational semantics [Plotkin 1981]. The small-step semantics requires to enhance λ_C^{region}-terms with residual terms as well as terms expressing further computation states. Consequently, the resulting calculus, λ_2^{region}, requires additional typing rules for the new terms.

4.1 The Two-Level Calculus λ_2^{region}

Figure 3 defines the syntax of λ_2^{region}. Placeholders may stand not only for region variables, but also for deallocated region variables, $\varrho^{\bullet} \in \mathsf{DeadRegionVar}$, as well as two unique global binding-time regions taken from BTRegion. We assume that DeadRegionVar, BTRegion and RegionVar are mutually disjoint. In analogy to previous work [Helsen and Thiemann 2000b], we deallocate a region by annotating the region variable with $\bullet$. For example, the expression $c\ \mathtt{at}\ \varrho^{\bullet}$ should only occur in dead code, since its evaluation dereferences a deallocated object.

Region placeholders can also be one of the *virtual* binding-time region variables, `s` or `d`. They are a technical device for proving soundness of the constraint analysis as we will explain below. The predicate, $\mathit{Clean}\,(A)$, holds for a set A iff $\mathit{fps}\,(A) \subseteq \mathsf{RegionVar}$. $\mathit{Clean}\,(A)$ enforces that the region placeholders in A only contain region variables, *i.e.*, no deallocated regions or binding-time regions.

A value $V \in \lambda_2^{region}\text{-}\mathsf{Value}$ 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 $c\ \mathtt{at}\ \varrho$ denotes a computation that performs the allocation

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

Fig. 4. Specialization Answers

of the constant. Similarly, there is a region closure pointer $\langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E \rangle_{\mathsf{p}}$. Unlike a value, a closure pointer cannot occur in isolation. It is either bound to a variable in a `letrec` expression or applied to some region arguments.

There are four intermediate constructs: The dynamic $\mathtt{let}^{\mathsf{d}}$ and $\mathtt{letrec}^{\mathsf{d}}$ terms are very much like `let` and `letrec` expressions, except that they require their header expressions to be residual (specialized code). The binding-time annotated region abstraction $\mathtt{new}^{\beta}\, \varrho.E$ is the result of reducing a constraint-annotated region abstraction $\mathtt{new}\, \varrho : C\, .E$. The binding time β is calculated by solving the constraint set C. It is fixed for the rest of the specialization.

The terms of λ_2^{region} also include 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.

The set of free expression variables of an expression E is denoted by $\mathit{free}\,(E)$. We write $\{x \mapsto E'\}E$ for the capture-avoiding substitution of all free occurrences of x in E by the λ_2^{region}-Term E'.

There are λ_2^{region}-Terms that never occur as the final answer of a specialization. The grammar in Figure 4 specifies two subsets of λ_2^{region}-Term to capture this notion. A specialization *answer*, Q, is a term that cannot be further reduced. It is either a residual term R or a *static answer*, Q^{s}. The latter is a value V embedded in a dynamic context: the empty context, a dynamic region abstraction, a $\mathtt{let}^{\mathsf{d}}$ expression or a $\mathtt{letrec}^{\mathsf{d}}$ expression. Having static answers that are not values admits some well-known transformations that improve the binding times, as we will explain below.

Not all region annotations $\delta \in \mathsf{BtAnn}$ make sense for specialization. Region deallocation only takes place at specialization time, so dead region variables must be annotated with s. Furthermore, the static region $\boldsymbol{s}$ is to be annotated with s and the dynamic region $\boldsymbol{d}$ ought to have annotation d. Hence, we define the *initial region annotation* $\delta_{\mathrm{init}} \in \mathsf{BtAnn}$ by $\delta_{\mathrm{init}} = \{\varrho^{\bullet} \mapsto \mathsf{s} \mid \varrho \in \mathsf{RegionVar}\} + \{\boldsymbol{s} \mapsto \mathsf{s}, \boldsymbol{d} \mapsto \mathsf{d}\}$.

4.2 Specialization Semantics

The small-step transition semantics expresses a reduction step by relating two configurations, where a configuration is a pair of a region annotation, $\delta \in \mathsf{BtAnn}$, and a term, $E \in \mathsf{Term}$. Formally, it is a relation $\rightsquigarrow \in \mathcal{P}(\mathsf{BtAnn} \times \mathsf{Term} \times \mathsf{BtAnn} \times \mathsf{Term})$. The region annotation δ records the binding times for regions as they are calculated during specialization. Figure 5, Figure 6, and Figure 7 specify the individual transition rules for $\rightsquigarrow$.

Figure 5 deals with computation steps at specialization time. It corresponds to a dynamic semantics of the original region calculus. Rules (2), (3), and (4) allocate

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

$$\delta, \lambda\, x.\ E\ \texttt{at}\ \varrho \rightsquigarrow \delta, \langle \lambda\, x.\ E\rangle_\varrho \quad \textit{if}\ \delta(\varrho) = \mathsf{s} \tag{3}$$

$$\begin{array}{c}\delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E_1\ \texttt{at}\ \varrho\ \texttt{in}\ E_2 \rightsquigarrow \\ \delta, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E_1\rangle_\varrho\ \texttt{in}\ E_2\end{array} \quad \textit{if}\ \delta(\varrho) = \mathsf{s} \tag{4}$$

$$\delta, \texttt{new}^{\mathsf{s}}\ \varrho.Q \rightsquigarrow \delta, \{\varrho \mapsto \varrho^\bullet\}Q \tag{5}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} \rightsquigarrow \delta, \langle c\rangle_{\varrho_2} \quad \textit{if}\ \delta(\varrho_2) = \mathsf{s} \tag{6}$$

$$\delta, \langle \lambda\, x.\ E\rangle_\varrho\ @\ Q \rightsquigarrow \delta, \texttt{let}\ x = Q\ \texttt{in}\ E \tag{7}$$

$$\delta, \texttt{let}\ x = V\ \texttt{in}\ E \rightsquigarrow \delta, \{x \mapsto V\}E \tag{8}$$

$$\delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E\rangle_\varrho\ [\mathsf{p}'_1, \ldots, \mathsf{p}'_n]\ \texttt{at}\ \varrho_0 \rightsquigarrow \delta, \langle \lambda\, x.\ \{\varrho_1 \mapsto \mathsf{p}'_1, \ldots, \varrho_n \mapsto \mathsf{p}'_n\}E\rangle_{\varrho_0} \tag{9}$$

$$\begin{array}{c}\delta, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E_1\rangle_{\mathsf{p}}\ \texttt{in}\ E_2 \rightsquigarrow \\ \delta, \{f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E_1\rangle_{\mathsf{p}}\ \texttt{in}\ E_1\rangle_{\mathsf{p}}\}E_2\end{array} \tag{10}$$

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

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, Q\ @\ E \rightsquigarrow \delta', Q\ @\ E'} \tag{12}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,E \rightsquigarrow \delta', \texttt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,E'} \tag{13}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let}\ x = E\ \texttt{in}\ E'' \rightsquigarrow \delta', \texttt{let}\ x = E'\ \texttt{in}\ E''} \tag{14}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{new}^{\mathsf{s}}\ \varrho.E \rightsquigarrow \delta', \texttt{new}^{\mathsf{s}}\ \varrho.E'} \tag{15}$$

Fig. 5. Dynamic Semantics of λ_2^{region} - Part I

static constants, lambdas, and region-lambda abstractions respectively. Rule (5) deallocates a static region after specializing the body. The `lift` in rule (6) moves a constant between two static regions and rule (7) prepares a beta-value reduction for static lambdas by translating it to an equivalent `let`. Rule (8) reduces a `let` expression if the header expression is already reduced to a value. The rule may not be applicable immediately after rule (7) because the header may have the form StaticAnswer, but not yet be a value. Static region application is defined in rule (9), and rule (10) unfolds `letrec`-bound functions. Finally, rules (11) to (15) specify a left-to-right call-by-value semantics for static parts of λ_2^{region}.

The next set of rules, in Figure 6, deal with reductions specific to the handling of constraints and the generation of residual code. The binding time of a region is calculated by rule (16). The rule extends the region annotation, δ, such that the constraint set C is solvable. By Lemma 3.12, such an extension always exists. While the rule works for any solution, in practice the partial evaluator will choose the least solution. The side condition $\varrho \notin \mathit{Dom}\,(\delta)$ can trivially be satisfied by α-renaming ϱ in C and E if required.

Rule (17) implements a first-order binding-time coercion: it transforms a static, specialization-time constant into a dynamic constant by moving it from a static region into a dynamic region. Such binding-time coercions are indispensable in

$$\delta, \texttt{new}\ \varrho : C\ .E \rightsquigarrow \delta + \{\varrho \mapsto \beta\}, \texttt{new}^{\beta}\ \varrho.E \quad \textit{if } \varrho \notin \textit{Dom}\,(\delta) \textit{ and } \delta + \{\varrho \mapsto \beta\} \models C \tag{16}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c \rangle_{\varrho_1} \rightsquigarrow \delta, c\ \texttt{at}\ \varrho_2 \quad \textit{if } \delta(\varrho_2) = \texttt{d} \tag{17}$$

$$\delta, c\ \texttt{at}\ \varrho \rightsquigarrow \delta, c\ \underline{\texttt{at}}\ \varrho \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{18}$$

$$\delta, \lambda\, x.\ R\ \texttt{at}\ \varrho \rightsquigarrow \delta, \underline{\lambda}\, x.\ R\ \underline{\texttt{at}}\ \varrho \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{19}$$

$$\delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R_1\ \texttt{at}\ \varrho\ \texttt{in}\ R_2 \rightsquigarrow \delta, \underline{\texttt{letrec}}\ f = \underline{\Lambda}\, \varrho_1, \ldots, \varrho_n.\underline{\lambda}\, x.\ R_1\ \underline{\texttt{at}}\ \varrho\ \underline{\texttt{in}}\ R_2 \tag{20}$$

$$\delta, \texttt{new}^{\texttt{d}}\ \varrho.R \rightsquigarrow \delta, \underline{\texttt{new}}\ \varrho.\ R \tag{21}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\ R \rightsquigarrow \delta, \underline{\texttt{lift}}\,[\varrho_1, \varrho_2]\ R \tag{22}$$

$$\delta, R_1\ @\ R_2 \rightsquigarrow \delta, R_1\ \underline{@}\ R_2 \tag{23}$$

$$\delta, \texttt{let}^{\texttt{d}}\ x = R_1\ \texttt{in}\ R_2 \rightsquigarrow \delta, \underline{\texttt{let}}\ x = R_1\ \underline{\texttt{in}}\ R_2 \tag{24}$$

$$\delta, f\ [\varrho_1, \ldots, \varrho_n]\ \texttt{at}\ \varrho \rightsquigarrow \delta, f\ [\varrho_1, \ldots, \varrho_n]\ \underline{\texttt{at}}\ \varrho \tag{25}$$

$$\delta, \texttt{let}\ x = R\ \texttt{in}\ E \rightsquigarrow \delta, \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E \tag{26}$$

$$\delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \rightsquigarrow \delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{27}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \lambda\, x.\ E\ \texttt{at}\ \varrho \rightsquigarrow \delta', \lambda\, x.\ E'\ \texttt{at}\ \varrho} \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{28}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E\ \texttt{at}\ \varrho\ \texttt{in}\ E'' \rightsquigarrow \delta', \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E'\ \texttt{at}\ \varrho\ \texttt{in}\ E''} \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{29}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{new}^{\texttt{d}}\ \varrho.E \rightsquigarrow \delta', \texttt{new}^{\texttt{d}}\ \varrho.E'} \tag{30}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E \rightsquigarrow \delta', \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E'} \tag{31}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \rightsquigarrow \delta', \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E'} \tag{32}$$

Fig. 6. Dynamic Semantics of λ_2^{region} - Part II

every partial evaluator [Jones et al. 1993; Henglein 1991].

The rules (18) through (25) generate residual code. Each of them requires that all subterms are already residual. In rule (22), we do not reduce the dynamic lift because a lift copies a value from one region to the other, so dropping it would lead to a potential region mismatch and a type error. It cannot be executed either because residual code R does not contain pointers (which are static values). The rules (28) and (29) are context rules that specialize under dynamic lambdas. These rules are the main deviation from a standard semantics. Rules (26) and (27) create the $\texttt{let}^{\texttt{d}}$ and $\texttt{letrec}^{\texttt{d}}$ intermediate expressions and the context rules (30), (31) and (32) specialize inside dynamic contexts.

Recall that static answers Q^{s} are static values embedded in a dynamic context. If a static answer occurs in a static context, the specializer floats the dynamic context

$$\delta, \mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,(\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,\mathrm{Q}^{\mathsf{s}}) \quad \textit{if } \varrho \notin \{\mathsf{p}_1, \mathsf{p}_2\} \tag{33}$$

$$\delta, \mathtt{let}\ x = (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E^{\mu}) \quad \textit{if } \varrho \notin \textit{fps}\,(\mu) \tag{34}$$

$$\delta, (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2^{\mu}) \quad \textit{if } \varrho \notin \textit{fps}\,(\mu) \tag{35}$$

$$\delta, \mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,(\mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ \mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ (\mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,\mathrm{Q}^{\mathsf{s}}) \tag{36}$$

$$\delta, \mathtt{let}\ y = (\mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ \mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ (\mathtt{let}\ y = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E) \quad \textit{if } x \notin \textit{free}\,(E) \tag{37}$$

$$\delta, (\mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ \mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2 \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ (\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2) \quad \textit{if } x \notin \textit{free}\,(Q_2) \tag{38}$$

$$\delta, \mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,(\mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ \mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ (\mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\,\mathrm{Q}^{\mathsf{s}}) \tag{39}$$

$$\delta, \mathtt{let}\ x = (\mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ \mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ (\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E) \quad \textit{if } f \notin \textit{free}\,(E) \tag{40}$$

$$\delta, (\mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ \mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2 \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda\,x.\ R\ \mathtt{at}\ \varrho\ \mathtt{in}\ (\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2) \quad \textit{if } f \notin \textit{free}\,(Q_2) \tag{41}$$

Fig. 7. Dynamic Semantics of $\lambda_{\mathcal{Z}}^{region}$ - Part III

of the answer outside the static context. Thus, the specializer moves the static context towards the embedded static value, allowing further specialization. The required transformation steps are derived from Moggi's work on the computational lambda calculus [Moggi 1991]. The specialization rules in Figure 7 implement these transformations in exactly the manner described in work by Danvy, Hatcliff, Lawall, and one of the authors [Hatcliff and Danvy 1997; Lawall and Thiemann 1997].

In $\lambda_{\mathcal{Z}}^{region}$, there are five such static contexts, which can be read from the context rules (11) through (15). However, as can be seen from reduction rule (7), we recast the static context induced by rule (12) to that of (14). Moreover, the static context from rule (15) need not be transported since rule (5) 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 (33), (36), and (39); a `let` expression, handled by the rules (34), (37), and (40); and the left subterm of function application, covered by the rules (35), (38), and (41).

The side conditions on rules (33), (34), and (35) are necessary to make the specialization step sound: they guarantee the condition $\varrho \notin \textit{fps}\,(TE, \mu)$ in the type rule for region abstraction. These constraints can always be satisfied by α-renaming ϱ in the static answer $\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}$. On the level of expression variables, α-renaming x and f in the rules (37) and (38), as well as (40) and (41), can always satisfy the conditions.

The relation $\overset{*}{\rightsquigarrow}$ is the reflexive and transitive closure of the specialization relation $\rightsquigarrow$. To specialize a program $e' \in \lambda^{region}$-Term, we need a constraint completion $e \in \lambda_C^{region}$-Term with associated region annotation $\delta_c = \{\varrho_1 \mapsto \mathsf{d}, \ldots, \varrho_n \mapsto \mathsf{d}\}$, as we have specified in Section 3.5. The actual input to the specializer is the region annotation $\delta_{\text{init}} + \delta_c$ with the λ_C^{region}-program e. As we shall prove in Section 5, our operational specialization semantics either does not terminate or produces a residual program R for such a completion, *i.e.* $\delta_{\text{init}} + \delta_c, e \overset{*}{\rightsquigarrow} \delta', R$.

4.3 Rewriting in λ^{region}

λ_2^{region} does not only extend λ_C^{region} with two-level binding-time annotations, it also provides further terms required by the small-step semantics. Similar terms are also needed to compute by rewriting in λ^{region} of Section 2. The following grammar extends the terms of λ^{region} with such terms.

$$\begin{aligned}\lambda^{region}\text{-}\mathsf{Term} \ni e \ ::= \ & v \mid x \mid c \ \mathtt{at}\ \mathsf{p} \mid \lambda x.\, e \ \mathtt{at}\ \mathsf{p} \mid e \mathbin{@} e \mid \mathtt{lift}\,[\mathsf{p}_1, \mathsf{p}_2]\, e \mid \\ & \mathtt{new}\ \varrho.\, e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid f\ [\mathsf{p}_1, \ldots, \mathsf{p}_n]\ \mathtt{at}\ \mathsf{p} \mid \\ & \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, e\ \mathtt{at}\ \mathsf{p}\ \mathtt{in}\ e\end{aligned}$$

$$\lambda^{region}\text{-}\mathsf{Value} \ni v ::= \langle c \rangle_{\mathsf{p}} \mid \langle \lambda x.\, e \rangle_{\mathsf{p}} \qquad \mathsf{p} \in \lambda^{region}\text{-}\mathsf{Placeholder} = \mathsf{RegionVar} \cup \{\bullet\}$$

The additional terms are very similar to those of λ_2^{region}, except that instead of annotating a deallocated region variable with $\bullet$, the symbol $\bullet$ stands for a distinguished *dead* region variable. This distinction is sufficient due to the absence of constraints in λ^{region}. In λ_2^{region} it is important to remember the actual region variables being deallocated when building the transitive closure of a constraint set C. Since constraint sets do not exist in λ^{region}, we prefer to stick to this formulation taken from earlier work [Helsen and Thiemann 2000b].

The erasure function for λ_C^{region}-terms extends to λ_2^{region}-terms in a straightforward way, *i.e*, $|\cdot| \in \lambda_2^{region}\text{-}\mathsf{Term} \to \lambda^{region}\text{-}\mathsf{Term}$. It produces a term in λ^{region} by stripping constraints sets, removing underlines from residual terms as well as the binding-time annotations from region binders and the `let` and `letrec` constructs. It also maps all placeholders from DeadRegionVar and BTRegion onto the distinct dead region variable, $\bullet \in \lambda^{region}$-Placeholder. For any two-level term E, it holds that $|E|$ is a term in λ^{region}. Henceforth, λ^{region} includes the extensions of this section.

To extract an operational semantics for λ^{region} from the rules in Figure 5, we proceed as follows. Each rule $\delta, E \rightsquigarrow \delta', E'$ in Figure 5 gives rise to a standard semantics rule $|E| \to |E'|$, where we replace all occurrences of $|Q|$ in $|E|$ and $|E'|$ with v. These rules define a binary reduction relation, $\to$, on λ^{region}-Terms. The relation $\xrightarrow{*}$ is the reflexive transitive closure of the relation $\to$.

For a program specialization $\delta_{\text{init}} + \delta_c, e \overset{*}{\rightsquigarrow} \delta', R$, we shall prove in Section 6 that $|R| \in \lambda^{region}\text{-}\mathsf{Term}$ is contextually equivalent to $|e| \in \lambda^{region}\text{-}\mathsf{Term}$ with respect to the reduction relation $\to$.

4.4 Examples

We will illustrate specialization with some examples.

4.4.1 *Example.* Consider expression e_1 from Example 3.1 again:

$$\begin{aligned}e_1 \equiv \lambda z.\, (&\mathtt{letrec}\ f = \Lambda\, \varrho_1, \varrho_2 . \lambda x.\, (\mathtt{lift}\,[\varrho_1, \varrho_2]\, x)\ \mathtt{at}\ \varrho_0\ \mathtt{in} \\ &(f\ [\varrho_3, \varrho_4]\ \mathtt{at}\ \varrho_5) \mathbin{@} (42\ \mathtt{at}\ \varrho_3))\ \mathtt{at}\ \varrho_6\end{aligned}$$

Recall from Section 3.5 that this expression is its initial constraint completion with initial constraint set $C_1 = \{\varrho_0 \le \varrho_5, \varrho_5 \le \varrho_0, \varrho_5 \le \varrho_3, \varrho_5 \le \varrho_4, \varrho_0 \le \varrho_3, \varrho_0 \le \varrho_4, \varrho_3 \le \varrho_4, \varrho_6 \le \varrho_4\}$ and region annotation $\delta_1 = \{\varrho_0 \mapsto \mathsf{s}, \varrho_3 \mapsto \mathsf{s}, \varrho_4 \mapsto \mathsf{d}, \varrho_5 \mapsto \mathsf{s}, \varrho_6 \mapsto \mathsf{d}\}$.

The expression e_1 specializes with region annotation δ_1 as follows:

$$\begin{aligned}
\delta_1, e_1 &\overset{*}{\rightsquigarrow} \delta_1, \lambda z.\, ((\langle \Lambda\, \varrho_1, \varrho_2.\lambda x.\, e_1' \rangle_{\varrho_0}\ [\varrho_3, \varrho_4]\ \mathtt{at}\ \varrho_5) \mathbin{@} (42\ \mathtt{at}\ \varrho_3))\ \mathtt{at}\ \varrho_6 \\
&\quad \textit{where } e_1' \equiv \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \varrho_2.\lambda x.\, \mathtt{lift}\, [\varrho_1, \varrho_2]\, x \rangle_{\varrho_0}\ \mathtt{in}\ \mathtt{lift}\, [\varrho_1, \varrho_2]\, x \\
&\overset{*}{\rightsquigarrow} \delta_1, \lambda z.\, (\langle \lambda x.\, e_1'' \rangle_{\varrho_5} \mathbin{@} \langle 42 \rangle_{\varrho_3})\ \mathtt{at}\ \varrho_6 \\
&\quad \textit{where } e_1'' \equiv \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \varrho_2.\lambda x.\, \mathtt{lift}\, [\varrho_1, \varrho_2]\, x \rangle_{\varrho_0}\ \mathtt{in}\ \mathtt{lift}\, [\varrho_3, \varrho_4]\, x \\
&\overset{*}{\rightsquigarrow} \delta_1, \lambda z.\, (\mathtt{lift}\, [\varrho_3, \varrho_4]\, \langle 42 \rangle_{\varrho_3})\ \mathtt{at}\ \varrho_6 \\
&\overset{*}{\rightsquigarrow} \delta_1, \lambda z.\, (42\ \mathtt{at}\ \varrho_4)\ \mathtt{at}\ \varrho_6 \\
&\overset{*}{\rightsquigarrow} \delta_1, \underline{\lambda}\, z.\, (42\ \underline{\mathtt{at}}\ \varrho_4)\ \underline{\mathtt{at}}\ \varrho_6
\end{aligned}$$

4.4.2 *Example.* We now give a somewhat larger example, where λ^{region} is extended with a conditional. The semantics of the conditional is standard: if the condition evaluates to 0, the *if*-expression evaluates to the `else`-branch. Otherwise, the *if*-expression evaluates to the `then`-branch.

In the following expression, d is some dynamic input and there are `lift` expressions around some expressions of type `int`.[6]

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

Applying region inference to the expression yields:

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

The annotations are obtained by pretty-printing and renaming intermediate code produced by the ML-kit [Tofte et al. 2001]. 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 of an expression.

The type with place of the program is $((\mathtt{int}, \varrho_2) \xrightarrow{\epsilon.\varphi} (\mathtt{int}, \varrho_3), \varrho_1)$ where the effect φ is $\{\varrho_2, \varrho_3\}$ and the program's effect is $\{\varrho_1\} \cup \varphi$. Next, we apply the constraint analysis to the region-annotated version of our program. Figure 8 shows the resulting term. A suitable initial well-formed constraint set for this example is $C = \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \varrho_3\}$ and the region annotation associated with the constraint completion is $\delta_c = \{\varrho_1 \mapsto \mathtt{d}, \varrho_2 \mapsto \mathtt{d}, \varrho_3 \mapsto \mathtt{d}\}$. Clearly, $\delta_{\text{init}} + \delta_c \models C$ and

[6] Recall that `lift` expressions in λ^{region} correspond to the identity function. Such `lift` expressions are automatically inserted after type inference, but before region inference, around *all* expressions of base type. For readability, the example only shows the interesting `lift`s.

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

Fig. 8. Example after Constraint Analysis

specialization with $\delta_{\text{init}} + \delta_c$ produces the following residual program:

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

All regions, except ϱ_{18} and the regions ϱ_1, ϱ_2 and ϱ_3 from the constraint set C obtain a static binding time. In this particular example, it is possible to calculate the binding times of the regions before the specialization phase, since the example does not make use of polymorphic region recursion here. However, the binding-time polymorphism of function *apply* is exploited in its two uses with different binding-time arguments.

4.4.3 *Example.* In the final example we show that the conservative treatment of the effect in type rule *(Lam)* is necessary to obtain sound specialization. Consider the program:

$$(\texttt{let}\ g = \lambda\, z.\, z\ \texttt{in}\ \lambda\, y.\,\texttt{let}\ f = \lambda\, x.\,(g \mathbin{@} (\lambda\, y.\, y))\ \texttt{in}\ (\lambda\, f.\, 4) \mathbin{@} f) \mathbin{@} 2$$

The following region-annotated version is typeable in the original formulation of the type system:

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

The problem here is the function bound to f. It is effectively *dead* code, which means that it is only allocated but never used. Region inference is normally 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 for the evaluation semantics $\rightarrow$, but is not sound with specialization $\rightsquigarrow$ if the region bound to 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 typing judgment $\vdash_t$ avoids this undesirable behavior, by imposing a slightly more conservative annotation:

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

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

4.5 Extending the Static Semantics

To prove soundness of the constraint analysis with respect to specialization, we extend the static semantics of Section 3.3 to all λ_2^{region}-Terms. All semantic objects retain their definitions, with the extension that region placeholders now also range over deallocated regions from DeadRegionVar and the two virtual binding-time regions. Only the definition of a substitution S_s requires a small extension: for all $\mathsf{p} \in \mathsf{DeadRegionVar} \cup \mathsf{BTRegion}$, it holds that $S_s(\mathsf{p}) = \mathsf{p}$.

The extension of the notion of a placeholder makes the type rules of Figure 2 work over deallocated regions as well. Since λ_2^{region}-Term is a superset of the terms covered there, all typing rules can be reused. Figures 9 and 10 show the typing rules for the remaining terms.

The rules *(Constv)*, *(Lamv)*, and *(Letrecv)* are the pointer counterparts of rules *(Const)*, *(Lam)*, and *(Letrec)* in Figure 2. In contrast to the allocating expressions, pointers have no effect. Moreover, the rules *(Constv)* and *(Lamv)* 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 only occur during specialization. The rule *(Regappv)* types the application of a region closure and is quite similar to rule *(Regapp)*. For more elaboration, we refer to work on a small-step semantics for the region calculus [Helsen and Thiemann 2000b].

The rule *(Constraint)* is an auxiliary rule, which removes constraints involving deallocated regions. However, this is only allowed if the constraint $\varrho^\bullet \leq \boldsymbol{s}$ 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 $C_{\downarrow\varrho^\bullet}$.

The rule *(Reglam-s)* deals with a static region abstraction. It is almost identical to rule *(Reglam)*, except for the additional binding-time constraint which guarantees that the abstracted region is statically annotated.

Similarly, rule *(Reglam-d)* in Figure 10 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 2, rule *(Letrec-d)* only adds a binding-time constraint, which requires that the `letrec`-bound function is dynamic.

The typing rules for residual expressions are also very similar to their non-residual counterparts from Figure 2. We only list those rules that slightly deviate from their non-residual counterpart. The rules *(Const-r)* and *(Lam-r)* add the constraint $\boldsymbol{d} \leq \varrho$ to their constraint sets. Similarly, the rules *(Regapp-r)*, *(Lift-r)* and *(Reglam-r)* add a constraint to guarantee that every solution for C maps the top-level region

(Constv)
$$\frac{C \vdash_{wft} (\mathtt{int}, \mathsf{p}) \quad C \rhd \{\mathsf{p} \leq s\}}{TE, C \vdash_c \langle c \rangle_{\mathsf{p}} : (\mathtt{int}, \mathsf{p}), \emptyset}$$

(Lamv)
$$\frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi \\ C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \mathsf{p}) \quad \varphi \subseteq \varphi' \quad C \rhd \{\mathsf{p} \leq s\} \end{array}}{TE, C \vdash_c \langle \lambda\, x.\, E \rangle_{\mathsf{p}} : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \mathsf{p}), \emptyset}$$

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

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

(Constraint)
$$\frac{TE, C \vdash_c E : \mu, \varphi \quad C \rhd \{\varrho^\bullet \leq s\}}{TE, C_{\downarrow \varrho^\bullet} \vdash_c E : \mu, \varphi \setminus \{\varrho^\bullet\}}$$

(Reglam-s)
$$\frac{TE, C \vdash_c E : \mu, \varphi \quad \varrho \notin fps\,(TE, \mu) \quad C \rhd \{\varrho \leq s\}}{TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}^{\mathsf{s}}\ \varrho.E : \mu, \varphi \setminus \{\varrho\}}$$

Fig. 9. Static Semantics of λ_2^{region} - Part I

onto d. At this point, it may be tempting to assign empty effects to residual syntax since the construction of code does not affect any region during specialization. However, to prove the specializer sound with respect to a standard semantics, the erasures of the above rules must reflect the corresponding standard typing rules.

Constraints of the form $\varrho \leq s$ and $\mathsf{d} \leq \varrho$ are only introduced in the type rules for terms that are calculated during specialization. Hence, they do not compromise the solvability of the constraint set of a constraint completion.

Under the extended type system for λ_2^{region}, it is easy to verify that the properties in Section 3.4 remain valid. Recall that the type system for λ_2^{region} induces a type system for λ^{region}, *i.e.*, $TE, C \vdash_c E : \mu, \varphi$ defines $|TE| \vdash_t |E| : |\mu|, |\varphi|$. This also holds for the extended rules of Figure 9 and Figure 10.

5. SOUNDNESS OF THE CONSTRAINT ANALYSIS

The correctness proof of the constraint-analysis closely follows the structure of the soundness proof for the region calculus [Helsen and Thiemann 2000b], extended to λ_2^{region}. It relies crucially on two substitution lemmas. They show that a syntactic substitution in an expression does not affect the expression's type.

Values in λ_2^{region} have no effect:

LEMMA 5.1. *If* $TE, C \vdash_c V : \mu, \varphi$ *then* $\varphi = \emptyset$.

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

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

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

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

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

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

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

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

Fig. 10. Static Semantics of λ_2^{region} - Part II

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

LEMMA 5.2. *Suppose* $TE, C \vdash_c R : (\tau, \varrho), \varphi$; $C \vdash_{\mathit{wft}} TE$; $\delta \models C$ *and for all* $(\sigma, \mathtt{p}') \in \mathit{Ran}\,(TE)$, *it holds that* $\delta(\mathtt{p}') = \mathtt{d}$. *Then,* $\delta(\varrho) = \mathtt{d}$.

PROOF. Rule induction on the derivation of $TE, C \vdash_c R : (\tau, \varrho), \varphi$. □

The condition that for all $(\sigma, \mathtt{p}) \in \mathit{Ran}\,(TE)$, it holds that $\delta(\mathtt{p}) = \mathtt{d}$ is used in several properties. It expresses that a type environment has to be dynamic in a specialization context. Using the less restrictive condition that "for all $x \in \mathit{free}\,(R)$ where $TE(x) = (\sigma, \mathtt{p})$ it holds that $\delta(\mathtt{p}) = \mathtt{d}$" would not make a difference.

The *first substitution lemma* for `let`-bound variables is mostly standard (see for instance Lemma 4.4 in [Wright and Felleisen 1994]).

LEMMA 5.3. *Suppose* $TE + \{x \mapsto \mu\}, C \vdash_c E : \mu', \varphi'$ *and* $TE, C \vdash_c V : \mu, \emptyset$, *then* $TE, C \vdash_c \{x \mapsto V\}E : \mu', \varphi'$.

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, $\{x \mapsto V\}x = V$ and, by the second assumption, $TE, C \vdash_c V : \mu, \emptyset$ since $\varphi' = \emptyset$, by Lemma 5.1.

All other cases are simple appeals to the induction hypothesis. □

Since the operational semantics of λ_2^{region} also substitutes a polymorphic region closure pointer – see reduction rule (10) – there is a *second substitution lemma*. It is slightly more technical:

LEMMA 5.4. *Suppose*

(1) $P = \{\vec{\varrho}\}$ *and* $(P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\mathsf{p}\}) = \emptyset$ *and*
$\{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset$ *and* $\sigma = \forall\,\vec{\alpha}\,.\hat{\sigma}$ *and* $\hat{\sigma} = \forall\,\vec{\epsilon}.\forall\,\vec{\varrho}.C^{\uparrow P} \Rightarrow \tau$

(2) $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$

(3) $TE + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_{\mathsf{p}} : (\tau, \mathsf{p}), \emptyset$

Then
$TE, C_{\downarrow P} \vdash_c \{f \mapsto \langle \Lambda\,\vec{\varrho}.\lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\,\vec{\varrho}.\lambda\, x.\, E_1 \rangle_{\mathsf{p}}\ \mathtt{in}\ E_1 \rangle_{\mathsf{p}}\} E_2 : \mu, \varphi$.

PROOF. The proof is an adaptation of the proof for Lemma 5.6 in [Helsen and Thiemann 2000b], extended to handle constraint sets. It can be found in Appendix A. □

Type preservation states that an expression's type is not affected 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, the new effect is smaller, the new constraint set and region annotation larger, and the latter still solves the former.

PROPOSITION 5.1. *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 $\mathsf{p} \leq \mathsf{s}$ or $\mathsf{d} \leq \mathsf{p}$ to reflect binding-time knowledge into the type system: in particular it holds that $C_{\downarrow\{\mathsf{s},\mathsf{d}\}} = C'_{\downarrow\{\mathsf{s},\mathsf{d}\}}$.

PROOF. See Appendix B □

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

LEMMA 5.5. *Suppose* $TE, C \vdash_c Q^\mu : \mu, \varphi$; $\mu = (\tau, \mathsf{p})$; $C \vdash_{wft} TE$; $\delta \models C$ *and for all* $(\sigma, \mathsf{p}') \in Ran\,(TE)$, *it holds that* $\delta(\mathsf{p}') = \mathsf{d}$, *then*

(1) *if* $\delta(\mathsf{p}) = \mathsf{d}$, *then* Q^μ *is residual code of the form* R

(2) *if* $\delta(\mathsf{p}) = \mathsf{s}$, *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* $\tau = \mathtt{int}$, *then* V *has the form* $\langle c \rangle_{\mathsf{p}}$ *for some constant* c
   (b) *if* $\tau = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$, *then* V *has the form* $\langle \lambda\, x.\, E \rangle_{\mathsf{p}}$ *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 9 and Figure 10 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. This is called *progress*.

PROPOSITION 5.2. *Suppose for a* λ_2^{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* (φ); $C \vdash_{wft} TE$; *and for all* $(\sigma, \mathsf{p}') \in Ran\,(TE)$, *it holds that* $\delta(\mathsf{p}') = \mathsf{d}$. *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 specialization immediately reduces static expression binders such as `let` and lambda. However, since specialization proceeds inside of dynamic lambdas and `let` expressions, the type environment is not generally empty as is usually the case in proofs of the progress property. The condition *Clean* (φ) avoids that deallocated regions or the global static or dynamic region are touched during reduction.

PROOF. See Appendix C □

The constraint analysis can now be proven sound by combining type preservation and progress in the usual way:

THEOREM 5.1. *Suppose a* λ_2^{region}*-program* E^μ *for which* $\{\}, C \vdash_c E^\mu : \mu, \varphi$. *Additionally, assume that* $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$ *and that Clean* (φ), *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* $\{\}, C' \vdash_c Q : \mu, \varphi'$ *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 $\delta, E \overset{*}{\rightsquigarrow} \delta', E'$. The proof is by induction on the length of the reduction. For the base case, assume that $E = E'$. Then, by Proposition 5.2 (Progress) either Item 1 or 2 applies. Alternatively, we have $\delta, E \rightsquigarrow \delta'', E''$ and $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', E'$. By Proposition 5.1 (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, the specialization of a constraint completion always yields a residual program, provided partial evaluation terminates.

COROLLARY 5.1. *Suppose* e *is a constraint completion with region annotation* δ_c, i.e. $\{\}, C \vdash_c e^{(\tau,\varrho)} : (\tau, \varrho), \varphi$; $\delta_c \models C$; $\delta_c(\varrho) = \mathsf{d}$ *and for all* $\mathsf{p}' \in \varphi$, *it holds that* $\delta_c(\mathsf{p}') = \mathsf{d}$.

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 $\delta = \delta_{\text{init}} + \delta_c$. The condition that for all $\mathsf{p}' \in \varphi$ holds $\delta_c(\mathsf{p}') = \mathsf{d}$, makes *Clean* (φ). Also, since $\delta_c \models C$, we automatically have $\delta \models C$ as well. Because of constraint-analysis soundness, we either have non-termination or $\delta, e \overset{*}{\rightsquigarrow} \delta', Q$, 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 the fact that $\delta \subseteq \delta'$), 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 it holds that $\{\,\}, C \vdash_c R : \mu, \varphi$ implies $\{\,\} \vdash_t |R| : |\mu|, |\varphi|$ and $|R| \in \lambda^{region}$-Term.

6. 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 [Milner 1978]. Beyond that, full specialization correctness requires that the residual program is *semantically equivalent* to the source program applied to the static parameters. This can be ensured by showing that the reductions of the specializer are contextual equivalences in the region calculus. In a companion paper [Helsen 2004], the first author has developed an equational theory for λ^{region} and proved it sound with respect to Morris-style contextual equivalence [Morris 1968]. The soundness proof of the specializer relies crucially on these results so the first subsection recalls this equational theory for λ^{region} and the second subsection applies the theory to the present specialization setting.

6.1 An Equational Theory for λ^{region}

The original formulation of the equational theory [Helsen 2004] is defined for a region calculus which treats recursive functions as first class objects. This calculus is a generalization of λ^{region} and it is a straightforward exercise to reformulate it accordingly. Additionally, for conciseness of the presentation, the current exposition only considers a monomorphic subset of λ^{region} without constants or the recursive `letrec`-construct.

Recall that the type system used by the constraint analysis is slightly more conservative than the usual region type system given in the literature [Tofte and Talpin 1997; Helsen and Thiemann 2000b; Helsen 2004]. Therefore, we define a type system for the judgment $TE \vdash^t_t e : \mu, \varphi$ with identical rules as for the judgment $TE \vdash_t e : \mu, \varphi$, where type rule *(Lam)* is replaced by the following rule:

$$\textit{(Lam')} \qquad \frac{TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash^t_t \lambda x.\, e \;\mathtt{at}\; \mathsf{p} : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \mathsf{p}), \{\mathsf{p}\}}$$

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

PROPOSITION 6.1. *Suppose* $TE \vdash_t e : \mu, \varphi$. *Then* $TE \vdash^t_t e : \mu, \varphi'$, *where* $\varphi' \subseteq \varphi$.

PROOF. By induction on the type derivation. All cases are immediate, except for rule *(Lam)*. So, suppose $TE \vdash_t \lambda x.\, e \;\mathtt{at}\; \mathsf{p} : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \mathsf{p}), \varphi \cup \{\mathsf{p}\}$ 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,

$$\textit{(Comp-Var)}\quad \frac{TE(x) = (\mu, \mathsf{p}) \quad \mathit{Clean}\,(\varphi)}{TE \rhd x\,\widehat{\mathcal{R}}\,x : (\mu, \mathsf{p}), \varphi}$$

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

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

$$\textit{(Comp-Reglam)}\quad \frac{TE \rhd e\,\mathcal{R}\,e' : \mu, \varphi' \quad \{\varrho, \epsilon\} \cap \mathit{fv}\,(TE, \mu) = \emptyset \quad \mathit{Clean}\,(\varphi)}{TE \rhd \mathtt{new}\ \varrho.\, e\,\widehat{\mathcal{R}}\,\mathtt{new}\ \varrho.\, e' : \mu, (\varphi' \setminus \{\varrho, \epsilon\}) \cup \varphi}$$

$$\textit{(Comp-App)}\quad \frac{\begin{array}{c} TE \rhd e_1\,\mathcal{R}\,e_1' : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi_1 \\ TE \rhd e_2\,\mathcal{R}\,e_2' : \mu_1, \varphi_2 \quad \epsilon' \notin \mathit{fev}\,(TE, \mu_2) \quad \mathit{Clean}\,(\varphi) \end{array}}{TE \rhd e_1\,@\,e_2\,\widehat{\mathcal{R}}\,e_1'\,@\,e_2' : \mu_2, (\varphi' \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \varrho\} \cup \varphi) \setminus \{\epsilon'\}}$$

$$\textit{(Comp-Let)}\quad \frac{\begin{array}{c} TE \rhd e_1\,\mathcal{R}\,e_1' : \mu', \varphi_1 \quad \mathit{Clean}\,(\varphi) \\ TE + \{x \mapsto \mu'\} \rhd e_2\,\mathcal{R}\,e_2' : \mu, \varphi_2 \quad \epsilon' \notin \mathit{fev}\,(TE, \mu) \end{array}}{TE \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\,\widehat{\mathcal{R}}\,\mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2' : \mu, (\varphi_1 \cup \varphi_2 \cup \varphi) \setminus \{\epsilon'\}}$$

Fig. 11. Compatible Refinement

we also have $\varphi' \subseteq \varphi''$ and we can apply rule *(Lam')* to obtain $TE \vdash_t^t \lambda x.\, e\ \mathtt{at}\ \mathsf{p} : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \mathsf{p}), \{\mathsf{p}\}$. Obviously it holds that $\{\mathsf{p}\} \subseteq \{\mathsf{p}\} \cup \varphi$, 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 $\mathsf{TypedRel} = \mathcal{P}(\mathsf{TypeEnv} \times \lambda^{region}\text{-}\mathsf{Term} \times \lambda^{region}\text{-}\mathsf{Term} \times \mathsf{TypeWPlace} \times \mathsf{Effect})$. 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 6.1. *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 $\mathit{Clean}\,(\varphi)$ 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 $\mathit{Clean}\,(\varphi')$, 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 $\mathit{Clean}\,(\varphi)$.

Given a typed relation $\mathcal{R}$, its *compatible refinement* $\widehat{\mathcal{R}} \in \mathsf{TypedRel}$ is the least relation closed under the rules of Figure 11. The refinement rules are very much binary extensions of the (monomorphic) type rules in Figure 2 where type rule *(Effect)* has been integrated in the others.

Furthermore, 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 typed contexts. This notion forms the basis for specifying an equational theory and a notion of contextual equivalence for λ^{region} below. 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.

$$\textit{(Eq-Comp)}\quad \frac{TE \rhd e \mathrel{\widehat{\triangleq}} e' : \mu, \varphi}{TE \rhd e \triangleq e' : \mu, \varphi} \qquad \textit{(Eq-Symm)}\quad \frac{TE \rhd e' \triangleq e : \mu, \varphi}{TE \rhd e \triangleq e' : \mu, \varphi}$$

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

$$\textit{(Eq-Lam)}\quad \frac{TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2, \varphi'' \quad \varphi'' \subseteq \varphi' \quad \varrho \in \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd \lambda\, x.\, e \;\mathtt{at}\; \varrho \triangleq \langle \lambda\, x.\, e \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi}$$

$$\textit{(Eq-Let)}\quad \frac{TE \vdash^t_t v : \mu', \emptyset \quad TE + \{x \mapsto \mu'\} \vdash^t_t e : \mu, \varphi' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd \mathtt{let}\; x = v \;\mathtt{in}\; e \triangleq \{x \mapsto v\}e : \mu, \varphi}$$

$$\textit{(Eq-App-Cbv)}\quad \frac{\begin{array}{c} TE \vdash^t_t \langle \lambda\, x.\, e_1 \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu, \varrho), \emptyset \\ TE \vdash^t_t e_2 : \mu_1, \varphi' \quad \varphi' \cup \varphi'' \cup \{\varrho, \epsilon\} \subseteq \varphi \quad \mathit{Clean}\,(\varphi) \end{array}}{TE \rhd \langle \lambda\, x.\, e_1 \rangle_\varrho \mathbin{@} e_2 \triangleq \mathtt{let}\; x = e_2 \;\mathtt{in}\; e_1 : \mu, \varphi}$$

$$\textit{(Eq-App-Let)}\quad \frac{\begin{array}{c} TE \vdash^t_t e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu, \varrho), \varphi_1 \quad TE \vdash^t_t e_2 : \mu_1, \varphi_2 \\ f \notin \mathit{free}\,(e_2) \quad \varphi' \cup \varphi_1 \cup \varphi_2 \cup \{\varrho, \epsilon\} \subseteq \varphi \quad \mathit{Clean}\,(\varphi) \end{array}}{TE \rhd e_1 \mathbin{@} e_2 \triangleq \mathtt{let}\; f = e_1 \;\mathtt{in}\; f \mathbin{@} e_2 : \mu, \varphi}$$

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

$$\textit{(Eq-Drop)}\quad \frac{TE \vdash^t_t e : \mu, \varphi' \quad \varrho \notin \mathit{fps}\,(TE, \mu) \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd e \triangleq \mathtt{new}\; \varrho.\, e : \mu, \varphi}$$

$$\textit{(Eq-dealloc)}\quad \frac{TE \vdash^t_t e : \mu, \varphi' \quad \varrho \notin \mathit{fps}\,(\mu) \cup \varphi' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd e \triangleq \{\varrho \mapsto \bullet\}e : \mu, \varphi}$$

Fig. 12. Equational Theory

There is now sufficient machinery to define a typed equational theory on λ^{region}-Terms. Define the relation $\triangleq$ as the least typed relation closed under the rules in Figure 12. The rules *(Eq-Comp)*, *(Eq-Symm)* and *(Eq-Trans)* build the equivalent and compatible closure. Rules *(Eq-Lam)*, *(Eq-App-Cbv)* and *(Eq-Let)* are the call-by-value beta-rules from the operational semantics, interpreted as equalities. Rule *(Eq-App-Let)* models the fact that function application is left-to-right in λ^{region}, 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 provably dead. It is easy to verify that the equational theory $\triangleq$ is well-formed.

Intuitively, two λ^{region}-terms are contextually equivalent whenever arbitrary (well-typed and closed) contexts exhibit identical termination behavior. A technique due to Lassen [Lassen 1998] formalizes this fact: a λ^{region}-term e converges ($e \downarrow$) iff there exists a value v such that $e \xrightarrow{*} v$.

The *context closure* $\mathcal{R}^c$ of a typed relation $\mathcal{R}$ is the least relation closed under

the rules

$$\frac{TE \rhd e_1 \,\mathcal{R}\, e_2 : \mu, \varphi}{TE \rhd e_1 \,\mathcal{R}^c\, e_2 : \mu, \varphi} \qquad \frac{TE \rhd e_1 \,\widehat{\mathcal{R}^c}\, e_2 : \mu, \varphi}{TE \rhd e_1 \,\mathcal{R}^c\, e_2 : \mu, \varphi}$$

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

Definition Contextual Equivalence. *Contextual equivalence, $\simeq$, is a well-formed typed relation on λ^{region}-terms such that $TE \rhd e_1 \simeq e_2 : \mu, \varphi$ iff for all e_1', e_2' μ' and φ' such that $\{\,\} \rhd e_1' \,\{(TE, e_1, e_2, \mu)\}^c\, e_2' : \mu', \varphi'$ 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 contextually equivalent:

Theorem 6.1. *If $TE \rhd e_1 \triangleq e_2 : \mu, \varphi$, then also $TE \rhd e_1 \simeq e_2 : \mu, \varphi$.*

Proof. The proof uses a theory of bisimilarity and can be found elsewhere [Helsen 2002; 2004]. □

6.2 Specialization Soundness

We now prove that the specialization rules from the Figures 5, 6, and 7 are sound. The first lemma establishes a technical property of the effect of an answer. The effect can always be shrunk to the point where all regions in it are annotated dynamic.

Lemma 6.1. *Suppose $TE, C \vdash_c Q^\mu : \mu, \varphi$ such that $C \vdash_{wft} TE$ and $\delta \models C$ and for all $(\sigma, \mathsf{p}') \in Ran\,(TE)$ it holds that $\delta(\mathsf{p}') = \mathsf{d}$. Then, $TE, C \vdash_c Q^\mu : \mu, \varphi_0$ with $\varphi_0 \subseteq \varphi$ and for all $\mathsf{p}' \in \varphi_0$ it holds that $\delta(\mathsf{p}') = \mathsf{d}$.*

Proof. By induction on $TE, C \vdash_c Q^\mu : \mu, \varphi$ using Lemma 5.1. □

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

Proposition 6.2. *Suppose $TE, C \vdash_c E : \mu, \varphi$ where $\delta \models C$ and $C \vdash_{wft} TE$ such that for all $(\sigma, \mathsf{p}') \in Ran\,(TE)$ it holds that $\delta(\mathsf{p}') = \mathsf{d}$ and also $Clean\,(\varphi)$. Then, $\delta, E \rightsquigarrow \delta', E'$ implies $TE \rhd |E| \triangleq |E'| : \mu, \varphi$.*

Proof. By Proposition 5.1 (type preservation) we have that $TE, C' \vdash_c E' : \mu, \varphi'$ and $\varphi' \subseteq \varphi$. Therefore, we also have $Clean\,(\varphi')$. We prove by rule induction on $\rightsquigarrow$ and consider each reduction rule from the Figures 5, 6 and 7, that does not reduce a polymorphic or constant term.

Rule (3). Immediate by *(Eq-Lam)*.

Rule (5). By type rule *(Reglam-s)* we have $TE, C \vdash_c \mathtt{new}^{\mathsf{s}}\, \varrho.Q : \mu, \varphi_0 \setminus \{\varrho\}$, and hence $TE, C \vdash_c Q : \mu, \varphi_0$ such that $\varrho \notin fps\,(TE, \mu)$ and $C \rhd \{\mathsf{s} \leq \varrho\}$. From Lemma 6.1 and the assumptions, we derive that $TE, C \vdash_c \mathtt{new}^{\mathsf{s}}\, \varrho.Q : \mu, \varphi_1$ where $\varrho \notin fps\,(TE, \mu) \cup \varphi_1$ and $\varphi_1 \subseteq \varphi_0$. Via Proposition 6.1, this implies $|TE| \vdash_t^t |\mathtt{new}^{\mathsf{s}}\, \varrho.Q| : |\mu|, \varphi_1'$ with $\varphi_1' \subseteq |\varphi_1| \subseteq |\varphi_0|$ and hence also $\varrho \notin fps\,(|TE|, |\mu|) \cup \varphi_1'$. The equality now easily follows from the rules *(Eq-Drop)* and *(Eq-Dealloc)*.

Rule (7). Immediate by equational rule *(Eq-App-Cbv)*.

Rule (8). Follows from rule *(Eq-Let)*.

Rules (11), (12), (14) and (15). Immediate applications of the induction hypothesis.

Rules (16), (19), (21), (23), (24) and (26). Here, both the left-hand and right-hand side yield identical (erased) expressions and reflexivity of $\triangleq$ proves these cases.

Rules (28), (30) and (31). Immediate applications of the induction hypothesis.

Rule (34). 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 (35). Analogous the previous case but with *(Comp-App)* instead.

Rule (37). Immediate from rule *(Eq-Let-Let)*.

Rule (38). Use rule *(Eq-App-Let)* on the left-hand side, followed by *(Eq-Let-Let)*. Application of *(Eq-App-Let)* then proves the case.

□

The soundness theorem for specialization now follows from the preceding proposition.

THEOREM 6.2. *Suppose* $\{\}, C \vdash_c E : \mu, \varphi$*;* $\delta \models C$*;* $\delta_{\text{init}} \subseteq \delta$*; and* $\mathit{Clean}\,(\varphi)$*. Additionally, we have* $\delta, E \overset{*}{\rightsquigarrow} \delta', Q$*. 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 $\triangleq$ and Theorem 6.1. Otherwise we have that $\delta, E \rightsquigarrow \delta'', E''$ and $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', Q$. We apply Proposition 6.2 to obtain $\{\} \rhd |E| \triangleq |E''| : \mu, \varphi$. Using Proposition 5.1 (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 $\{\} \rhd |E''| \triangleq |Q| : \mu, \varphi$ and we prove the claim by rule *(Eq-Trans)* and Theorem 6.1. □

7. ASSESSMENT

7.1 Binding-time Analysis

Most binding-time analyses used in practice transform an un-annotated term into a completion that is in some sense minimal. It would be nice to have a similar notion to characterize the results of our analysis.

In our framework, an accurate region reconstruction is a good preprocessing phase for the constraint analysis. If only one region is allocated for the entire program, no specialization whatsoever will take place as this region has to be dynamic. Hence, the region analysis should separate values in as many regions as possible since every region can have a different binding time. Indeed, a good region separation coincides with the goals of region inference in a memory management setting, where more regions allow more fine-grained reclamation of memory [Tofte and Talpin 1997]. This coincidence motivates the use of region inference for binding-time analysis purposes. Unfortunately, it is still an open question if the type system of λ^{region} has principal types. As a consequence, our approach does not provide a principal binding-time type either, unlike Dussart et al. [1995]. It also remains to be seen if a more direct way of defining a polymorphic BTA for an ML core language can yield better results than our framework.

The constraint analysis in Section 3 should be considered as a specification. Its efficient implementation is an important but separate issue. In particular, the constraint completion constructed in the proof of Theorem 3.1 can be computed in linear time, but it is not very interesting. It satisfies the constraints by mapping all variables to `d`, so that no specialization will take place. We conjecture that it is possible to apply constraint simplification rules along the lines of Dussart et al. [1995] to define a normalized constraint completion, given an explicitly region-typed program. Pursuing this line of work would give rise to a relative completeness result.

Our framework, like every framework dealing with binding-time polymorphism, requires some binding-time computations at specialization time. The total number of constraints that must be solved at each region allocation is bounded by the number of regions in the program since the set C in the expression $\texttt{new}\,\varrho : C\,.e$ only contains constraints that mention ϱ. In other words, we have at most $O(n)$ constraints to solve at each region allocation during specialization. Whether this bound presents a problem in practice can only be determined by experimentation.

Generally, program transformations may have an impact on the outcome of the binding-time analysis. For the region calculus, it is well-known that sometimes transformations are required to improve the space behavior of the program. These transformations usually shrink the scope of a region or split a region into smaller regions. Since splitting a region also leads to a more precise constraint analysis, it may leaad to better binding-time behavior, too.

On the other hand, the cps-transformation [Plotkin 1975] is known to improve the binding-time separation [Consel and Danvy 1991] while it is deadly for the space behavior of a program in the region calculus. The problem is that after cps-transformation, all functions return only at the end of the program. Since allocation and deallocation are tied to entry and exit of the lexical scope of the `new` expression, all deallocation happens at the end of the program.

However, deallocation is part of the store-allocation aspect of the region calculus, which is an additional layer on top of the data-flow aspect. So, even for a cps-transformed program, the region analysis would make sense as a data-flow analysis for the constraint analysis. This observation suggests that a more direct flow analysis on the ML-core calculus, ignoring memory allocation issues, could be derived from the region-based presentation in this paper. However, this defeats one of the main advantages of using the region calculus, namely our ability to reuse existing region inference algorithms and implementations.

7.2 Program-point Specialization

The present framework does not address program-point specialization or memoization [Thiemann 1996; Jones et al. 1993]. That is, (recursive) functions are either static and completely unfolded or dynamic and never unfolded. Program-point specialization enables the creation of specialized instances of a recursive function, indexed by the static values supplied. Program-point specialization is an important feature because it is the only mechanism to avoid non-terminating specialization, in general.[7]

[7] Of course, additional program analysis is required to determine where to use this mechanism [Glenstrup and Jones 1996; Lee et al. 2001; Lee 2002a; 2002b].

For now, we have refrained from adding program-point specialization for two reasons. First, it is not obvious how to add it to a framework based on a small-step semantics. Second, the semantics would become even more involved because it would have to maintain a cache of completed and pending specializations (see [Helsen and Thiemann 2000a] for a denotational approach for a first-order functional language). We consider the addition of program-point specialization an important area of future work.

7.3 Generating Extensions

To avoid problems with encoding types in an interpretation-based partial evaluator for a typed source language, it is desirable to implement specialization by translating the annotated source language program back into the source language itself [Birkedal and Welinder 1993; Launchbury and Holst 1991], by using appropriate code generating functions to implement the annotations [Thiemann 1999a]. The development of such a translation for the specialization technique presented in this work is particularly challenging because the binding times only become known at specialization time. That is, the same variable may be bound to a function in one context and to code (for a function) in another. The standard way of resolving this conflict in ML is to introduce runtime tagging, and hence inefficiency. This problem may be addressed using a richer type language, for instance with dependent types or first-class modules, and requires further investigation.

7.4 Towards Standard ML

The present work does not deal with all features of the Standard-ML core language. For example, algebraic datatype and pattern matching are omitted. However, the techniques to do so are well-known and it should be a routine matter to extend the existing work [Birkedal and Welinder 1994].

A more challenging aspect of Standard ML is coverage of the module language. We believe that our approach can deal with it, mainly because the ML-Kit which is based on the region calculus can deal with it, too. Given that, it should be a matter of extending the constraint analysis. There has been some earlier work in that direction, although in the context of a simply-typed language [Thiemann 1999b].

Another important aspect of ML are computational effects like exceptions and destructive updates. Regions and effects have been used previously to specialize programs with computational effects, *e.g.*, side effects due to destructive updates on reference cells [Thiemann 1997a]. That work deals with a simply-typed language. Clearly it would have to be extended to deal with polymorphism. Exceptions can be covered in a similar way [Thiemann 1998]. Fitting that work into the present framework could be accomplished via the extension of the storeless small-step semantics for λ^{region} from this paper with an explicit store [Calcagno et al. 2002]. This operational extension caters for ML-style reference cells and may provide a basis for specialization-time reductions of store operations. However, such additions are not straightforward [Thiemann and Dussart 1997] and complicate the soundness proof as well as the implementation of the specializer considerably.

8. CONCLUSION

The present work provides the semantic foundations for correct offline partial evaluation for the programming language ML. Building upon the region analysis of Tofte and Talpin, we have presented a general approach for polymorphic specialization for functional programming languages with an ML-style type discipline. The framework modularizes the problem of the binding-time analysis by separating the flow analysis (in our approach performed by the region inference) from the binding-time annotation phase (here the 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. Further work is required to transform the theory presented in this work into a practical tool.

Acknowledgement

We are indebted to the anonymous reviewers for the extensive and thoughtful comments, which lead to significant improvements in the presentation of this work.

REFERENCES

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. 1999. A core calculus of dependency. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, A. Aiken, Ed. ACM Press, San Antonio, Texas, USA, 147–160.

Banerjee, A., Heintze, N., and Riecke, J. 1999. Region analysis and the polymorphic lambda calculus. In *14th Annual Symposium on Logic in Computer Science, LICS'99*. IEEE Computer Society Press, Trento, Italy, 88–97.

Birkedal, L. and Tofte, M. 2001. A constraint-based region inference algorithm. *Theoretical Computer Science 58*, 299–392.

Birkedal, L. and Welinder, M. 1993. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen.

Birkedal, L. and Welinder, M. 1994. Binding-time analysis for Standard-ML. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, P. Sestoft and H. Søndergaard, Eds. University of Melbourne, Australia, Orlando, Fla., 61–71. Technical Report 94/9, Department of Computer Science.

Bondorf, A. and Jørgensen, J. 1993. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming 3,* 3 (July), 315–346.

Bulyonkov, M. 1993. Extracting polyvariant binding time analysis from polyvariant specializer. See Schmidt [1993], 59–65.

Calcagno, C. 2001. Stratified operational semantics for safety and correctness of the region calculus. See Nielson [2001], 155–165.

Calcagno, C., Helsen, S., and Thiemann, P. 2002. Syntactic type soundness results for the region calculus. *Information and Computation 173,* 2, 199–221.

Consel, C. 1989. Analyse de programmes, evaluation partielle et génération de compilateurs. Ph.D. thesis, Université de Paris VI, Paris, France.

Consel, C. 1990. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. ACM Press, Nice, France, 264–272.

Consel, C. 1993a. Polyvariant binding-time analysis for applicative languages. See Schmidt [1993], 66–77.

Consel, C. 1993b. A tour of Schism. See Schmidt [1993], 134–154.

Consel, C. and Danvy, O. 1991. For a better support of static data flow. See Hughes [1991], 496–519.

Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Charleston, South Carolina, 493–501.

Consel, C., Jouvelot, P., and Ørbæk, P. 1994. Separate polyvariant binding-time reconstruction. Tech. Rep. CRI-A/261, Ecole des Mines, Paris. Oct.

Danvy, O., Malmkjær, K., and Palsberg, J. 1995. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation 8,* 3 (July), 209–227.

Danvy, O., Malmkjær, K., and Palsberg, J. 1996. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems 18,* 6 (Nov.), 730–751.

Dussart, D., Heldal, R., and Hughes, J. 1997. Module-sensitive program specialisation. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation*. ACM Press, Las Vegas, NV, USA, 206–214.

Dussart, D., Henglein, F., and Mossin, C. 1995. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proceedings of the 1995 International Static Analysis Symposium*, A. Mycroft, Ed. Number 983 in Lecture Notes in Computer Science. Springer-Verlag, Glasgow, Scotland, 118–136.

Gengler, M. and Rytz, B. 1992. A polyvariant binding time analysis handling partially known values. In *Proceedings of the 2nd Workshop on Static Analysis*. Bigre Journal, vol. 81–82. IRISA, Rennes, France, 322–330.

Gifford, D. and Lucassen, J. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. 28–38.

Glenstrup, A. J. and Jones, N. D. 1996. Bta algorithms to ensure termination of offline partial evaluation. See PSI '96 [1996], 273–284.

Glynn, K., Stuckey, P., Sulzmann, M., and Søndergaard, H. 2001. Boolean constraints for binding-time analysis. In *Programs as Data Objects II*. Number 2053 in Lecture Notes in Computer Science. Springer-Verlag, Aarhus, Denmark, 39–62.

Harper, R. 1994. A simplified account of polymorphic references. *Information Processing Letters 51,* 4 (Aug.), 201–206. See also note [Harper 1996].

Harper, R. 1996. A note on: "A simplified account of polymorphic references". *Information Processing Letters 57,* 1 (Jan.), 15–16. See also [Harper 1994].

Hatcliff, J. and Danvy, O. 1997. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science 7,* 5, 507–542.

Heldal, R. 2001. The treatment of polymorphism and modules in a partial evaluator. Ph.D. thesis, Chalmers University of Technology.

Heldal, R. and Hughes, J. 2000. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science 248*, 99–145.

Heldal, R. and Hughes, J. 2001. 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. Springer-Verlag, Novosibirsk, Russia, 191–204.

Helsen, S. 2002. Region-based program specialization – an operational approach to polymorphic offline partial evaluation for ml-like languages. Ph.D. thesis, Unversität Freiburg, Germany. Available from `http://www.freidok.uni-freiburg.de/volltexte/538/`.

Helsen, S. 2004. Bisimilarity for the region calculus. *Higher-Order and Symbolic Computation 17,* 4. (to appear).

Helsen, S. and Thiemann, P. 2000a. Fragmental specialization. In *Semantics, Applications and Implementation of Program Generation (SAIG'00)*, W. Taha, Ed. Number 1927 in Lecture Notes in Computer Science. Springer-Verlag, Montreal, Canada, 51–71.

Helsen, S. and Thiemann, P. 2000b. Syntactic type soundness for the region calculus. In *ACM Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, A. Jeffrey, Ed. Electronic Notes in Theoretical Computer Science, vol. 41(3). Elsevier Science, Montreal, Canada, 1–20.

Henglein, F. 1991. Efficient type inference for higher-order binding-time analysis. See Hughes [1991], 448–472.

Henglein, F. and Mossin, C. 1994. Polymorphic binding-time analysis. In *Proceedings of European Symposium on Programming*, D. Sannella, Ed. Lecture Notes in Computer Science, vol. 788. Springer-Verlag, 287–301.

Hornof, L. and Noyé, J. 2000. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science 248,* 1–2 (Oct.), 3–27.

Hughes, J., Ed. 1991. *Functional Programming Languages and Computer Architecture*. Number 523 in Lecture Notes in Computer Science. Springer-Verlag, Cambridge, MA.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Jouvelot, P. and Gifford, D. 1991. Algebraic reconstruction of types and effects. In *Proc. 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Orlando, Florida, 303–310.

Lassen, S. 1998. Relational reasoning about contexts. In *Higher Order Operational Techniques in Semantics (HOOTS)*, A. Gordon and A. Pitts, Eds. Publications of the Newton Institute. Cambridge University Press, 91–136.

Launchbury, J. and Holst, C. K. 1991. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*. Glasgow University, Skye, Scotland, 210–218.

Lawall, J. and Thiemann, P. 1997. Sound specialization in the presence of computational effects. In *Proceedings of the Theoretical Aspects of Computer Software*. Number 1281 in Lecture Notes in Computer Science. Springer-Verlag, Sendai, Japan, 165–190.

Lee, C. S. 2002a. Finiteness analysis in polynomial time. In *Static Analysis: 9th International Symposium, SAS 2002*, M. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer, 493–508.

Lee, C. S. 2002b. Program termination analysis in polynomial time. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, D. Batory, C. Consel, and W. Taha, Eds. Lecture Notes in Computer Science, vol. 2487. ACM, Springer, 218–235.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. 2001. The size-change principle for program termination. See Nielson [2001], 81–92.

Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Mogensen, T. 1989. Binding time analysis for polymorphically typed higher order languages. In *TAPSOFT '89*, J. Díaz and F. Orejas, Eds. Number 351,352 in Lecture Notes in Computer Science. Springer-Verlag, Barcelona, Spain, II, 298–312.

Moggi, E. 1991. Notions of computations and monads. *Information and Computation 93*, 55–92.

Morris, J. 1968. Lambda calculus models of programming languages. Ph.D. thesis, MIT Press.

Nielson, F. and Nielson, H. R. 1988. Automatic binding-time analysis for a typed lambda calculus. *Science of Computer Programming 10*, 139–176.

Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press.

Nielson, H. R., Ed. 2001. *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, London, England.

Odersky, M., Sulzmann, M., and Wehr, M. 1999. Type inference with constrained types. *Theory and Practice of Object Systems 5,* 1, 35–55.

Plotkin, G. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science 1*, 125–159.

Plotkin, G. 1981. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark.

PSI '96 1996. *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*. Number 1181 in Lecture Notes in Computer Science. Springer-Verlag, Novosibirsk, Russia.

Rytz, B. and Gengler, M. 1992. A polyvariant binding time analysis. In *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, C. Consel, Ed. Yale University, San Francisco, CA, 21–28. Report YALEU/DCS/RR-909.

Schmidt, D., Ed. 1993. *Proceedings of the 1993 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, Denmark.

Talpin, J.-P. and Jouvelot, P. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming 2,* 3 (July), 245–272.

Thiemann, P. 1996. Implementing memoization for partial evaluation. In *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, H. Kuchen and D. Swierstra, Eds. Number 1140 in Lecture Notes in Computer Science. Springer-Verlag, Aachen, Germany, 198–212.

Thiemann, P. 1997a. Correctness of a region-based binding-time analysis. In *Proceedings of the 1997 Conference on Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science, vol. 6. Carnegie Mellon University, Elsevier Science BV, Pittsburgh, PA, 26. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

Thiemann, P. 1997b. A unified framework for binding-time analysis. In *TAPSOFT '97: Theory and Practice of Software Development*, M. Bidoit and M. Dauchet, Eds. Number 1214 in Lecture Notes in Computer Science. Springer-Verlag, Lille, France, 742–756.

Thiemann, P. 1998. A generic framework for specialization. In *Proc. 7th European Symposium on Programming*, C. Hankin, Ed. Number 1381 in Lecture Notes in Computer Science. Springer-Verlag, Lissabon, Portugal, 267–281.

Thiemann, P. 1999a. Combinators for program generation. *Journal of Functional Programming 9,* 5 (Sept.), 483–525.

Thiemann, P. 1999b. Interpreting specialization in type theory. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, O. Danvy, Ed. San Antonio, Texas, USA, 30–43. BRICS Notes Series NS-99-1.

Thiemann, P. 2000. *The PGG System—User Manual*. Universität Freiburg, Freiburg, Germany. Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

Thiemann, P. 2001. Enforcing safety properties using type specialization. In *Proceedings of the 2001 European Symposium on Programming*, D. Sands, Ed. Lecture Notes in Computer Science. Springer-Verlag, Genova, Italy.

Thiemann, P. 2002. A prototype dependency calculus. In *Proc. 11th European Symposium on Programming*, D. L. Metayér, Ed. Number 2305 in Lecture Notes in Computer Science. Springer-Verlag, Grenoble, France, 228–242.

Thiemann, P. and Dussart, D. 1997. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen. Apr.

Tofte, M. and Birkedal, L. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems 20,* 5, 724–767.

Tofte, M. and Birkedal, L. 2000. Unification and polymorphism in region inference. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olensen, T., and Sestoft, P. 2001. *Programming with Regions in the ML Kit, Version 4.0.0*. Available from `http://www.it.edu/research/mlkit/dist/mlkit-4.0.0.pdf`.

Tofte, M. and Talpin, J.-P. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 1994 ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Portland, OR, 188–201.

Tofte, M. and Talpin, J.-P. 1997. Region-based memory management. *Information and Computation 132,* 2, 109–176.

Wright, A. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

A. PROOF OF THE SECOND SUBSTITUTION LEMMA

Suppose

(1) $P = \{\varrho_1 \ldots, \varrho_n\}$ *and* $(P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\mathsf{p}\}) = \emptyset$ *and* $\{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset$ *and* $\sigma = \forall\,\vec{\alpha}\,.\hat{\sigma}$ *and* $\hat{\sigma} = \forall\,\vec{\epsilon}.\forall\,\varrho_1, \ldots, \varrho_n\,.C^{\uparrow P} \Rightarrow \tau$

(2) $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$

(3) $TE + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_{\mathsf{p}} : (\tau, \mathsf{p}), \emptyset$

Then
$TE, C_{\downarrow P} \vdash_c \{f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x . \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_{\mathsf{p}}\ \mathtt{in}\ E_1 \rangle_{\mathsf{p}}\} E_2 : \mu, \varphi$.

PROOF. By induction on the derivation of $TE + \{f \mapsto (\sigma, \mathsf{p})\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$.

Except for free occurrences of f in E_2, all cases are relatively straightforward applications of the induction hypothesis. However, to be able to use the induction hypothesis, we must guarantee that the constraint set in the above judgment always has the form $C_{\downarrow P}$, where P is fixed.

For the type rules where the constraint set does not change, this is trivially true. For the rules where the judgment takes out constraints with respect to the premise, for example *(Reglam)*, it is easy to see that the above requirement is satisfied as well because of Lemma 3.2.

The only type rule where the premise may have a smaller constraint set in its premise is rule *(Regappv)*. In this case, we need to apply the induction hypothesis on a judgment with constraint set C' where we have that $C_{\downarrow P} \rhd S_r(C') \cup \{\mathsf{p}', \mathsf{p}, \mathsf{p}'_1, \ldots, \mathsf{p}'_n\}$. Therefore we have by definition of consistency that $S_r(C') \subseteq C_{\downarrow P}$ and hence $P \cap S_r(C') = \emptyset$. By α-renaming the region variables $\{\varrho_1, \ldots, \varrho_n\}$, we can also assume without loss of generality that $\{\varrho_1, \ldots, \varrho_n\} \cap P = \emptyset$. Therefore, we have that $C' = C'_{\downarrow P}$, which makes the induction hypothesis applicable to the judgment in the premise.

Consider now the base case where we assume that E_2 is of the form f. In that case, the final derivation is as follows:

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

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

$$\underline{P} \cap (fps\,(TE, C) \cup \{\mathsf{p}, \mathsf{p}'\} \cup P \cup Img\,(S_s)) = \emptyset \tag{42}$$

Observe that the type schemes $\underline{\sigma}$ and σ are equal up to α-renaming and similarly for $\underline{\hat{\sigma}}$ and $\hat{\sigma}$.

Abbreviating $A = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_{\mathsf{p}}$, we must show that

$$TE, C_{\downarrow P} \vdash_c \langle \Lambda\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n . \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1) \rangle_{\mathsf{p}}\ [\mathsf{p}'_1, \ldots, \mathsf{p}'_n]\ \mathtt{at}\ \mathsf{p}' : (\tau', \mathsf{p}'), \{\mathsf{p}, \mathsf{p}'\}$$

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

—$TE, C'' \vdash_c \langle \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}'} : (\tau'', \mathsf{p}'), \emptyset$;
—$\underline{P} \cap (\mathit{fps}\,(TE) \cup \{\mathsf{p}, \mathsf{p}'\}) = \emptyset$ (which is 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 $S_s = S_r \circ S'_s$. Since $(C', \tau') \prec \underline{\sigma}$ *via* S_s, the type τ'' satisfies the third requirement.

To show the last requirement, note that also via $(C', \tau') \prec \underline{\sigma}$ *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 judgment of the first requirement, which we can formulate as follows:

$$TE, \chi(C) \vdash_c \langle \lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1\rangle_{\mathsf{p}}\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}'} : (S'_s(\chi(\tau)), \mathsf{p}'), \emptyset \tag{43}$$

To see this, it is first shown that

$$TE, \chi(C) \vdash_c \langle \lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1\rangle_{\mathsf{p}}\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}} : (\chi(\tau), \mathsf{p}), \emptyset \tag{44}$$

Assumption (3) makes that

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, C \vdash_c \langle \lambda\, x.\, E_1\rangle_{\mathsf{p}} : (\tau, \mathsf{p}), \emptyset \tag{45}$$

holds for any μ_1. Applying Lemma 3.7 with the renaming substitution χ, we obtain

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}, \chi(C) \vdash_c \langle \lambda\, x.\, \chi(E_1)\rangle_{\mathsf{p}} : (\chi(\tau), \mathsf{p}), \emptyset$$

Because of rule *(Lamv)*, we can take $\tau = \mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2$ and for some $\varphi'' \subseteq \varphi'$ have the judgment

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \mathsf{p})\} + \{x \mapsto \chi(\mu_1)\}, \chi(C) \vdash_c \chi(E_1) : \chi(\mu_2), \chi(\varphi'') \tag{46}$$

where

$$\chi(C) \vdash_{\mathit{wft}} (\chi(\mu_1) \xrightarrow{\epsilon'.\chi(\varphi')} \chi(\mu_2), \mathsf{p}) \tag{47}$$

Applying Lemma 3.8 to (46) for $\hat{\sigma} \prec \sigma$ and observing that $TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \mathsf{p})\} + \{x \mapsto \chi(\mu_1)\} = TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \mathsf{p})\}$ because $x \neq f$, yields

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\sigma, \mathsf{p})\}, \chi(C) \vdash_c \chi(E_1) : \chi(\mu_2), \chi(\varphi'') \tag{48}$$

Observe now that $\chi(C) = \chi(C)_{\downarrow P}$ since $\mathit{fps}\,(\chi(C)) \cap P = \emptyset$. Hence, from rule *(Letrecv)*, assumptions (1) and (42), and the judgments (45) and (48), we obtain

$$\begin{aligned} TE + \{x \mapsto \chi(\mu_1)\}, \chi(C) \vdash_c\ & \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1\rangle_{\mathsf{p}}\ \mathtt{in}\ \chi(E_1) \\ & : \chi(\mu_2), \chi(\varphi'') \end{aligned} \tag{49}$$

Using rule *(Lamv)* and the judgments (47) and (49), we can derive judgment (44). Now, applying Lemma 3.7 with S'_s to judgment (44) yields

$$S'_s(TE), S'_s(\chi(C)) \vdash_c S'_s(\langle \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}}) : S'_s(\chi(\tau), \mathsf{p}), S'_s(\emptyset) \quad (50)$$

The support of $S'_s = (S_t, S_e, I_r)$ is a subset of $\{\vec{\epsilon}, \vec{\alpha}\}$ since we have the instantiation $(C', \tau') \prec \sigma$ *via* (S_t, S_e, S_r). By assumption (1), the set $\{\vec{\epsilon}, \vec{\alpha}\}$ is disjoint from $fv\,(TE)$. Therefore,

—$S'_s(TE) = TE$;
—$S'_s(\langle \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}}) = \langle \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1)\rangle_{\mathsf{p}}$ due to I_r;
—$S'_s(\chi(C)) = \chi(C)$ due to I_r; and
—$S'_s(\chi(\tau), \mathsf{p}) = (S'_s(\chi(\tau)), \mathsf{p})$.

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

Given (47), we also have $\chi(C) \vdash_{wft} (\chi(\mu_1) \xrightarrow{\epsilon'.\chi(\varphi')} \chi(\mu_2), \mathsf{p}')$ since $\{\mathsf{p} \le \mathsf{p}', \mathsf{p}' \le \mathsf{p}\} \subseteq C_0 \subseteq C_{\downarrow P} \subseteq C$ and therefore, by assumption (1), $\{\mathsf{p} \le \mathsf{p}', \mathsf{p}' \le \mathsf{p}\} \subseteq \chi(C)$. Judgment (43) can now be derived by replacing p for p' in rule *(Lamv)*. This concludes the proof. □

B. 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 the type rules *(Effect)* and *(Constraint)* 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 $\epsilon \notin fev\,(TE, \mu, \varphi)$, 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 *(Constraint)*, then we also have $C \rhd \{\varrho^\bullet \le s\}$. 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 3.1.1, 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 $C' \rhd \{\varrho^\bullet \le s\}$. Applying rule *(Constraint)* proves the case.

If the last rule in the proof of $TE, C \vdash_c E : \mu, \varphi$ is not *(Effect)* or *(Constraint)*, 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 $C \vdash_{wft} (\mathtt{int}, \varrho)$. So, take $C'' = C \cup \{\varrho \le s\}$ and $C' = \widetilde{C''}$. Surely, $C \subseteq C'$ and since $\delta \models C$ and $\delta \models \{\varrho \le s\}$, we also have $\delta \models C'$ by Lemma 3.10. Of course we have $C' \vdash_{wft} (\mathtt{int}, \varrho)$ and hence by *(Constv)* 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 (3) and (4). Analogous.

Case $\delta, \texttt{new}^{\mathsf{s}}\, \varrho.Q \rightsquigarrow \delta, \{\varrho \mapsto \varrho^\bullet\}Q$. We have by assumption $TE, C_{\downarrow\varrho} \vdash_c \texttt{new}\, \varrho.\, Q : \mu, \varphi \setminus \{\varrho\}$ and $\delta \models C_{\downarrow\varrho}$. Hence, from rule *(Reglam-s)*, this implies that $TE, C \vdash_c Q : \mu, \varphi$ with $\varrho \notin \mathit{fps}\,(TE, \mu)$ and $C \rhd \{\varrho \leq \mathsf{s}\}$. Using Lemma 3.7, we obtain that $TE, \{\varrho \mapsto \varrho^\bullet\}C \vdash_c \{\varrho \mapsto \varrho^\bullet\}Q : \mu, \{\varrho \mapsto \varrho^\bullet\}\varphi$. The fact that $C \rhd \{\varrho \leq \mathsf{s}\}$ implies that $\{\varrho \mapsto \varrho^\bullet\}C \rhd \{\varrho^\bullet \leq \mathsf{s}\}$. So from rule *(Constraint)*, we obtain $TE, \{\varrho \mapsto \varrho^\bullet\}C_{\downarrow\varrho^\bullet} \vdash_c \{\varrho \mapsto \varrho^\bullet\}Q : \mu, (\{\varrho \mapsto \varrho^\bullet\}\varphi) \setminus \{\varrho^\bullet\}$. The claim is proven by observing that $\{\varrho \mapsto \varrho^\bullet\}C_{\downarrow\varrho^\bullet} = C_{\downarrow\varrho}$ and $(\{\varrho \mapsto \varrho^\bullet\}\varphi) \setminus \{\varrho^\bullet\} = \varphi \setminus \{\varrho\}$.

Case $\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} \rightsquigarrow \delta, \langle c\rangle_{\varrho_2}$ *if* $\delta(\varrho_2) = \mathsf{s}$. So, we have the typing judgment $TE, C \vdash_c \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} : (\texttt{int}, \varrho_2), \varphi \cup \{\varrho_1, \varrho_2\}$. Because of rule *(Lift)* and rule *(Constv)*, we have that $\varphi = \emptyset$; $C \rhd \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \mathsf{s}\}$ and $C \vdash_{\mathit{wft}} (\texttt{int}, \varrho_1)$. Take C' such that for $C'' = C \cup \{\varrho_1 \leq \mathsf{s}\}$, $C' = \widetilde{C''}$. Now, we obviously have $C' \vdash_{\mathit{wft}} (\texttt{int}, \varrho_1)$ and also $\delta \models C'$ because of $\delta \models C$; $\delta \models \{\varrho_2 \leq \mathsf{s}\}$ and Lemma 3.10. Type rule *(Constv)* makes that $TE, C \vdash_c \langle c\rangle_{\varrho_2} : (\texttt{int}, \varrho_2), \emptyset$ and since $\emptyset \subseteq \{\varrho_1, \varrho_2\}$, the case is proven.

Case $\delta, \langle \lambda\, x.\, E\rangle_\varrho \mathbin{@} Q \rightsquigarrow \delta, \texttt{let}\, x = Q\, \texttt{in}\, E$. By assumption and Lemma 5.1 we have $TE, C \vdash_c \langle \lambda\, x.E\rangle_\varrho \mathbin{@} Q : \mu_2, \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$ 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 *(Lamv)* this makes $TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi''$ too, with $\varphi'' \subseteq \varphi$. Using *(Let)* on the latter two judgments for Q and E, we obtain $TE, C \vdash_c \texttt{let}\, x = Q\, \texttt{in}\, E : \mu_2, \varphi'' \cup \varphi'$, where $\varphi'' \cup \varphi' \subseteq \varphi \cup \varphi' \subseteq \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$.

Case $\delta, \texttt{let}\, x = V\, \texttt{in}\, E \rightsquigarrow \delta, \{x \mapsto V\}E$. Immediate by rule *(Let)* and Lemma 5.3.

Case $\dfrac{\delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E\rangle_\varrho\, [\mathsf{p}'_1, \ldots, \mathsf{p}'_n]\ \texttt{at}\ \varrho_0 \rightsquigarrow \delta, \langle \lambda\, x.\, \{\varrho_1 \mapsto \mathsf{p}'_1, \ldots, \varrho_n \mapsto \mathsf{p}'_n\}E\rangle_{\varrho_0}}{}$. So, rule *(Regappv)* applies:

$$\frac{\begin{array}{c} TE, C' \vdash_c \langle \lambda\, x.\, E\rangle_{\varrho_0} : (\tau, \varrho_0), \emptyset \\ \{\varrho_1, \ldots, \varrho_n\} \cap (\mathit{fps}\,(TE) \cup \{\varrho, \mathsf{p}_0\}) = \emptyset \quad S_r = \{\varrho_1 \mapsto \mathsf{p}'_1, \ldots, \varrho_n \mapsto \mathsf{p}'_n\} \\ \tau' = S_r(\tau) \quad C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \mathsf{p}'_1, \ldots, \varrho_0 \leq \mathsf{p}'_n\} \end{array}}{TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E\rangle_\varrho\, [\mathsf{p}'_1, \ldots, \mathsf{p}'_n]\ \texttt{at}\ \varrho_0 : (\tau', \varrho_0), \{\varrho, \varrho_0\}}$$

By Lemma 3.7, 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 $C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \mathsf{p}'_1, \ldots, \varrho_0 \leq \mathsf{p}'_n\}$, the case is proven by Lemma 3.5.

Case $\dfrac{}{\delta, \texttt{letrec}\, f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\rangle_\rho\, \texttt{in}\, E_2 \rightsquigarrow \delta, \{f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, \texttt{letrec}\, f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\rangle_\rho\, \texttt{in}\, E_1\rangle_\rho\}E_2}$.
The left-hand side can only be typed by rule *(Letrecv)*. This, however, gives exactly the requirements for Lemma 5.4, which on its turn proves the case.

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

Cases for the rules (12) through (15). Analogous to the previous case.

Case $\dfrac{\delta, \mathtt{new}\,\varrho : C_0\,.E \rightsquigarrow}{\delta + \{\varrho \mapsto \mathtt{s}\}, \mathtt{new}^{\mathtt{s}}\,\varrho.E}$ *if* $\varrho \notin \mathit{Dom}\,(\delta)$ *and* $\delta + \{\varrho \mapsto \mathtt{s}\} \models C_0$. First, we observe that $\delta \subseteq \delta + \{\varrho \mapsto \mathtt{s}\}$. By assumption there exists a C such that $C_0 = C^{\uparrow\varrho}$ and $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}\,\varrho : C_0.\,E : \mu, \varphi \setminus \{\varrho\}$ via *(Reglam)*, and $\delta \models C_{\downarrow\varrho}$. Hence, we have $TE, C \vdash_c E : \mu, \varphi$ where $\varrho \notin \mathit{fps}\,(\mu, TE)$. Take $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq \boldsymbol{s}\}$. So, by Lemma 3.5, we have that $TE, C' \vdash_c E : \mu, \varphi$. As $C' \rhd \{\varrho \leq \boldsymbol{s}\}$, we can apply rule *(Reglam-s)* to obtain $TE, {C'}_{\downarrow\varrho} \vdash_c \mathtt{new}\,\varrho.\,E : \mu, \varphi \setminus \{\varrho\}$. 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 $\delta + \{\varrho \mapsto \mathtt{s}\} \models C_0 (= C^{\uparrow\varrho})$, applying Lemma 3.12 yields that $\delta + \{\varrho \mapsto \mathtt{s}\} \models C$. Also, we have $\delta + \{\varrho \mapsto \mathtt{s}\} \models \{\varrho \leq \boldsymbol{s}\}$, so Lemma 3.10 makes that $\delta + \{\varrho \mapsto \mathtt{s}\} \models C'$. The case is now proven as we obviously have $\delta + \{\varrho \mapsto \mathtt{s}\} \models {C'}_{\downarrow\varrho}$ too.

Case $\dfrac{\delta, \mathtt{new}\,\varrho : C\,.E \rightsquigarrow}{\delta + \{\varrho \mapsto \mathtt{d}\}, \mathtt{new}^{\mathtt{d}}\,\varrho.E}$ *if* $\varrho \notin \mathit{Dom}\,(\delta)$ *and* $\delta + \{\varrho \mapsto \mathtt{d}\} \models C$. This is 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* $\delta(\varrho_2) = \mathtt{d}$. Analogous to the case for reducing `lift` with $\delta(\varrho_2) = \mathtt{s}$.

Cases for the rules (18) through (25). These rules are all fairly straightforward by the design of the type rules for residual expressions. Reductions (20), (21) and (24) rely on the design of the type rules *(Letrec-d)*, *(Reglam-d)* and *(Let-d)* respectively. Reduction rule (22) needs Lemma 5.2 and reduction (25) relies on the assumption about the type environment.

Cases for the rules (26) and (27). 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* $\delta(\varrho) = \mathtt{d}$. We have to type the expressions with rule *(Lam)*, which gives us that $TE, C \vdash_c \lambda x.\ E\ \mathtt{at}\ \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \{\varrho\} \cup \varphi$, 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 (28) 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 *(Lam)* to prove the case.

Cases for the rules (29) through (32). All similarly simple appeals to the induction hypothesis.

Case $\delta, \mathtt{lift}\,[p_1, p_2]\,(\mathtt{new}^{\mathtt{d}}\,\varrho.\mathrm{Q}^{\mathtt{s}}) \rightsquigarrow \delta, \mathtt{new}^{\mathtt{d}}\,\varrho.(\mathtt{lift}\,[p_1, p_2]\,\mathrm{Q}^{\mathtt{s}})$ *if* $\varrho \notin \{p_1, p_2\}$. To type the left-hand side, we first use *(Lift)* and then *(Reglam-d)*, giving us the following derivation:

$$\dfrac{\dfrac{\dfrac{TE, C \vdash_c \mathrm{Q}^{\mathtt{s}} : (\mathtt{int}, \mathtt{p}_1), \varphi \quad \varrho \notin \mathit{fps}\,(TE) \cup \{\mathtt{p}_1\} \quad C \rhd \{\boldsymbol{d} \leq \varrho\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathtt{d}}\,\varrho.\mathrm{Q}^{\mathtt{s}} : (\mathtt{int}, \mathtt{p}_1), \varphi \setminus \{\varrho\}} \qquad C_{\downarrow\varrho} \rhd \{\mathtt{p}_1 \leq \mathtt{p}_2\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{lift}\,[\mathtt{p}_1, \mathtt{p}_2]\,(\mathtt{new}^{\mathtt{d}}\,\varrho.\mathrm{Q}^{\mathtt{s}}) : (\mathtt{int}, \mathtt{p}_2), (\varphi \setminus \{\varrho\}) \cup \{\mathtt{p}_1, \mathtt{p}_2\}}$$

The case is now proven by re-ordering: first, we have $TE, C \vdash_c \mathrm{Q}^{\mathtt{s}} : (\mathtt{int}, \mathtt{p}_1), \varphi$. Now, obviously we have $\{\mathtt{p}_1 \leq \mathtt{p}_2\} \subseteq C$ and by Lemma 3.4 therefore also that $C \rhd \{\mathtt{p}_1 \leq \mathtt{p}_2\}$. By rule *(Lift)*, this makes $TE, C \vdash_c \mathtt{lift}\,[\mathtt{p}_1, \mathtt{p}_2]\,\mathrm{Q}^{\mathtt{s}} : (\mathtt{int}, \mathtt{p}_2), \varphi \cup \{\mathtt{p}_1, \mathtt{p}_2\}$. Using the fact that $\varrho \notin \{\mathtt{p}_1, \mathtt{p}_2\}$, we have that $\varrho \notin \mathit{fps}\,(TE) \cup \{\mathtt{p}_2\}$. Since $C \rhd \{\boldsymbol{d} \leq \varrho\}$, we can apply *(Reglam-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathtt{d}}\,\varrho.(\mathtt{lift}\,[\mathtt{p}_1, \mathtt{p}_2]\,\mathrm{Q}^{\mathtt{s}}) :$

$(\mathtt{int}, \mathsf{p}_2), (\varphi \cup \{\mathsf{p}_1, \mathsf{p}_2\}) \setminus \{\varrho\}$. And obviously we have $(\varphi \cup \{\mathsf{p}_1, \mathsf{p}_2\}) \setminus \{\varrho\} \subseteq (\varphi \setminus \{\varrho\}) \cup \{\mathsf{p}_1, \mathsf{p}_2\}$, concluding the case.

Case $\delta, \mathtt{let}\ x = (\mathtt{new}^{\mathsf{d}}\ \varrho.\mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\ \varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E^{\mu})$ *if* $\varrho \notin \mathit{fps}\,(\mu)$. Construct the following (part of a) type derivation for the left-hand side, using *(Let)* and *(Reglam-d)*:

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

First, observe by Lemma 3.1.1, Lemma 3.4 and Lemma 3.5, 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 $\varrho \notin \mathit{fps}\,(TE)$; $C \rhd \{\mathtt{d} \leq \varrho\}$ and also that $\varrho \notin \mathit{fps}\,(\mu)$, we can apply rule *(Reglam-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\ \varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E) : \mu, (\varphi \cup \varphi') \setminus \{\varrho\}$. 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 (35) through (41). Proven analogous to the two previous cases: the derivation tree of the original expression is reordered to obtain a proof for the result expression.

□

C. PROOF OF PROGRESS

Suppose for a λ_2^{region}-Term E^{μ} *that* $TE, C \vdash_c E^{\mu} : \mu, \varphi$. *Additionally, assume that* $\delta \models C$; $\delta_{\mathrm{init}} \subseteq \delta$; *Clean* (φ); $C \vdash_{wft} TE$; *and for all* $(\sigma, \mathsf{p}') \in \mathit{Ran}\,(TE)$, *it holds that* $\delta(\mathsf{p}') = \mathsf{d}$. *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 5.1, we can assume that the derivation does not end with *(Effect)* or *(Constraint)*. Otherwise, we can use exactly the same arguments to apply the induction hypothesis, using Proposition 5.1 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 $TE, C \vdash_c c\ \mathtt{at}\ \mathsf{p} : (\mathtt{int}, \mathsf{p}), \{\mathsf{p}\}$. Since *Clean* $(\{\mathsf{p}\})$, we have $\mathsf{p} \in \mathsf{RegionVar}$. Either $\delta(\mathsf{p}) = \mathsf{s}$ in which case reduction rule (2) applies or, if $\delta(\varrho) = \mathsf{d}$, we can use rule (18).

Case $TE, C \vdash_c \lambda x.\, E'\ \mathtt{at}\ \mathsf{p} : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \mathsf{p}), \{\mathsf{p}\} \cup \varphi$. Whenever $\delta(\mathsf{p}) = \mathsf{s}$ we can reduce with rule (3), otherwise, we have that $\delta(\mathsf{p}) = \mathsf{d}$. From type rule *(Lam)* we also have that

$$TE + \{x \mapsto \mu_1\}, C \vdash_c E' : \mu_2, \varphi \tag{51}$$

where $\varphi \subseteq \varphi'$ and $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho)$. 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 $C \vdash_{wft} TE + \{x \mapsto \mu_1\}$ with for all $(\sigma, \mathsf{p}') \in \mathit{Ran}\,(TE)$ that $\delta(\mathsf{p}') = \mathsf{d}$.

Applying the induction hypothesis, we either have $\delta, E' \rightsquigarrow \delta', E''$, in which case we can reduce with the context rule (28). Alternatively, we have that E' is answer of the form Q. However, we can apply the canonical forms Lemma 5.5 to see that Q must be of the form R. Hence, the expression is reducible with rule (19).

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\ \mathtt{at}\ p\ \mathtt{in}\ E_2 : \mu, \varphi \cup \varphi'$. Either $\delta(\mathsf{p}) = \mathsf{s}$ holds and we apply reduction rule (4). Alternatively, we have $\delta(\mathsf{p}) = \mathsf{d}$. By an analogous argument as in the previous case (using well-formedness of C) we can either reduce the expression with the context rule (29) or with rule (27).

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

$$TE, C \vdash_c E_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \mathsf{p}), \varphi_1 \tag{52}$$

and

$$TE, C \vdash_c E_2 : \mu_1, \varphi_2 \tag{53}$$

Now, apply the induction hypothesis on judgment (52). Either $\delta, E_1 \rightsquigarrow \delta', E_1'$ which means we can use reduction rule (11) 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 judgment (53). If we have that $\delta, E_2 \rightsquigarrow \delta', E_2'$, we can apply reduction rule (12) on $Q_1 \mathbin{@} E_2$. If $E_2 = Q_2$, we make a case distinction for Q_1.

So, suppose first $\delta(\mathsf{p}) = \mathsf{d}$, then because of judgment (52) and the canonical forms lemma, it must be that Q_1 is residual code R_1. However, by Lemma 3.9, it also holds that $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \mathsf{p})$, and since $\delta \models C$, we have that for $\mu_1 = (\tau_1, \mathsf{p}_1)$ it must be that $\delta(\mathsf{p}_1) = \mathsf{d}$. So, by the canonical forms lemma and judgment (53), 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 (23).

Alternatively, suppose $\delta(\mathsf{p}) = \mathsf{s}$. Then by judgment (52) 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 (7) applies. If Q_1^{s} is not a value, we can shift the static expression context with the rules (35), (38) or (41).

Case $TE, C \vdash_c \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 : \mu, \varphi \cup \varphi'$. 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 (14) 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 (8). If Q_1 is a static answer Q_1^{s}, but not a value, then either one of the rules (34), (37) or (40) applies. If Q_1 is residual code R, then we use reduction (26).

Case $TE, C \vdash_c f\ [p_1', \ldots, p_n']\ \mathtt{at}\ p' : (\tau', p'), \{p, p'\}$. This term can be immediately reduced with rule (25).

Case $TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}\ \varrho : C^{\uparrow \varrho}.E : \mu, \varphi \setminus \{\varrho\}$. By Lemma 3.11, we can satisfy the conditions of rule (16).

Case $TE, C \vdash_c \mathtt{lift}\,[p_1, p_2]\, E : (\mathtt{int}, p_2), \varphi \cup \{p_1, p_2\}$. By type rule *(Lift)* we have that $TE, C \vdash_c E : (\mathtt{int}, \mathsf{p}_1), \varphi$ and $C \rhd \{\mathsf{p}_1 \leq \mathsf{p}_2\}$. Since $\mathit{Clean}\,(\varphi \cup \{\mathsf{p}_1, \mathsf{p}_2\})$, we

have that $\{\mathsf{p}_1, \mathsf{p}_2\} \subseteq \mathsf{RegionVar}$. By application of the induction hypothesis we either have that $\delta, E \rightsquigarrow \delta', E'$, in which case we can reduce with rule (13), or we have that E is an answer Q. First, consider the case that $\delta(\mathsf{p}_1) = \mathsf{d}$. Then obviously also $\delta(\mathsf{p}_2) = \mathsf{d}$, implying by the canonical forms lemma that Q is residual code R. In this case, we can reduce with rule (22).

However, if $\delta(\mathsf{p}_1) = \mathsf{s}$, 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 (33), (36) or (39) are applicable. On the other hand, if Q^{s} is a value V, then by the canonical forms lemma it must be that $V = \langle c \rangle_{\mathsf{p}_1}$. If now $\delta(\mathsf{p}_2) = \mathsf{s}$, we reduce with rule (6), otherwise, if $\delta(\mathsf{p}_2) = \mathsf{d}$, we reduce with rule (17).

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

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

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_p\ \mathtt{in}\ E_2 : \mu, \varphi$. We reduce this term with reduction rule (10).

Case $TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E \rangle_p\ [p'_1, \ldots, p'_n]\ \mathtt{at}\ p' : (\tau', p'), \{p, p'\}$.
Since *Clean* $(\{\mathsf{p}, \mathsf{p}'\})$ we have that $\{\mathsf{p}, \mathsf{p}'\} \subseteq \mathsf{RegionVar}$. Hence, we reduce the expression with rule (9).

Case $TE, C \vdash_c \mathtt{new}^{\mathsf{s}}\ \varrho.E : \mu, \varphi \setminus \{\varrho\}$. Since *Clean* $(\varphi \setminus \{\varrho\})$, we obviously have *Clean* (φ). Take a region annotation $\delta' = \delta + \{\varrho \mapsto \mathsf{s}\}$ where $\varrho \notin (\mathit{Dom}\,(\delta) \cup \mathit{fps}\,(C))$ and $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq \boldsymbol{s}\}$. It is given that $\delta \models C$, hence also $\delta' \models C$. Obviously we have $\delta' \models \{\varrho \leq \boldsymbol{s}\}$ too, so by Lemma 3.10 this implies that $\delta' \models C'$. Since $C = C'_{\downarrow \varrho}$, type rule *(Reglam-s)* gives us that $TE, C' \vdash_c E : \mu, \varphi$ with $\varrho \notin \mathit{fps}\,(TE, \mu)$. 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 (15) or if E is an answer, we can use rule (5).

Case $TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\ \varrho.E : \mu, \varphi \setminus \{\varrho\}$. With almost identical arguments (using the constraint $\boldsymbol{d} \leq \varrho$ instead of $\varrho \leq \boldsymbol{s}$) 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 (21). Finally, if E can be further reduced, we can use transition rule (30) 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 $TE, C \vdash_c R : (\tau', \mathsf{p}'), \varphi'$ and $TE + \{x \mapsto \mu'\}, C \vdash_c E' : \mu, \varphi$. By Lemma 5.2 we have that $\delta(\mathsf{p}') = \mathsf{d}$ and hence, we can apply the induction hypothesis on E'. So, either we have $\delta, E' \rightsquigarrow \delta', E''$ and reduction rule (31) applies. Whenever E' is an answer Q, we consider two cases. If Q is residual code R', we can reduce with rule (24). Otherwise, if Q is a static answer Q^{s}, then the whole expression is a static answer as well.

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, R\ \mathtt{at}\ p\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'$.
Analogous to the previous case using rules (20) and (32).

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 immediate, since residual terms are answers.

□