Electronic Notes in Theoretical Computer Science 6 (1997)
URL: `http://www.elsevier.nl/locate/entcs/volume6.html` 25 pages

Correctness of a Region-Based Binding-Time Analysis

Peter J. Thiemann [1]

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
D-72076 Tübingen, Germany.

Abstract

A binding-time analysis is the first pass of an offline partial evaluator. It determines which parts of a program may be executed at specialization time. Region-based binding-time analysis applies to higher-order programming languages with first-class references. The consideration of effects in the determination of binding time properties makes it possible to have a partial evaluator perform assignments at specialization time.

We present such a region-based binding-time analysis and prove its correctness with respect to a continuation-style semantics for an annotated call-by-value lambda calculus with ML-style references. We provide a relative correctness proof that relies on the correctness of region inference and on the correctness of a binding-time analysis for an applied lambda calculus. The main tool in the proof is a translation from terms with explicit region annotations to an extended continuation-passing store-passing style. The analysis is monovariant/monomorphic, however, essentially the same techniques can be used to derive a polyvariant/polymorphic analysis.

Key words: partial evaluation, state, continuations, region inference

1 Introduction

Offline partial evaluation is a successful specialization technique [16]. Binding-time analysis (BTA) is a necessary prerequisite for offline partial evaluation. It annotates all program points of a program as either *static* (executable at specialization time) or *dynamic* advising the specializer to generate code for the respective program point. BTA is driven by a binding-time specification of the input parameters of the program. The specializer proper is just an interpreter of annotated programs.

A BTA is *correct* if the specializer need not perform checks whether some value is data (static) or code (dynamic): the BTA guarantees that the spe-

[1] Email: thiemann@informatik.uni-tuebingen.de

cializer never confuses them. For example, it never builds residual code which contains a static function. This property is quite similar to type safety. The correctness of BTAs for functional languages has already been considered in some depth [12–14, 23, 24, 33]. With the advent of offline partial evaluators for functional languages with state the correctness issue of BTA comes up again. The novelty of these partial evaluators [11] is their ability to perform operations on first-class references at specialization time. This facility greatly enhances the power of partial evaluation, which now applies to programs in message-passing style, unification with references, and interpreters that implement laziness using updatable closures.

Type information alone is not enough to specify a BTA for such a partial evaluator. We start from work on region and effect inference [30, 32] and show how to internalize the results of region inference using a translation to extended continuation-passing store-passing style (ECPSPS) [27,29]. The idea here is to structure the store according to the regions that are active at a given point of the program.

The proposed analysis starts with the output of a region inference algorithm: a program with explicit region annotations and its region-inference derivation. Similar to type-based BTA regions, types, and expressions get binding-time annotations. The well-known well-formedness criteria for binding-time annotated types [9,31] are refined to take into account the annotations of regions. The RBTA (region-based BTA) computes an annotation that fulfills the extended well-formedness criteria. Furthermore the annotation is minimal in the sense that we cannot improve the annotation without breaking the well-formedness criterion. In practice, the well-formedness criterion applied to a region-inference derivation gives rise to a set of constraints on binding-time annotations. Each solution of this set of constraints corresponds to an annotation that fulfills the well-formedness criterion. The RBTA computes the minimal solution, hence it constructs the minimal annotation.

The underlying idea of the correctness proof for the RBTA is the reduction of the problem to a BTA problem for the lambda calculus, which is well-studied [12]. More precisely, there are two steps. First, we modify our translation to ECPSPS so that it respects binding-time annotations. The result is a translation $\mathcal{ET}\,[\![\,]\!]$ from RBTA-annotated terms to annotated lambda terms. By construction, $|\mathcal{ET}\,[\![E]\!]\,|$ [2] has the same semantics as the original term $|E|$. Second, the resulting two-level terms $\mathcal{ET}\,[\![E]\!]$ are well-annotated according to a BTA for the lambda calculus. Hence, the translation preserves the semantics and the well-annotatedness which concludes the proof.

In this work, we concentrate on the correctness proof for a monovariant RBTA. Neither do we cover the specification and implementation of the specializer, nor polyvariant program-point specialization. These issues are discussed elsewhere [11].

[2] $|E|$ denotes the underlying standard term of an annotated term E, i.e., E with all annotations removed.

$$
\begin{array}{rlll}
e & ::= & x & \text{value identifier} \\
 & | & e@e & \text{application} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & \mathbf{rec}\ f(x).e & \text{recursive function definition} \\
 & | & \mathbf{new}\ e & \text{initialization} \\
 & | & \mathbf{get}\ e & \text{dereference} \\
 & | & \mathbf{set}\ e\ e & \text{assignment}
\end{array}
$$

Fig. 1. Syntax of Source Language

2 Example

An informal example illustrates our approach. Consider the source term (in a slightly extended language with respect to the one treated in the following technical development)

$$\lambda y.\mathbf{let}\ x = \mathbf{new}\ 5\ \mathbf{in}\ (\mathbf{get}\ x) + y$$

where **new** 5 allocates a new cell initialized to 5 and returns a reference to it and **get** x retrieves the contents of the reference bound to x.

The region translation applied to this term yields

$$\lambda y.\mathbf{letregion}\ \rho\ \mathbf{in}\ \mathbf{let}\ x = (\mathbf{new}\ 5\ \mathbf{at}\ \rho)\ \mathbf{in}\ (\mathbf{get}\ x\ \mathbf{at}\ \rho) + y$$

The region-annotated term reveals that the lifetime of the reference bound to x is confined to the abstraction. Therefore, the binding time of the reference is independent of the binding time of that abstraction.

BTA applied to this term annotates the abstraction and the addition as dynamic (denoted by a superscript ^{d}), while the remaining operations can be performed at specialization time (denoted by superscript ^{s}). The region ρ is also assigned a binding time which applies to all references that are stored in this region. In the example, ρ is static because its scope does not interfere with a dynamic abstraction.

$$\lambda^d y.\mathbf{letregion}^s\ \rho\ \mathbf{in}\ \mathbf{let}^s\ x = (\mathbf{new}^s\ 5\ \mathbf{at}\ \rho)\ \mathbf{in}\ \mathbf{lift}\ (\mathbf{get}^s\ x\ \mathbf{at}\ \rho) +^d y$$

Other BTAs for languages with comparable features [3] annotate all expressions as dynamic.

3 Region Inference

3.1 Source Language

Figure 1 shows the grammar of source language expressions $e \in \Lambda$ (cf. [30]). The dynamic semantics is identical to the one used in work on region inference [30], i.e., a call-by-value lambda calculus with first-class references in the fashion of Core-ML. We adopt the standard conventions that application associates to the left and that the scope of lambda and rec abstractions extends

$$
\begin{array}{rll}
E ::= & x & \text{value identifier} \\
\mid & E@E & \text{application} \\
\mid & \lambda x.E & \text{abstraction} \\
\mid & \mathbf{rec}\ f(x).E & \text{recursive function definition} \\
\mid & \mathbf{letregion}\ \rho\ \mathbf{in}\ E & \text{region binding} \\
\mid & \mathbf{new}\ E\ \mathbf{at}\ \rho & \text{initialization} \\
\mid & \mathbf{get}\ E\ \mathbf{at}\ \rho & \text{dereference} \\
\mid & \mathbf{set}\ E\ E\ \mathbf{at}\ \rho & \text{assignment}
\end{array}
$$

Fig. 2. Syntax of Region Language

as far to the right as possible. We disambiguate parsing by parentheses where necessary. We also abbreviate lambda abstractions like $\lambda x.\lambda y.e$ to $\lambda xy.e$.

3.2 Region Language

Unfortunately Talpin and Jouvelot [30] do not define a language which makes region allocation and deallocation explicit. Such a language is a convenient intermediate step in our translation. We define an explicit region language Λ_r (see Fig. 2) which extends Λ by explicit region binding and region annotations for the operations on references. It is adapted from the presentation of Tofte and Talpin [32], which addresses implicit storage allocation but not first-class references. Let $\rho \in$ **RegVars** range over a denumerable infinite set of *region variables*. **letregion** ρ **in** E binds ρ to a newly allocated empty region of memory, **new** E **at** ρ allocates a new cell in region ρ and returns a reference to it, **get** E **at** ρ dereferences a reference from region ρ, and **set** E E **at** ρ assigns to a reference in region ρ.

$\mathbf{fv}(E)$ denotes the set of free variables of E. We adopt the standard notion of free variables in E for region variables ($\mathbf{frv}(E)$), too. Bound region variables may be renamed so as to avoid name clashes, similar to α-conversion in the lambda calculus [2].

We write **letregion** $\overline{\rho}$ **in** E for **letregion** ρ_1 **in** …**letregion** ρ_n **in** E.

3.3 Translation

Source expressions and region expressions are connected via a translation that is guided by types and effects. Effect systems usually distinguish between initialization, dereference, and assignment effects. For simplicity we rather treat all of them as access effects. Hence, an effect is described by the set of regions that it may affect. We also omit type, region, and effect polymorphism since we want to concentrate on the essentials, first, and since it does not lead

$$(var)\quad \Gamma\{x:\tau\} \vdash x \leadsto x : \tau, \emptyset$$

$$(app)\ \frac{\Gamma \vdash e_1 \leadsto E_1 : \tau_2 \xrightarrow{\epsilon} \tau_1, \epsilon_1 \quad \Gamma \vdash e_2 \leadsto E_2 : \tau_2, \epsilon_2}{\Gamma \vdash e_1@e_2 \leadsto E_1@E_2 : \tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

$$(abs)\ \frac{\Gamma\{x:\tau_2\} \vdash e \leadsto E : \tau_1, \epsilon}{\Gamma \vdash \lambda x.e \leadsto \lambda x.E : \tau_2 \xrightarrow{\epsilon} \tau_1, \emptyset}$$

$$(rec)\ \frac{\Gamma\{f : \tau_2 \xrightarrow{\epsilon} \tau_1, x : \tau_2\} \vdash e \leadsto E : \tau_1, \epsilon}{\Gamma \vdash \mathbf{rec}\ f(x).e \leadsto \mathbf{rec}\ f(x).E : \tau_2 \xrightarrow{\epsilon} \tau_1, \emptyset}$$

$$(new)\ \frac{\Gamma \vdash e \leadsto E : \tau, \epsilon}{\Gamma \vdash \mathbf{new}\ e \leadsto \mathbf{new}\ E\ \mathbf{at}\ \rho : \mathbf{ref}_\rho\ \tau, \epsilon \cup \{\rho\}}$$

$$(get)\ \frac{\Gamma \vdash e \leadsto E : \mathbf{ref}_\rho\ \tau, \epsilon}{\Gamma \vdash \mathbf{get}\ e \leadsto \mathbf{get}\ E\ \mathbf{at}\ \rho : \tau, \epsilon \cup \{\rho\}}$$

$$(set)\ \frac{\Gamma \vdash e_1 \leadsto E_1 : \mathbf{ref}_\rho\ \tau, \epsilon_1 \quad \Gamma \vdash e_2 \leadsto E_2 : \tau, \epsilon_2}{\Gamma \vdash \mathbf{set}\ e_1\ e_2 \leadsto \mathbf{set}\ E_1\ E_2\ \mathbf{at}\ \rho : \mathbf{unit}, \epsilon_1 \cup \epsilon_2 \cup \{\rho\}}$$

$$(does)\ \frac{\Gamma \vdash e \leadsto E : \tau, \epsilon \quad \epsilon \subseteq \epsilon'}{\Gamma \vdash e \leadsto E : \tau, \epsilon'}$$

$$(mask)\ \frac{\Gamma \vdash e \leadsto E : \tau, \epsilon \cup \{\overline{\rho}\} \quad (\mathbf{Obs}(\Gamma) \cup \mathbf{Obs}(\tau)) \cap \{\overline{\rho}\} = \emptyset}{\Gamma \vdash e \leadsto \mathbf{letregion}\ \overline{\rho}\ \mathbf{in}\ E : \tau, \epsilon}$$

Fig. 3. The Region Translation

to new problems, in principle. First, we fix a type language.

$$\begin{array}{rcll} \epsilon & \subseteq & \mathbf{RegVars} & \text{effects} \\ \tau & ::= & \mathbf{unit} \mid \mathbf{ref}_\rho\ \tau \mid \tau \xrightarrow{\epsilon} \tau & \text{types} \end{array}$$

Similar to ML, **unit** is a one-element type with single element (), $\mathbf{ref}_\rho\ \tau$ is the type of references to values of type τ allocated in region ρ, and $\tau_1 \xrightarrow{\epsilon} \tau_2$ is the type of functions that map values of type τ_1 to values of type τ_2 and have a latent effect described by ϵ.

The translation rules are not new to this paper (cf. [30,32]). They define a judgement of the form $\Gamma \vdash e \leadsto E : \tau, \epsilon$ which means "under type assumptions Γ source expression e translates into region expression E which has type τ and effect ϵ." As usual, a type assumption is a finite set of pairs $x : \tau$ associating a variable uniquely with a type. Figure 3 shows the translation rules.

The typing part of the rules mirrors the simply-typed lambda calculus. An access to a variable *(var)* does not have an effect, executing an application *(app)* has the effects of the function part, of the argument part, and the latent effects of the function, an abstraction *(abs)* makes the effect of its body latent while it does not have an effect itself. The rule *(rec)* is similar. The *(new)*, *(get)*, and *(set)* rules just accumulate the effects of their arguments and add the effect of variable access to the reference. The subeffecting rule (*(does)*) can be used whenever there is an effect mismatch in the rule (*(set)*) or in the application rule (*(app)*). The (*(mask)*) rule implements *(effect) masking* [18]. This rule makes it possible to delimit regions of memory and their lifetimes by masking out all effects on regions that are not observable anymore. It only retains those effects ϵ that affect regions mentioned in the assumptions Γ or in the result type τ. The set of regions $\{\overline{\rho}\}$ captures exactly those regions that are only used inside E. These regions $\overline{\rho}$ are abstracted from E. The auxiliary function **Obs** corresponds to the function Observe of Tofte and Talpin [32]. It extracts the observable regions from types and type assumptions:

$$\begin{aligned}
\mathbf{Obs}(\mathbf{unit}) &= \emptyset \\
\mathbf{Obs}(\mathbf{ref}_\rho\ \tau) &= \{\rho\} \cup \mathbf{Obs}(\tau) \\
\mathbf{Obs}(\tau_1 \xrightarrow{\epsilon} \tau_2) &= \mathbf{EffRegions}(\epsilon) \cup \mathbf{Obs}(\tau_2) \\
\mathbf{Obs}(\Gamma\{x : \tau\}) &= \mathbf{Obs}(\Gamma) \cup \mathbf{Obs}(\tau) \\
\mathbf{Obs}(\{\}) &= \emptyset
\end{aligned}$$

The function $\mathbf{EffRegions}(\epsilon)$ extracts the underlying set of region variables from effect ϵ; here $\mathbf{EffRegions}(\epsilon) = \epsilon$.

The result of a good region inference algorithm is a Λ_r expression that "makes all regions as distinct and local as possible" [32].

4 Extended Continuation-Passing Store-Passing Style

Now that we have made region information explicit, we can take the next step. Similar to the way the continuation-passing style translation [26] internalizes the evaluation order, a store-passing style translation translates imperative operations to explicit operations on a concrete representation of the store in an augmented lambda calculus [28]. Since we are also interested in exploiting the beneficial impact of continuation-passing style on the binding-time properties [4], we develop a translation to continuation-passing store-passing style, similar to that used in the definitional interpreters of Reynolds [27]. Our translation differs from these approaches [27,28] in that we do not use a monolithic store, but rather a structured store, where the structure is determined by region and effect information. This is important since we are interested in a typed translation that can be extended to a translation of (binding-time) annotated expressions and annotated types.

4.1 Target Language

The source language of the translation is the region language Λ_r. The target language Λ_{sto} of the translation is a simply-typed lambda calculus augmented with recursive definitions, finite products, and primitives that manipulate heaps:

$$
\begin{array}{lll}
t ::= x \mid t@t \mid \lambda x.t \mid \textbf{rec}\ f(x).t & & \\
\quad \mid (t,\ldots,t) \mid \pi_{(n,i)}(t) & & \text{product formation and selection} \\
\quad \mid \bullet & & \text{an empty heap} \\
\quad \mid \textbf{fnew}\ t\ t & & \text{allocate new heap cell} \\
\quad \mid \textbf{fget}\ t\ t & & \text{heap lookup} \\
\quad \mid \textbf{fupd}\ t\ t\ t & & \text{heap update}
\end{array}
$$

The tuple selector $\pi_{(n,i)}$ selects the ith component of an n-tuple. The nullary product construction $()$ serves as the single value of type **unit**.

The type language is somewhat richer than that of Λ and includes products, heaps, and heap locations.

$$\mu ::= \textbf{unit} \mid \textbf{loc}\ \mu \mid \textbf{heap}\ \mu \mid \mu \times \ldots \times \mu \mid \mu \rightarrow \mu$$

The typing rules for the lambda calculus part with products are the standard ones and we do not repeat them here. However, the rules dealing with heaps are not standard. We assume an infinite type-indexed family of the operations on heaps with the following typing rules.

$$\Gamma \vdash \bullet : \textbf{heap}\ \mu$$

$$\frac{\Gamma \vdash t_1 : \textbf{heap}\ \mu \qquad \Gamma \vdash t_2 : \mu}{\Gamma \vdash \textbf{fnew}\ t_1\ t_2 : (\textbf{loc}\ \mu \times \textbf{heap}\ \mu \rightarrow \mu') \rightarrow \mu'}$$

$$\frac{\Gamma \vdash t_1 : \textbf{heap}\ \mu \qquad \Gamma \vdash t_2 : \textbf{loc}\ \mu}{\Gamma \vdash \textbf{fget}\ t_1\ t_2 : (\mu \rightarrow \mu') \rightarrow \mu'}$$

$$\frac{\Gamma \vdash t_1 : \textbf{heap}\ \mu \qquad \Gamma \vdash t_2 : \textbf{loc}\ \mu \qquad \Gamma \vdash t_3 : \mu}{\Gamma \vdash \textbf{fupd}\ t_1\ t_2\ t_3 : (\textbf{heap}\ \mu \rightarrow \mu') \rightarrow \mu'}$$

Explanation: The operation **fnew** t_1 t_2 allocates a new cell in heap t_1 and initializes it with the value of t_2. It passes the address (reference, location) of the new cell paired with the new heap to its continuation. The operation **fget** t_1 t_2 dereferences heap t_1 at location t_2. It passes the contents of the referenced cell to its continuation. The operation **fupd** t_1 t_2 t_3 updates the heap t_1 at location t_2 with the value of t_3. It passes the updated heap to its continuation.

Lower level definitions of these operations may be found in standard texts on denotational semantics [21,29]. For example, they can be easily defined in terms of Λ_{sto} extended with numbers.

We will make use of a pattern matching notation for abstractions, e.g. $\lambda(x, y).t$, in order simplify the decomposition of pairs that are passed as arguments. We omit the explicit application @.

4.2 Translation

For the purpose of this section we assume that no region variable has more than one binding occurrence in a region expression. Second, we assume a fixed total linear order on the set **RegVars** of region variables. Third, we assume that the region inference algorithm is fine enough that for each region ρ there is exactly one type τ such that there are references of type $\mathbf{ref}_\rho\ \tau$. This is a reasonable assumption since we can construct a region derivation simply by taking a standard derivation for simple types and enclose the resulting expression by one **letregion** ρ_τ **in** ... for each type of value that is stored in or retrieved from a reference.

4.2.1 Types

The idea of the translation to extended continuation passing store passing style is to make store (region) passing explicit and translate a Λ_r-expression of type τ and effect ϵ into a Λ_{sto}-expression of type $\mathcal{ET}\ [\![\tau, \epsilon]\!]$.

$$\begin{aligned}
\mathcal{ET}\ [\![\tau, \epsilon]\!] &= \mathbf{Cont}(\tau, \epsilon) \to \mathbf{Store}(\epsilon) \to \mathbf{Answer} \\
\mathbf{Cont}(\tau, \epsilon) &= \mathcal{EV}\ [\![\tau]\!] \to \mathbf{Store}(\epsilon) \to \mathbf{Answer} \\
\mathbf{Store}(\{\rho_1, \ldots, \rho_m\}) &= \mathbf{heap}\ \mathcal{EV}\ [\![\tau(\rho_1)]\!] \times \ldots \times \mathbf{heap}\ \mathcal{EV}\ [\![\tau(\rho_m)]\!] \\
\mathcal{EV}\ [\![\mathbf{unit}]\!] &= \mathbf{unit} \\
\mathcal{EV}\ [\![\mathbf{ref}_\rho\ \tau]\!] &= \mathbf{loc}\ \mathcal{EV}\ [\![\tau]\!] \\
\mathcal{EV}\ [\![\tau_1 \xrightarrow{\epsilon} \tau_2]\!] &= \mathcal{EV}\ [\![\tau_1]\!] \to \mathcal{ET}\ [\![\tau_2, \epsilon]\!] \\
\mathcal{EV}\ [\![\{x : \tau, \ldots\}]\!] &= \{x : \mathcal{EV}\ [\![\tau]\!], \ldots\}
\end{aligned}$$

Answer is the type of the final answer. $\mathbf{Cont}(\tau, \epsilon)$ is the type of continuations that accept a value of type τ and propagate a store whose structure is prescribed by ϵ. The type $\mathbf{Store}(\epsilon)$ is the product of the types of the heaps associated with the regions mentioned in the effect ϵ. That is, each region (variable) is mapped to a separate heap. Remember that we assume a fixed assignment from region variables to types, which we indicate by writing $\tau(\rho_i)$, and a fixed linear order on region variables. Whenever we refer to regions (or effects) we list them in this particular order. Finally, $\mathcal{EV}\ [\![]\!]$ is the translation of types for values and types in assumptions.

The difference to standard store-passing style lies in the fact that the structure of the store is not fixed but rather corresponds to the set of currently accessed regions as expressed in $\mathbf{Store}(\epsilon)$.

4.2.2 Expressions

For the definition of the translation of expressions it is not sufficient to only take type and effect information into account. Rather the translation maps a whole derivation tree of a region expression into an expression in Λ_{sto}. For readability, we omit the translation part of the region inference system, i.e., we erase the connection with the source language and consider judgements of the form $\Gamma \vdash E : \tau, \epsilon$. Further, we abbreviate derivation trees ending in a judgement of the above form by $\Delta\Big(\Gamma \vdash E : \tau, \epsilon\Big)$.

The simplest case of the translation is the translation of variables.

$$\mathcal{ET}\,[\![\Gamma \vdash x : \tau, \emptyset]\!] = \lambda k\sigma.kx\sigma$$

The translation of an abstraction and a recursive definition is straightforward.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(\Gamma\{x : \tau_2\} \vdash E : \tau_1, \epsilon\Big)}{\Gamma \vdash \lambda x.E : \tau_2 \xrightarrow{\epsilon} \tau_1, \emptyset}\right]\!\!\right] = \lambda k\sigma.k(\lambda x.\mathcal{ET}\,[\![\Delta]\!])\sigma$$

The translation of recursive definitions does not require masking.

$$\begin{aligned}&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(\Gamma\{f : \tau_2 \xrightarrow{\epsilon} \tau_1, x : \tau_2\} \vdash E : \tau_1, \epsilon\Big)}{\Gamma \vdash \mathbf{rec}\ f(x).E : \tau_2 \xrightarrow{\epsilon} \tau_1, \emptyset}\right]\!\!\right]\\ &= \lambda k\sigma.k(\mathbf{rec}\ f(x).\mathcal{ET}\,[\![\Delta]\!])\sigma\end{aligned}$$

In all three cases, the outer store σ has type $\mathbf{Store}(\emptyset) \cong \mathbf{unit}$ as it corresponds to the empty effect. However, the inner store that is expected by the expression $\mathcal{ET}\,[\![\Delta]\!]$ in the abstraction has its type $\mathbf{Store}(\epsilon)$ determined by the latent effect of the function.

The first difficulty arises with the translation of applications. The region inference rule for application states:

$$\frac{\Gamma \vdash E_1 : \tau_2 \xrightarrow{\epsilon} \tau_1, \epsilon_1 \qquad \Gamma \vdash E_2 : \tau_2, \epsilon_2}{\Gamma \vdash E_1@E_2 : \tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

The problem here is that each single phase in the application (evaluation of E_1, evaluation of E_2, and evaluation of the body of the called function) affects only a part of the store, while the full store corresponding to the effect $\epsilon \cup \epsilon_1 \cup \epsilon_2$ must be passed to the translation of the application. Consequently, we need a construction that masks the unneeded parts of the store while passing the needed parts to a computation. The following construction $\mathbf{MASK}(\epsilon, \epsilon')t$ serves that purpose: under the assumption that $\epsilon \supseteq \epsilon'$ it extends a computation t on $\mathbf{Store}(\epsilon')$ to a computation on $\mathbf{Store}(\epsilon)$ while bypassing the store components that t does not affect.

$$
\begin{aligned}
&\mathbf{MASK}(\epsilon, \epsilon')f \\
&= \lambda k\sigma. f(\lambda y\sigma'.ky(\sigma''_1, \ldots, \sigma''_n))(\pi_{(n,i_1)}(\sigma), \ldots, \pi_{(n,i_r)}(\sigma)) \\
&\quad \text{where } \sigma''_j = \begin{cases} \pi_{(r,\nu)}(\sigma') \ \exists \nu. i_\nu = j \\ \pi_{(n,j)}(\sigma) \ \text{otherwise} \end{cases} \\
&\quad \text{and } \{\rho_1, \ldots, \rho_n\} = \mathbf{EffRegions}(\epsilon) \text{ and } \{\rho_{i_1}, \ldots, \rho_{i_r}\} = \mathbf{EffRegions}(\epsilon') \\
&\quad \text{and } \{i_1, \ldots, i_r\} \subseteq \{1, \ldots, n\}
\end{aligned}
$$

The whole **MASK** construction serves as a macro, that is, it is expanded at translation time. Using **MASK** we can specify the translation of the application: the computation of E_1 and E_2 is masked to effect ϵ_1 and ϵ_2, respectively, while the computation of the function's body is masked to effect ϵ.

$$
\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta_1\Big(\Gamma \vdash E_1 : \tau_2 \xrightarrow{\epsilon} \tau_1, \epsilon_1\Big) \qquad \Delta_2\Big(\Gamma \vdash E_2 : \tau_2, \epsilon_2\Big)}{\Gamma \vdash E_1 @ E_2 : \tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2} \right]\!\!\right] \\
&= \lambda k\sigma. \mathbf{MASK}(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon_1)(\mathcal{ET}\,[\![\Delta_1]\!]) \\
&\qquad (\lambda y_1 \sigma. \mathbf{MASK}(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon_2)(\mathcal{ET}\,[\![\Delta_2]\!]) \\
&\qquad\quad (\lambda y_2 \sigma. \mathbf{MASK}(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon)(y_1 @ y_2)k\sigma)\sigma)\sigma
\end{aligned}
$$

The next interesting case is the introduction of a new region. It involves the addition of new empty heap components to the store.

$$
\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(\Gamma \vdash E : \tau, \epsilon \cup \{\overline{\rho}\}\Big) \quad (\mathbf{Obs}(\Gamma) \cup \mathbf{Obs}(\tau)) \cap \{\overline{\rho}\} = \emptyset}{\Gamma \vdash \mathbf{letregion}\ \overline{\rho}\ \mathbf{in}\ E : \tau, \epsilon} \right]\!\!\right] \\
&= \lambda k(\sigma_{i_1}, \ldots, \sigma_{i_r}). \mathcal{ET}\,[\![\Delta]\!]\,(\lambda y(\sigma'_1, \ldots, \sigma'_n).ky(\sigma'_{i_1}, \ldots, \sigma'_{i_r}))(\sigma''_1, \ldots, \sigma''_n) \\
&\quad \text{where } \{\rho_1, \ldots, \rho_n\} = \mathbf{EffRegions}(\epsilon) \cup \{\overline{\rho}\} \\
&\quad \text{and } \{\rho_{i_1}, \ldots, \rho_{i_r}\} = \mathbf{EffRegions}(\epsilon) \\
&\quad \text{and } \{i_1, \ldots, i_r\} \subseteq \{1, \ldots, n\} \\
&\quad \text{and } \sigma''_i = \begin{cases} \sigma_i \text{ if } i \in \{i_1, \ldots, i_r\} \\ \bullet \ \text{ otherwise} \end{cases}
\end{aligned}
$$

For the translation of the operations on references, we need some auxiliary definitions. Like **MASK** they define *translation-time functions*. The first operation allocates a new reference in region ρ of a store of type $\mathbf{Store}(\epsilon)$ where $\rho \in \mathbf{EffRegions}(\epsilon) = \{\rho_1, \ldots, \rho_n\}$.

$$
\begin{aligned}
&\mathbf{alloc}_{\rho_i}^{\{\rho_1, \ldots, \rho_n\}} \\
&= \lambda yk(\sigma_1, \ldots, \sigma_n). \mathbf{fnew}\ \sigma_i\ y\ \lambda(\alpha, \sigma'_i).k\alpha(\sigma_1, \ldots \sigma'_i \ldots, \sigma_n)
\end{aligned}
$$

Dereferencing and assignment are just slight variations of the same theme.

$$\mathbf{deref}_{\rho_i}^{\{\rho_1,\ldots,\rho_n\}} = \lambda\alpha k(\sigma_1,\ldots,\sigma_n).\mathbf{fget}\ \sigma_i\ \alpha\lambda y.ky(\sigma_1,\ldots,\sigma_n)$$

$$\mathbf{assign}_{\rho_i}^{\{\rho_1,\ldots,\rho_n\}} = \lambda\alpha yk(\sigma_1,\ldots,\sigma_n).\mathbf{fupd}\ \sigma_i\ \alpha\ y\ \lambda\sigma_i'.k()(\sigma_1,\ldots\sigma_i'\ldots,\sigma_n)$$

Using the three auxiliary constructions just described, we can continue with the actual translation. The allocation of a new cell uses **alloc** and **MASK** since allocation is an effect.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(\Gamma\vdash E:\tau,\epsilon\Big)}{\Gamma\vdash \mathbf{new}\ E\ \mathbf{at}\ \rho:\mathbf{ref}_\rho\ \tau,\epsilon\cup\{\rho\}}\right]\!\!\right]$$
$$= \lambda k\sigma.\mathbf{MASK}(\epsilon\cup\{\rho\},\epsilon)(\mathcal{ET}\,[\![\Delta]\!])(\lambda y\sigma.\mathbf{alloc}_\epsilon^{\epsilon\cup\{\rho\}}yk\sigma)\sigma$$

Dereferencing is done analogously.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(\Gamma\vdash E:\mathbf{ref}_\rho\ \tau,\epsilon\Big)}{\Gamma\vdash \mathbf{get}\ E\ \mathbf{at}\ \rho:\tau,\epsilon\cup\{\rho\}}\right]\!\!\right]$$
$$= \lambda k\sigma.\mathbf{MASK}(\epsilon\cup\{\rho\},\epsilon)(\mathcal{ET}\,[\![\Delta]\!])(\lambda y\sigma.\mathbf{deref}_\epsilon^{\epsilon\cup\{\rho\}}yk\sigma)\sigma$$

The final case is updating a reference.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta_1\Big(\Gamma\vdash E_1:\mathbf{ref}_\rho\ \tau,\epsilon_1\Big)\qquad\Delta_2\Big(\Gamma\vdash E_2:\tau,\epsilon_2\Big)}{\Gamma\vdash \mathbf{set}\ E_1\ E_2\ \mathbf{at}\ \rho:\mathbf{unit},\epsilon_1\cup\epsilon_2\cup\{\rho\}}\right]\!\!\right]$$
$$= \lambda k\sigma.\mathbf{MASK}(\epsilon_1\cup\epsilon_2\cup\{\rho\},\epsilon_1)(\mathcal{ET}\,[\![\Delta_1]\!])$$
$$(\lambda y_1\sigma.\mathbf{MASK}(\epsilon_1\cup\epsilon_2\cup\{\rho\},\epsilon_2)(\mathcal{ET}\,[\![\Delta_2]\!])$$
$$(\lambda y_2\sigma.\mathbf{assign}_\rho^{\epsilon_1\cup\epsilon_2\cup\{\rho\}}y_1y_2k\sigma)\sigma)\sigma$$

The translation maps well-typed Λ_r-expressions to well-typed Λ_{sto}-expressions.

Lemma 4.1 *If* $\Delta\Big(\Gamma\vdash E:\tau,\epsilon\Big)$ *then* $\mathcal{EV}\,[\![\Gamma]\!]\vdash\mathcal{ET}\,[\![\Delta]\!]:\mathcal{ET}\,[\![\tau,\epsilon]\!]$.

Proof. Induction over a region inference derivation of $\Gamma\vdash E:\tau,\epsilon$.

Since the target language Λ_{sto} of our translation has a well-defined model of computation (the simply-typed call-by-value lambda calculus with products and the semantic algebra of heaps) we can consider the translation as a denotational semantics definition of the dynamic semantics of Λ. In contrast to the semantics defined by our translation, the dynamic semantics of the region language as presented by Tofte and Talpin [32] always passes the entire store. This does no harm in the usual applications of region inference (memory management using a stack discipline [32], parallel execution [18,30]), but in our case it leads to unnecessarily bad results as will become obvious in the description of the BTA in the next section.

5 Binding-Time Annotations

So far we have considered a language Λ with references, a language Λ_r with references and explicit region annotations, and a language Λ_{sto}, which is an augmented lambda calculus. We have defined translations between these languages that define a dynamic semantics of Λ.

Next we wish to establish a specialization semantics for Λ. For this endeavor we start from the target language Λ_{sto}. There exists a well-developed theory for binding-time analysis and specialization for languages like Λ_{sto}. Our aim is to use the translation developed in the previous section to obtain a similar theory for Λ_r.

First, we define annotated versions for all languages involved, the type languages, Λ_{sto}, and Λ_r. Next, we define well-annotatedness criteria for types and expressions. Finally, we extend the ECPSPS translation to annotated expressions so that it maps well-annotated expressions into well-annotated expressions.

5.1 Annotated Languages

Binding-time annotations $b \in B = \{s, d\}$ are ordered by $s < d$, where s marks static entities and d marks dynamic entities. A binding-time annotated Λ_r type has an annotation on each type constructor. Regions are associated to binding times via an explicit region environment R.

$$\begin{aligned} \epsilon &\subseteq \mathbf{RegVars} \\ \tau &::= \mathbf{unit}^b \mid \mathbf{ref}^b_\rho\ \tau \mid \tau \xrightarrow{\epsilon}{}^b\ \tau \end{aligned}$$

$\mathbf{top}(\tau)$ denotes the top-level binding-time annotation of τ, for example $\mathbf{top}(\tau_1 \xrightarrow{\epsilon}{}^b \tau_2) = b$. Not all binding-time annotated types are acceptable:

- the binding time of a reference type must be identical to the binding time of its region;
- the binding time of the value stored in a reference needs to be greater than or equal to the binding time of the reference itself; this rules out dynamic references (and regions) that holds static values;
- a dynamic function must neither take static parameters nor deliver static results;
- a dynamic function must not have an effect on a static region.

The last point already sheds some light as to why our translation is careful not to pass a region to a computation that does not depend on the region. If the binding-time analysis classifies a function as dynamic then every region that is passed to the function must be classified as dynamic. Therefore, if the translation passes superfluous regions to dynamic functions this results in regions that are unnecessarily classified as dynamic.

The above informal description of acceptable annotated types leads to the following definition [11] of well-formed binding-time-annotated types.

$$R \vdash \mathbf{wft}\ \mathbf{unit}^b$$

$$\frac{R \vdash \mathbf{wft}\ \tau \quad b \leq \mathbf{top}(\tau) \quad R(\rho) = b}{R \vdash \mathbf{wft}\ \mathbf{ref}_\rho^b\ \tau}$$

$$\frac{R \vdash \mathbf{wft}\ \tau_1 \quad R \vdash \mathbf{wft}\ \tau_2 \quad b \leq \mathbf{top}(\tau_1) \quad b \leq \mathbf{top}(\tau_2) \quad \forall \rho \in \epsilon. b \leq R(\rho)}{R \vdash \mathbf{wft}\ \tau_1 \xrightarrow{\epsilon}{}^b \tau_2}$$

Fig. 4. Well-annotated Types

Definition 5.1 Suppose $R = \{\rho_1 : b_1, \ldots, \rho_n : b_n\}$ is a binding-time assumption which associates binding times with region variables. A binding-time annotated type τ is *well-annotated* with respect to R if $R \vdash \mathbf{wft}\ \tau$ can be derived using the rules in Fig. 4.

Binding-time annotated Λ_r-expressions are defined by adding a binding-time annotation to each construct (cf. Fig. 5). Given a region derivation for a Λ_r-expression we can decorate the derivation with binding-time annotations. Figure 5 defines a well-decorated derivation. As before, we have omitted the translation part, i.e., the source language expressions. However, we have added a binding-time assumption on region variables. The resulting judgement is $R, \Gamma \vdash E : \tau, \epsilon$ meaning "with binding-time assumption R and type assumption Γ the annotated Λ_{sto}-expression E has annotated type τ and effect ϵ."

Annotated target expressions and their annotated types are standard [9, 31]. For completeness, we define them here again:

$$\begin{array}{rl} t ::= & x \mid t @^b t \mid \lambda^b x.t \mid \mathbf{rec}^b\ f(x).t \\ \mid & (t, \ldots, t)^b \mid \pi^b_{(n,i)}(t) \\ \mid & \bullet^b \mid \mathbf{fnew}^b\ t\ t \mid \mathbf{fget}^b\ t\ t \mid \mathbf{fupd}^b\ t\ t\ t \end{array}$$

The annotated type language:

$$\mu ::= \mathbf{unit}^b \mid \mathbf{loc}^b\ \mu \mid \mathbf{heap}^b\ \mu \mid (\mu \times \ldots \times \mu)^b \mid \mu \rightarrow^b \mu$$

with the standard well-formedness criterion:

Definition 5.2 A Λ_{sto}-type μ is well-formed iff $\mathbf{wft}\ \mu$ can be derived using the rules in Fig. 6.

This is the standard well-formedness criterion for specialization with partially static data: In Λ_{sto} there can be static products whose components may be (independent of each other) dynamic. The well-formedness criterion only forces the components of a dynamic tuple to be dynamic. We will exploit this subsequently to represent the store by a partially static tuple in the extended translation.

If we want to extract the underlying unannotated type or expression from its annotated counterpart, we write $|E|$ ($|\tau|$ respectively) for the expression E (the type τ) with all binding-time annotations removed.

$$R, \Gamma\{x : \tau\} \vdash x : \tau, \emptyset$$

$$\frac{R, \Gamma \vdash E_1 : \tau_2 \xrightarrow{\epsilon}{}^b \tau_1, \epsilon_1 \quad R, \Gamma \vdash E_2 : \tau_2, \epsilon_2}{R, \Gamma \vdash E_1 @^b E_2 : \tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

$$\frac{R, \Gamma\{x : \tau_1\} \vdash E : \tau_2, \epsilon \quad R \vdash \mathbf{wft}\ \tau_1 \xrightarrow{\epsilon}{}^b \tau_2}{R, \Gamma \vdash \lambda^b x.E : \tau_1 \xrightarrow{\epsilon}{}^b \tau_2, \emptyset}$$

$$\frac{R, \Gamma\{f : \tau_1 \xrightarrow{\epsilon}{}^b \tau_2, x : \tau_1\} \vdash E : \tau_2, \epsilon \quad R \vdash \mathbf{wft}\ \tau_1 \xrightarrow{\epsilon}{}^b \tau_2}{R, \Gamma \vdash \mathbf{rec}^b\ f(x).E : \tau_1 \xrightarrow{\epsilon}{}^b \tau_2, \emptyset}$$

$$\frac{R\{\rho : b\}, \Gamma \vdash E : \tau, \epsilon \quad \rho \notin \mathbf{Obs}(\Gamma) \cup \mathbf{Obs}(\tau)}{R, \Gamma \vdash \mathbf{letregion}^b\ \rho\ \mathbf{in}\ E : \tau, \epsilon \setminus \{\rho\}}$$

$$\frac{R, \Gamma \vdash E : \tau, \epsilon \quad R \vdash \mathbf{wft}\ \mathbf{ref}^b_\rho\ \tau}{R, \Gamma \vdash \mathbf{new}^b\ E\ \mathbf{at}\ \rho : \mathbf{ref}^b_\rho\ \tau, \epsilon \cup \{(\rho, b)\}}$$

$$\frac{R, \Gamma \vdash E : \mathbf{ref}^b_\rho\ \tau, \epsilon}{R, \Gamma \vdash \mathbf{get}^b\ E\ \mathbf{at}\ \rho : \tau, \epsilon \cup \{(\rho, b)\}}$$

$$\frac{R, \Gamma \vdash E_1 : \mathbf{ref}^b_\rho\ \tau, \epsilon_1 \quad R, \Gamma \vdash E_2 : \tau, \epsilon_2}{R, \Gamma \vdash \mathbf{set}^b\ E_1\ E_2\ \mathbf{at}\ \rho : \mathbf{unit}, \epsilon_1 \cup \epsilon_2 \cup \{(\rho, b)\}}$$

$$\frac{R, \Gamma \vdash E : \tau, \epsilon \quad \epsilon \subseteq \epsilon'}{R, \Gamma \vdash E : \tau, \epsilon'}$$

Fig. 5. Well-annotatedness Rules for Region Derivations

$$\mathbf{wft}\ \mathbf{unit}^b \qquad \frac{\mathbf{wft}\ \mu \quad b \leq \mathbf{top}(\mu)}{\mathbf{wft}\ \mathbf{loc}^b\ \mu} \qquad \frac{\mathbf{wft}\ \mu \quad b \leq \mathbf{top}(\mu)}{\mathbf{wft}\ \mathbf{heap}^b\ \mu}$$

$$\frac{\mathbf{wft}\ \mu_1 \ldots \mathbf{wft}\ \mu_n \quad b \leq \mathbf{top}(\mu_1) \ldots b \leq \mathbf{top}(\mu_n)}{\mathbf{wft}\ (\mu_1 \times \ldots \times \mu_n)^b}$$

$$\frac{\mathbf{wft}\ \mu_1 \quad \mathbf{wft}\ \mu_2 \quad b \leq \mathbf{top}(\mu_1) \quad b \leq \mathbf{top}(\mu_2)}{\mathbf{wft}\ \mu_1 \rightarrow^b \mu_2}$$

Fig. 6. Well-annotated Pure Lambda Expressions

We sometimes omit the s-annotation if it is understood that the expression or type is annotated. We sometimes drop $@^s$ completely in favor of denoting function application by juxtaposition.

5.2 Translation of Annotated Types

The ECPSPS translation induces a translation of types τ and environments Γ. We extend this translation to annotated types and environments as follows.

Definition 5.3 (Translation of annotated types)

$$\begin{array}{ll}
\mathcal{ET}^b [\![\tau, \epsilon]\!] & = \mathbf{Cont}^b(\tau, \epsilon) \rightarrow^b \mathbf{Store}^b(\epsilon) \rightarrow^b \mathbf{Answer} \\
\mathbf{Cont}^b(\tau, \epsilon) & = \mathcal{EV} [\![\tau]\!] \rightarrow^b \mathbf{Store}^b(\epsilon) \rightarrow^b \mathbf{Answer} \\
\mathbf{Store}^b(\{\rho_1, \dots, \rho_n\}) & = (\mathcal{EV} [\![\tau(\rho_1)]\!] \times \dots \times \mathcal{EV} [\![\tau(\rho_n)]\!])^b \\
\mathcal{EV} [\![\mathbf{unit}^b]\!] & = \mathbf{unit}^b \\
\mathcal{EV} [\![\mathbf{ref}^b_\rho\ \tau]\!] & = \mathbf{loc}^b\ \rho \mathcal{EV} [\![\tau]\!] \\
\mathcal{EV} [\![\tau_1 \stackrel{\epsilon}{\rightarrow}^b \tau_2]\!] & = \mathcal{EV} [\![\tau_1]\!] \rightarrow^b \mathcal{ET}^b [\![\tau_2, \epsilon]\!] \\
\mathcal{EV} [\![\{x : \tau, \dots\}]\!] & = \{x : \mathcal{EV} [\![\tau]\!], \dots\}
\end{array}$$

The notation $\mathbf{Store}^b(\epsilon)$ means that the store is a tuple of binding time b. The well-formedness criterion for product types forces the components of tuples to be dynamic if the tuple constructor itself is dynamic.

The translation on types already gives us some idea of the following translation on expressions. However, we cannot simply translate expressions of binding time b to expressions of type $\mathcal{ET}^b [\![\tau, \epsilon]\!]$ for suitable τ and ϵ. If $b = d$ then we would be unable to retain the static division of the store in regions, let alone have static components in the store. We will see that there are only a few points in the translation (translation of dynamic abstractions and dynamic recursive definitions) where this is really necessary. We will see that these types only arise for intermediate expressions. In all other places we are able to retain the structure of the store.

5.3 Translation of Annotated Expressions

The translation to ECPSPS that we presented above applies only to Λ_r-expressions without annotations. These translation rules are still valid for the constructs annotated with s. It remains to supply the translations for the d-annotated constructs. They are derived from the translation of the static constructs by eta-expansion in strategic places. Such eta-redexes serve as binding-time coercions [6–8].

These coercions work as follows. Suppose there is some dynamic function f of type $\tau_1 \rightarrow^d \tau_2$. It is possible to coerce f to a static function that maps τ_1 to τ_2 by eta-expansion to $\lambda^s z.f@^d z : typeFun[s]\tau_1\tau_2$. Of course, it is still the case that $\mathbf{top}(\tau_1) = \mathbf{top}(\tau_2) = d$.

The same approach works for dynamic pairs p of type $\tau_1 \times^d \tau_2$. Eta-expansion of p yields $(\pi^d_{(2,1)}(p), \pi^d_{(2,2)}(p))^s$ of type $\tau_1 \times^s \tau_2$ where $\mathbf{top}(\tau_1) = \mathbf{top}(\tau_2) = d$.

The first case is dynamic abstraction.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R, \Gamma\{x : \tau_1\} \vdash E : \tau_2, \epsilon\Big) \quad R \vdash \mathbf{wft}\ \tau_1 \xrightarrow{\epsilon}{}^d \tau_2}{R, \Gamma \vdash \lambda^d x.E : \tau_1 \xrightarrow{\epsilon}{}^d \tau_2, \emptyset}\right]\!\!\right]$$

$$\begin{aligned}
&= \lambda k\sigma.k(\lambda^d xk\sigma.\\
&\qquad \mathcal{ET}\,[\![\Delta]\!]\,(\lambda z\sigma.k@^d z@^d(\pi^s_{(n,1)}(\sigma), \ldots, \pi^s_{(n,n)}(\sigma))^d)\\
&\qquad\qquad (\pi^d_{(n,1)}(\sigma), \ldots, \pi^d_{(n,n)}(\sigma)))\sigma\\
&\text{where } n = |\mathbf{EffRegions}(\epsilon)|
\end{aligned}$$

Since the lambda abstraction $\lambda^d xk\sigma.$ on the right side of the translation must be dynamic, the parameters k and σ are forced to be dynamic (cf. 5.2). Therefore, we lose the structure of the continuation k and the store σ. In order to regain the structure we eta-expand k and σ as outlined above. This is possible since k is known to be a function and the effect ϵ prescribes the structure that must be imposed on σ.

Ry now it should be clear why the latent effect ϵ of a function must be as small as possible: If the function is classified as dynamic then we have to translate it using the above case. Rut that means that the structure of the store passed to the abstraction is lost and that all components of the passed store are dynamic. We can restore the structure as demonstrated previously, but we cannot reconstruct the values of components of the store.

The translation of dynamic recursive definitions has essentially the same problems.

$$\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R, \Gamma\{f : \tau_2 \xrightarrow{\epsilon}{}^d \tau_1, x : \tau_2\} \vdash E : \tau_1, \epsilon\Big) \quad R \vdash \mathbf{wft}\ \tau_2 \xrightarrow{\epsilon}{}^d \tau_1}{R, \Gamma \vdash \mathbf{rec}^d\ f(x).E : \tau_2 \xrightarrow{\epsilon}{}^d \tau_1, \emptyset}\right]\!\!\right]$$

$$\begin{aligned}
&= \lambda k\sigma.k(\mathbf{rec}^d\ f(x).\lambda^d k\sigma.\\
&\qquad \mathcal{ET}\,[\![\Delta]\!]\,(\lambda z\sigma.k@^d z@^d(\pi^s_{(n,1)}(\sigma), \ldots, \pi^s_{(n,n)}(\sigma))^d)\\
&\qquad\qquad (\pi^d_{(n,1)}(\sigma), \ldots, \pi^d_{(n,n)}(\sigma)))\sigma\\
&\text{where } n = |\mathbf{EffRegions}(\epsilon)|
\end{aligned}$$

The translation of a dynamic application involves similar problems as the translation of static ones. Therefore, we first need to define an annotated version of the **MASK** construct.

$$
\begin{aligned}
&\mathbf{MASK}^d(\epsilon, \epsilon')f \\
&= \lambda k\sigma.f @^d(\lambda^d y\sigma'.ky(\sigma''_1, \ldots, \sigma''_n)^s)@^d(\pi^s_{(n,i_1)}(\sigma), \ldots, \pi^s_{(n,i_k)}(\sigma))^d \\
&\text{where } \sigma''_j = \begin{cases} \pi^d_{(r,\nu)}(\sigma') \ \exists\nu.i_\nu = j \\ \pi^s_{(n,j)}(\sigma) \ \text{otherwise} \end{cases} \\
&\text{and } \{\rho_1, \ldots, \rho_n\} = \mathbf{EffRegions}(\epsilon) \\
&\text{and } \{\rho_{i_1}, \ldots, \rho_{i_r}\} = \mathbf{EffRegions}(\epsilon')
\end{aligned}
$$

The computation f is applied to a dynamic continuation $(\lambda^d y\sigma'.\ldots)$ and to a dynamic state whose structure is described by ϵ'. The dynamic state is constructed as a dynamic tuple of the statically extracted components (regions) mentioned in ϵ'. Here we use eta-expansion to coerce in the reverse direction, from a static tuple to a dynamic one.

The argument of the static continuation k is constructed by (dynamically) extracting components from the dynamic state σ' that is returned from f and statically extracting components from σ. This has, again, the effect of coercing a dynamic structure to a static one.

We are now able to state the translation of an application.

$$
\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta_1\Big(R, \Gamma \vdash E_1 : \tau_2 \xrightarrow{\epsilon}{}^d \tau_1, \epsilon_1\Big) \qquad \Delta_2\Big(R, \Gamma \vdash E_2 : \tau_2, \epsilon_2\Big)}{R, \Gamma \vdash E_1 @^d E_2 : \tau_1, \epsilon \cup \epsilon_1 \cup \epsilon_2} \right]\!\!\right] \\
&= \lambda k\sigma.\mathbf{MASK}(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon_1)(\mathcal{ET}\, [\![\Delta_1]\!]) \\
&\qquad (\lambda y_1\sigma.\mathbf{MASK}(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon_2)(\mathcal{ET}\, [\![\Delta_2]\!]) \\
&\qquad\quad (\lambda y_2\sigma.\mathbf{MASK}^d(\epsilon \cup \epsilon_1 \cup \epsilon_2, \epsilon)(y_1 @^d y_2)k\sigma)\sigma)\sigma
\end{aligned}
$$

The first two occurrences of **MASK** use the previous definition (or the identical $\mathbf{MASK}^s$). Only the call of the dynamic function needs $\mathbf{MASK}^d$.

The translation of region abstraction remains almost the same.

$$
\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R\{\rho : d\}, \Gamma \vdash E : \tau, \epsilon\Big) \qquad \rho \notin \mathbf{Obs}(\Gamma) \cup \mathbf{Obs}(\tau)}{R, \Gamma \vdash \mathbf{letregion}^d\ \rho\ \mathbf{in}\ E : \tau, \epsilon \setminus \{\rho\}} \right]\!\!\right] \\
&= \lambda k(\sigma_{i_1}, \ldots, \sigma_{i_r}).\mathcal{ET}\, [\![\Delta]\!]\, (\lambda y(\sigma'_1, \ldots, \sigma'_n).ky(\sigma'_{i_1}, \ldots, \sigma'_{i_r}))(\sigma''_1, \ldots, \sigma''_n) \\
&\text{where } \mathbf{EffRegions}(\epsilon) \setminus \{\rho\} = \{\rho_{i_1}, \ldots, \rho_{i_r}\} \\
&\text{and } \mathbf{EffRegions}(\epsilon) = \{\rho_1, \ldots, \rho_n\} \\
&\text{and } \{i_1, \ldots, i_r\} \subseteq \{1, \ldots, n\} \\
&\text{and } \sigma''_i = \begin{cases} \sigma_i \ \text{if } i \in \{i_1, \ldots, i_r\} \\ \bullet^d \ \text{otherwise} \end{cases}
\end{aligned}
$$

The remaining translations of the dynamic versions of the operations on references are straightforward. The annotated versions of the allocation primitives differ from the standard versions only in the annotations of the built-in operations on heaps.

$$
\begin{aligned}
\mathbf{alloc}^{d,\{\rho_1,\dots,\rho_n\}}_{\rho_i} &= \lambda yk(\sigma_1,\dots,\sigma_n).\\
&\quad \mathbf{fnew}^d\ \sigma_i\ y\ \lambda^d(\alpha,\sigma_i').k\alpha(\sigma_1,\dots\sigma_i'\dots,\sigma_n)\\
\mathbf{deref}^{d,\{\rho_1,\dots,\rho_n\}}_{\rho_i} &= \lambda\alpha k(\sigma_1,\dots,\sigma_n).\mathbf{fget}^d\ \sigma_i\ \alpha\ \lambda^d y.k(y)(\sigma_1,\dots,\sigma_n)\\
\mathbf{assign}^{d,\{\rho_1,\dots,\rho_n\}}_{\rho_i} &= \lambda\alpha yk(\sigma_1,\dots,\sigma_n).\mathbf{fupd}^d\ \sigma_i\ \alpha\ y\ \lambda^d\sigma_i'.k()(\sigma_1,\dots\sigma_i'\dots,\sigma_n)
\end{aligned}
$$

The same applies to the actual translation.

$$
\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R,\Gamma\vdash E:\tau,\epsilon\Big)\quad R\vdash\mathbf{wft}\ \mathbf{ref}^d_\rho\ \tau}{R,\Gamma\vdash\mathbf{new}^d\ E\ \mathbf{at}\ \rho:\mathbf{ref}^d_\rho\ \tau,\epsilon\cup\{\rho\}}\right]\!\!\right]\\
&= \lambda k\sigma.\mathbf{MASK}(\epsilon\cup\{\rho\},\epsilon)(\mathcal{ET}\,[\![\Delta]\!])(\lambda y\sigma.\mathbf{alloc}^{d,\epsilon\cup\{\rho\}}_\epsilon yk\sigma)\sigma\\
&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R,\Gamma\vdash E:\mathbf{ref}^d_\rho\ \tau,\epsilon\Big)}{R,\Gamma\vdash\mathbf{get}^d\ E\ \mathbf{at}\ \rho:\tau,\epsilon\cup\{\rho\}}\right]\!\!\right]\\
&= \lambda k\sigma.\mathbf{MASK}(\epsilon\cup\{\rho\},\epsilon)(\mathcal{ET}\,[\![\Delta]\!])(\lambda y\sigma.\mathbf{deref}^{d,\epsilon\cup\{\rho\}}_\epsilon yk\sigma)\sigma\\
&\mathcal{ET}\left[\!\!\left[\frac{\Delta_1\Big(R,\Gamma\vdash E_1:\mathbf{ref}^d_\rho\ \tau,\epsilon_1\Big)\qquad\Delta_2\Big(R,\Gamma\vdash E_2:\tau,\epsilon_2\Big)}{R,\Gamma\vdash\mathbf{set}^d\ E_1\ E_2\ \mathbf{at}\ \rho:\mathbf{unit}^d,\epsilon_1\cup\epsilon_2\cup\{\rho\}}\right]\!\!\right]\\
&= \lambda k\sigma.\mathbf{MASK}(\epsilon_1\cup\epsilon_2\cup\{\rho\},\epsilon_1)(\mathcal{ET}\,[\![\Delta_1]\!])\\
&\qquad(\lambda y_1\sigma.\mathbf{MASK}(\epsilon_1\cup\epsilon_2\cup\{\rho\},\epsilon_2)(\mathcal{ET}\,[\![\Delta_2]\!])\\
&\qquad\quad(\lambda y_2\sigma.\mathbf{assign}^{d,\epsilon_1\cup\epsilon_2\cup\{\rho\}}_\rho y_1y_2k\sigma)\sigma)\sigma
\end{aligned}
$$

The thus defined translation on annotated terms does not differ significantly from the previously defined translation. Indeed, the results of the two translations are eta-convertible after stripping off the annotations.

Lemma 5.4 *Let $\Rightarrow^*_\eta$ denote the reflexive, transitive, and compatible closure of the eta-reduction rules $\lambda x.e@x\to e$ ($x\notin \boldsymbol{fv}(e)$) and $(\pi_{(n,1)}(e),\dots,\pi_{(n,n)}(e))\to e$. Then*

$$|\mathcal{ET}\,[\![E]\!]\,|\Rightarrow^*_\eta\mathcal{ET}\,[\![|E|]\!]\,.$$

Proof. Ry inspection of the translation rules.

6 Correctness Proof

Two steps remain to be done to establish our result. First, we show that the translation of types maps well-formed types to well-formed types. Second, we show that the translation of expression maps well-formed expressions (in the annotated Λ_r) to well-formed expressions (in the annotated Λ_{sto}).

Lemma 6.1 *Let R be a binding-time assumption on region variables. Suppose $R\vdash$ **wft** τ is well-annotated according to Definition 5.1 then* **wft** $\mathcal{ET}^s\,[\![\tau]\!]$ *is* *well-annotated according to Definition 5.2.*

Proof. Induction on the structure of an annotated type, taking advantage of the facts that

(i) **top**(**Answer**) is d, the binding time of the outermost expression which is specialized and

(ii) **top**(**Store**) is d when it is passed to or returned from dynamic functions. $\square$

Lemma 6.2 *If* $\Delta\Big(R, \Gamma \vdash E : \tau, \epsilon\Big)$ *is a well-decorated binding-time decorated region derivation then* $\mathcal{EV}[\![\Gamma]\!] \vdash \mathcal{ET}[\![\Delta]\!] : \mathcal{ET}^s[\![\tau, \epsilon]\!]$ *is a well-decorated binding-time type derivation according to the standard definition [16, 31].*

Proof. Induction on the structure of the derivation $\Delta\Big(R, \Gamma \vdash E : \tau, \epsilon\Big)$. As an example, we present the case for dynamic abstraction.

$$\begin{aligned}
&\mathcal{ET}\left[\!\!\left[\frac{\Delta\Big(R, \Gamma\{x : \tau_1\} \vdash E : \tau_2, \epsilon\Big) \quad R \vdash \mathbf{wft}\ \tau_1 \xrightarrow{\epsilon}{}^{d} \tau_2}{R, \Gamma \vdash \lambda^d x.E : \tau_1 \xrightarrow{\epsilon}{}^{d} \tau_2, \emptyset} \right]\!\!\right] \\
&= \lambda k'\sigma'.k'(\lambda^d xk\sigma. \\
&\qquad \mathcal{ET}[\![\Delta]\!](\lambda z\sigma.k@^d z@^d(\pi^s_{(n,1)}(\sigma), \ldots, \pi^s_{(n,n)}(\sigma))^d) \\
&\qquad\quad (\pi^d_{(n,1)}(\sigma), \ldots, \pi^d_{(n,n)}(\sigma)))\sigma' \\
&\text{where } n = |\mathbf{EffRegions}(\epsilon)|
\end{aligned}$$

By induction we can assume that

$$\mathcal{EV}[\![\Gamma\{x : \tau_1\}]\!] \vdash \mathcal{ET}[\![\Delta]\!] : \mathcal{ET}^s[\![\tau_2, \epsilon]\!]$$

and we have to show that the right side is type correct.

By Definition 5.3 of the type translation we have that

$$\begin{aligned}
\mathcal{ET}^s[\![\tau_2, \epsilon]\!] &= \mathbf{Cont}^s(\tau_2, \epsilon) \to \mathbf{Store}(\epsilon) \to \mathbf{Answer} \\
&= (\mathcal{EV}[\![\tau_2]\!] \to \mathbf{Store}(\epsilon) \to \mathbf{Answer}) \to \mathbf{Store}(\epsilon) \to \mathbf{Answer}
\end{aligned}$$

Hence we need to show that

$$(\lambda z\sigma.k@^d z@^d(\pi^s_{(n,1)}(\sigma), \ldots, \pi^s_{(n,n)}(\sigma))^d) : (\mathcal{EV}[\![\tau_2]\!] \to \mathbf{Store}(\epsilon) \to \mathbf{Answer})$$

and

$$(\pi^d_{(n,1)}(\sigma), \ldots, \pi^d_{(n,n)}(\sigma)) : \mathbf{Store}(\epsilon)$$

under the type assumption

$$\mathcal{EV}[\![\Gamma]\!]\{x : \mathcal{EV}[\![\tau_1]\!], k : \mathbf{Cont}^d(\tau_2, \epsilon), \sigma : \mathbf{Store}^d(\epsilon)\}$$

This is easily verified.

Since

$$\lambda^d xk\sigma.\ldots$$

now has type

$$\mathcal{EV}[\![\tau_1]\!] \to^d \mathbf{Cont}^d(\tau_2, \epsilon) \to^d \mathbf{Store}^d(\epsilon) \to^b \mathbf{Answer}$$

which is the same type as

$$\mathcal{ET}^d \left[\!\left[\tau_1 \xrightarrow{\epsilon}{}^d \tau_2\right]\!\right]$$

It is easy to see that the right side of the translation is well-typed: Under the assumption

$$\mathcal{EV} [\![\Gamma]\!] \{k' : \mathbf{Cont}^s(\tau_1 \xrightarrow{\epsilon}{}^d \tau_2, \emptyset), \sigma' : \mathbf{Store}(\emptyset)\}$$

we find the type of the right side is

$$\mathbf{Cont}^s(\tau_1 \xrightarrow{\epsilon}{}^d \tau_2, \emptyset) \to \mathbf{Store}(\emptyset) \to \mathbf{Answer}$$

which is the same as

$$\mathcal{ET} \left[\!\left[\tau_1 \xrightarrow{\epsilon}{}^d \tau_2, \emptyset\right]\!\right].$$

(In the above calculation we have often used weakening without mentioning it explicitly, to drop assumptions on k or σ from some extension of Γ.) □

6.1 Result

Theorem 6.3 *The region-based BTA is correct.*

Proof.

(i) The translation from the source language to the region language Λ_r preserves the semantics [30, 32].

(ii) The ECPSPS translation from Λ_r to Λ_{sto} preserves the semantics, in fact we can view it as defining the semantics.

(iii) The annotated version of the ECPSPS translation is eta-equivalent to the unannotated translation.

(iv) The annotated ECPSPS translation maps RBTA-well-annotated Λ_r expressions (according to Def. 5.1) to standard BTA-well-annotated expressions in Λ_{sto} (according to Def. 5.2).

□

Based on this result we can forget about the translation to Λ_{sto} and can instead apply RBTA to expressions of Λ_r directly using the well-annotatedness criterion of Definition 5.1.

7 Towards the Specializer

The actual implementation of the specializer [11] does not correspond to arbitrary derivations, but it rather presupposes normalized region translations.

Definition 7.1 A region translation is *normalized* if the rules *(mask)* and *(does)* are restricted to only form the premise for an application of the rule *(abs)*.

If we are only interested in normalized translations we can replace the rules *(abs)*, *(does)*, and *(mask)* altogether by the combined rule *(abs-comb)*.

$$\textit{(abs-comb)}\ \frac{\Gamma\{x:\tau_2\} \vdash e \rightsquigarrow E : \tau_1, \epsilon \cup \{\overline{\rho}\} \quad (\mathbf{Obs}(\Gamma\{x:\tau_2\}) \cup \mathbf{Obs}(\tau_1)) \cap \{\overline{\rho}\} = \emptyset \quad \epsilon \subseteq \epsilon'}{\Gamma \vdash \lambda x.e \rightsquigarrow \lambda x.E : \tau_2 \xrightarrow{\epsilon'} \tau_1, \emptyset}$$

It is easy to see that the restriction to normalized region translations does not affect the binding-time annotation of source language expressions. We have already remarked that the location of application of the *(does)* rule does not affect the binding-time annotation and we also see that extending the scope of a **letregion** ρ **in** E construct (confined to the body of a lambda abstraction) does not do so, either.

In the implementation of the specializer [11], we do *not* translate the source program as outlined here, but we rather translate the specializer to extended continuation-passing store-passing style. All continuations that are annotated as static by our translation are now turned into continuations of the specializer. Likewise, all static regions of the store are now passed as a static store by the specializer. The dynamic aspects of the continuation and the store remain implicit.

The above peculiarities of the translation to ECPSPS turn into peculiarities in the definition of the specializer. Since a dynamic lambda abstraction does not accept a static store parameter, the specializer does not pass a static store to it in a dynamic application and it starts a new empty static store inside each dynamic abstraction. Even the structure of the static store can be left implicit. The specializer can merge the static regions without adversely affecting the binding times.

8 Related Work

BTA for higher-order languages with first-class references has been considered by Dussart and Thiemann [11]. That BTA delivers results identical to the RBTA described here, because it relies on a similar effect system in an ad-hoc setup without region inference in mind. They describe a polynomial inference algorithm for their BTA. Furthermore, they describe a specialization algorithm for the language. The current work explores and formalizes the connection of their BTA to region inference. This connection, the ECPSPS translation, and the resulting correctness proof form the contribution of the present work.

Partial evaluators for languages with first-class references that can perform assignments at specialization time have received some attention recently. Dussart and Thiemann [11] describe an offline partial evaluator for a language similar to the one treated in this paper. Dussart and others [10] describe a type specializer for an extension of Moggi's computational metalanguage [20] by first-class references and the operations thereof. Type specialization performs specialization by type inference and is thus able to overcome some limitations of partial evaluators for typed languages, for example it achieves optimal specialization for a variety of typed higher-order languages. The type specializer models the static part of the store also by an implicit translation to store-passing style where the shape of the store changes over time. This translation is even more fine-grained than the translation presented here: it models the store by a dynamically expanding tuple of contents of cells.

Partial evaluation for languages with pointers [1, 5] has similar problems as discussed in this paper. An interesting related approach is put forward by Moura and others [22]. They propose a method for static analysis and specialization of imperative programs (the source language is C) via a translation to static single assignment form and a sophisticated variant of a translation to store-passing style. Their approach does not rely on region inference, hence they cannot rely on soundness proofs for region and effect inference in the design of their translation. As another difference, our interest in the translation to ECPSPS is its use in the correctness proof, the intended specification of the specializer is presented elsewhere [11].

The correctness of BTA for a lambda calculus has been considered by Nielson [23] using small-step operational semantics, by Gomard [12] on the basis of a denotational specification of the specializer, by Palsberg [24] for a reduction semantics for the two-level lambda calculus, and by Wand [33] for Mogensen's specializer for the lambda calculus [19]. Hatcliff [13] specifies and proves correct a BTA and specializer for PCF using ELF [25]. The specializer is again given by an operational semantics. Hatcliff and Danvy [14] give a correctness proof for a BTA with respect to a specializer based on Moggi's computational metalanguage [20].

9 Conclusion

Region inference provides a solid basis for a BTA for a higher-order imperative language that enables a specializer to perform some operations on first-class references at specialization time. This is an application of region inference not foreseen by its creators. Two simple additions to the standard algorithm provide all the information necessary for BTA.

This work follows other work on the correctness of BTAs in that it ignores pragmatic issues that arise due to polyvariant program-point specialization etc.

The analysis as presented in the paper is a monovariant analysis. There are two ways to improve on that. One way uses the translation to ECPSPS to translate the program into a purely functional one, without any merging of regions. A polyvariant specializer for a purely functional language may then be able to achieve even better specialization results. This approach is closely related to the work of Moura and others [22]. The other way exploits the polymorphism present in the usual region and effect systems. This polymorphism could be combined with binding-time polymorphism in the style of Dussart, Henglein, and Mossin [9, 15]. We expect that such a system could be proved correct in much the same way as the present system.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 16(2):151–195, 1991.

[4] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519, Cambridge, MA, 1991. Springer-Verlag.

[5] Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, Fla., January 1996. ACM Press.

[6] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM Press.

[7] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[8] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, July 1995.

[9] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.

[10] Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[11] Dirk Dussart and Peter Thiemann. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997.

[12] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[13] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Doaitse Swierstra and Manuel Hermenegildo, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[14] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 1997. To appear.

[15] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proc. 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301, Edinburgh, UK, April 1994. Springer-Verlag.

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[17] *Proc. of the 4th Annual Symposium on Logic in Computer Science*, Pacific Grove, CA, June 1989. IEEE Computer Society Press.

[18] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988. ACM Press.

[19] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[20] Eugenio Moggi. Computational lambda-calculus and monads. In LICS 89 [17], pages 14–23.

[21] Peter D. Mosses. *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11. Elsevier Science Publishers, Amsterdam, 1990.

[22] Bàrbara Moura, Charles Consel, and Julia Lawall. Bridging the gap between functional and imperative languages. Publication interne 1027, Irisa, Rennes, France, July 1996.

[23] Flemming Nielson. A formal type system for comparing partial evaluators. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 349–384, Amsterdam, 1988. North-Holland.

[24] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–364, July 1993.

[25] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In LICS 89 [17], pages 313–322.

[26] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[27] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.

[28] Amr Sabry and John Field. Reasoning about explicit and implicit representations of state. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, pages 17–30, Copenhagen, Denmark, June 1993. Yale University, Department of Computer Science, New Haven, CT. Technical Report YALEU/DCS/RR-968.

[29] David A. Schmidt. *Denotational Semantics, A Methodology for Software Development.* Allyn and Bacon, Inc, Massachusetts, 1986.

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.

[31] Peter Thiemann. A unified framework for binding-time analysis. In *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 742–756, Lille, France, April 1997. Springer-Verlag.

[32] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.

[33] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.