Partial Evaluation for Higher-Order Languages with State

Peter Thiemann* pjt@cs.nott.ac.uk
Dirk Dussart† did.cimad.com

Abstract

We have designed and implemented an offline partial evaluator for a higher-order language with first-class references. Its distinguishing feature over other partial evaluators is its ability to perform assignments to local and global references at specialization time for a higher-order language. The partial evaluator consists of a region-based monovariant binding-time analysis and a specializer in essentially continuation-passing store-passing style, thus generalizing type-based binding-time analysis and continuation-based partial evaluation.

The partial evaluator yields good results for realistic problems such as object-oriented programming, unification, and specializer generation.

Keywords: higher-order programming, program transformation, partial evaluation, state

Categories: D.1.1 Applicative (Functional) Programming, D.1.2 Automatic Programming, D.3.1 Formal Definitions and Theory, Semantics, D.3.2 Language Classifications, Applicative languages, D.3.4 Processors, I.2.2 Automatic Programming, Program transformation

1 Introduction

Partial evaluation is an automatic program transformation that performs aggressive constant propagation [44]. *Offline* partial evaluation separates this transformation in two stages. A *binding-time analysis* determines those parts of a program that do not depend on *dynamic* (unknown) data, regardless of the actual value of *static* (known) data. Subsequently, the specializer reduces all static parts and generates *residual code* for the dynamic ones.

During specialization, a partial evaluator should perform all operations that do not depend on dynamic data. In imperative languages, assignments are the essential operations. Therefore, offline partial evaluators for traditional imperative languages like C [2,20], Modula-2 [15], and FORTRAN 77 [47] perform assignments at specialization time. In contrast, current partial evaluators for higher-order languages like ML or Scheme are much more conservative [12]: They defer all operations on references and global variables as well as I/O operations to run time [7, 9, 51]. This treatment of side effects is overly conservative and it seriously limits the

*Dept. of Computer Science, University of Nottingham, University Park, Nottingham, NG7 2RD, England. Much of this work was done while at Universität Tübingen, Germany.

†Cimad Consultants. This work was done while at Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium. Supported by the National Fund for Scientific Research Belgium (N.F.W.O.). This work was initiated during a visit at Universität Tübingen funded by a grant of the Ministerium für Wissenschaft und Forschung.

```
;;; source program
(define-data object (object set get add))
;; a record type with constructor "object" and selectors "set", "get", "add"
(define (main)
  (let ((counter-class
         (lambda ()
           (let* ((slot 0)
                  (mset (lambda (x) (set! slot x) x))
                  (mget (lambda () slot))
                  (madd (lambda (x) (set! slot (+ slot x)) x)))
             (object mset mget madd )))))
    (let ((cnt (counter-class)))
      ((set cnt) 21)
      ((add cnt) ((get cnt)))
      ((get cnt)))))

;;; program specialized with Similix (after assignment elimination)
(define (main-0)
  (let ((slot_1 (make-cell 0)))
    (cell-set! slot_1 21)
    (let* ((g_2 (cell-ref slot_1)) (g_3 (cell-ref slot_1)))
      (cell-set! slot_1 (+ g_3 g_2))
      (cell-ref slot_1))))

;;; program specialized with our partial evaluator
(define ($goal-1) 42)
```

Figure 1: Specializing counter objects

quality of specialization. Therefore, we have designed and implemented a specializer for the full Scheme language that does not have these deficiencies.

1.1 Motivation

Message-passing is a common programming style to emulate classes and objects in Scheme [1]. A typical representation for a class is a function that maps the initial values of the instance variables to a tuple of closures, the representation of an object. The closures represent the possible messages and they share the current values of the instance variables among them. These values are not accessible otherwise, they are local to the object. Sending a message to the object is implemented by calling one of the closures with appropriate arguments. Thus, we have an instance of a programming technique that employs higher-order functions with shared local state.

Consider the Scheme code in the top half of Fig. 1 defining a class of counter objects using the above encoding. The instance variable `slot` is always initialized to zero, so the class function is parameterless. A counter object `cnt` is a triple (or record) `(object mset mget madd)`, where `(set cnt)` is the function that sets the counter, `(get cnt)` is the function that reads the counter, and `(add cnt)` is the function that adds to the counter. All these functions operate on the encapsulated shared state in `slot`.

Let's try to specialize this program. First, we apply Similix [11], a representative of the conservative camp, to this program. The middle part of the figure shows the resulting residual program. Similix removes the message dispatch, but defers all operations on the local state `slot` to run time.

In contrast, our binding-time analysis instructs the specializer to perform *all* operations at specialization time. In effect, it reduces the source program to its value `42` as shown in the lower part of the figure.

Since Similix 5.0 does not handle `set!` directly, we performed an assignment elimination transformation that introduces explicit boxing operations `make-cell`, `cell-ref`, and `cell-set!` by hand before submitting to Similix. This transformation is built into our specializer.

1.2 Contribution

We have designed and implemented a partial evaluator for Scheme [46] that performs imperative operations at specialization time. The system performs *polyvariant program point specialization*, i.e., the specializer may generate many residual expressions from a single source expression (*polyvariance*) by specializing it with respect to different static values and stores and it memoizes its state at certain program points to avoid processing an expression more than once with respect to the same static values and store (*program point specialization*) [44].

Our system relies on a *monovariant binding-time analysis*. Such an analysis assigns each expression in a source program a fixed binding time, just like a monomorphic type system assigns each expression a fixed type.

The main contributions are:

- The specializer performs assignments at specialization time.
- The specializer is written in a novel extended continuation-passing and store-passing style. This style is required for the soundness of the specializer in the presence of side effects [49].
- The binding-time analysis is based on an effect system. It runs in polynomial time.
- The partial evaluator delivers good results for problems that are hard to solve with other specializers:

  Specialization of unification. Unification is hard to specialize and requires rewriting the algorithm to continuation-passing style [19]. Our system specializes a straightforward unification algorithm where variables in terms are represented by references with good results (see 6.2).

  Specializer generation. The specializer with state can automatically generate a specializer for a lazy first-order language from an interpreter which uses updatable thunks (see 6.3). This has not been achieved with other specializers.

The implementation is available for anonymous FTP at
`ftp://ftp.informatik.uni-tuebingen.de/pub/PU/thiemann/software/pgg/`.

$$
\begin{array}{llll}
\text{expressions} & e & ::= & c \mid \text{succ } e \mid \text{pred } e \mid x \mid \lambda x.e \mid e@e \mid \text{if0 } e\ e\ e \mid \text{rec } f(x).e \mid \text{ref } e \mid\ !e \mid e := e \\
\text{types} & \tau & ::= & \text{int} \mid \text{ref } \tau \mid \tau \to \tau
\end{array}
$$

$$
\text{(t-cst)}\ A \vdash c : \text{int} \qquad \text{(t-var)}\ A\{x : \tau\} \vdash x : \tau
$$

$$
\text{(t-succ)}\ \frac{A \vdash e : \text{int}}{A \vdash \text{succ } e : \text{int}} \qquad \text{(t-pred)}\ \frac{A \vdash e : \text{int}}{A \vdash \text{pred } e : \text{int}}
$$

$$
\text{(t-abs)}\ \frac{A\{x : \tau_1\} \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \text{(t-app)}\ \frac{A \vdash e_1 : \tau_2 \to \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 @ e_2 : \tau_1}
$$

$$
\text{(t-if)}\ \frac{A \vdash e_0 : \text{int} \quad A \vdash e_1 : \tau \quad A \vdash e_2 : \tau}{A \vdash \text{if0 } e_0\ e_1\ e_2 : \tau}
$$

$$
\text{(t-rec)}\ \frac{A\{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash e : \tau_2}{A \vdash \text{rec } f(x).e : \tau_1 \to \tau_2}
$$

$$
\text{(t-ref)}\ \frac{A \vdash e_1 : \tau}{A \vdash \text{ref } e_1 : \text{ref } \tau} \qquad \text{(t-deref)}\ \frac{A \vdash e_1 : \text{ref } \tau}{A \vdash\ !e_1 : \tau} \qquad \text{(t-assn)}\ \frac{A \vdash e_1 : \text{ref } \tau \quad A \vdash e_2 : \tau}{A \vdash e_1 := e_2 : \tau}
$$

Figure 2: Syntax and typing rules

1.3 Overview

Section 2 defines the syntax and the denotational semantics of the source language Λ^{ref}, a simply typed call-by-value lambda calculus with first-class references. Section 3 presents a polyvariant specializer for Λ^{ref} in denotational style as an extension of the standard semantics. Section 4 specifies a monovariant binding-time analysis for Λ^{ref}. The analysis performs monomorphic region inference with an efficient reconstruction algorithm. Section 5 discusses some implementation issues. Section 6 presents three example applications of the specializer: programs with cyclic data structures, unification with references, and specializer generation for a lazy language. Finally, Section 7 surveys related work, and Section 8 concludes.

2 Source Language

In this section, we describe the syntax and the denotational semantics of the source language of the partial evaluator.

2.1 Syntax

Figure 2 defines the syntax and typing rules of Λ^{ref}, a call-by-value lambda calculus with first-class references, a recursion operator, integers, and a conditional. The type system derives judgements of the form $A \vdash e : \tau$ (read: under type assumptions A expression e has type τ). The typing rules specify a system of simple types. Types may be recursive without an explicit recursion operator. In addition to the standard constructs of the lambda calculus there are integer constants "c", the successor operation "succ e", the predecessor operation "pred e", and an integer conditional "if0 e e e". "rec $f(x).e$" defines a recursive function f with argument x and body e. Furthermore, "ref e" creates a new memory cell that contains the value of e and returns its address, " $!e$" dereferences a reference, and "$e_1 := e_2$" assigns to a reference. A let-expression "let $x = e_1$ in e_2" is syntactic sugar for "$(\lambda x.e_2)@e_1$."

$$
\begin{array}{rlcl}
 & \text{Comp} & = & \text{Cont} \to \text{Store} \to \text{Val} \\
k \in & \text{Cont} & = & \text{Val} \to \text{Store} \to \text{Val} \\
\rho \in & \text{Env} & = & \text{Var} \to \text{Val} \\
y \in & \text{Val} & = & \text{Int}_\perp \oplus \text{Loc}_\perp \oplus (\text{Val} \to \text{Comp})_\perp \oplus \{\text{error}\}_\perp \\
\sigma \in & \text{Store} & = & \text{Loc} \to (\text{Val} \oplus \{\text{unused}\}_\perp) \\
 & \text{Loc} & = & \text{unspecified infinite set of store locations}
\end{array}
$$

Figure 3: Semantic domains

2.2 Semantics

The semantic domains are sub-domains of some universal domain [60,67]. Figure 3 shows their defining equations. The sub-domain Val of *semantic values* is a coalesced sum of the flat domains of integers $\text{Int}_\perp$, locations $\text{Loc}_\perp$, the lifted continuous function space $(\text{Val} \to \text{Comp})_\perp$, and a lifted one-point domain indicating errors. The sub-domain Comp of *computations* consists of (continuous) functions that map a continuation and a store to a value. The sub-domain Cont of *continuations* consists of functions that map a value and a store to the final answer of a computation, in this case an element of Val. An element σ of the sub-domain Store of *stores* is a finite mapping from store locations to stored values or "unused". An *environment* $\rho \in \text{Env}$ is a finite mapping from variables to values.

The semantic equations of Λ^{ref} in Fig. 4 define a call-by-value semantics which evaluates subexpressions from left to right. The definitions for the constructs are standard in semantics with continuations and store [65] [66, Chap. 9]. Only two points deserve mentioning: the conditional tests an integer for a non-zero value and the assignment returns the assigned value.

The equations make use of the following conventions. Environment update $\rho[v/x]$ is defined by $\rho[v/x](x) = v$ and $\rho[v/x](y) = \rho(y)$ for $y \neq x$. The function $\textbf{int}() : \text{Expr} \to \text{Int}$ maps the syntactic representation of an integer constant to its integer value. The constant **fix** denotes a fixpoint operator at type Val $\to$ Comp (i.e., its type is $((\text{Val} \to \text{Comp}) \to (\text{Val} \to \text{Comp})) \to (\text{Val} \to \text{Comp})$.)

As customary, we have reduced clutter by omitting the injections into the sum type Val. For example, the full equations for the successor and for recursion are:

$$
\begin{array}{lcl}
\mathcal{E}[\![\text{succ } e_1]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_1]\!]\rho(\lambda y.\lambda\sigma.\ \text{case } y \text{ of } \begin{array}[t]{lcl} \text{In}_1(y') & \Rightarrow & k(\text{In}_1(y'+1))\sigma \\ \mid\ z & \Rightarrow & \text{In}_4(\text{error}))\sigma \end{array} \\
\mathcal{E}[\![\text{rec } f(x).e]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.k(\text{In}_3(\textbf{fix}\ \lambda g.\lambda y.\mathcal{E}[\![e]\!]\rho[\text{In}_3(g)/f][y/x]))\sigma
\end{array}
$$

Here, case and pattern matching notation serve to project out of the Val domain, using $\text{In}_i(x)$ to indicate the ith summand of the definition. Pattern matching includes "dropping" of the lifted argument. As a function, $\text{In}_i(x)$ lifts and injects into the ith summand.

Eta-reducing expressions of the form $\lambda\sigma.e\ \sigma$ simplifies many of the equations. The use of σ' indicates those places where eta-reduction is not possible.

$$
\begin{array}{lcl}
\mathcal{E} : \text{Expr} \to \text{Env} \to \text{Comp} & & \\
 & & \\
\mathcal{E}[\![x]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.k(\rho(x))\sigma \\
\mathcal{E}[\![c]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.k(\mathbf{int}(c))\sigma \\
\mathcal{E}[\![\text{succ } e_1]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_1]\!]\rho(\lambda y.\lambda\sigma.k(y+1)\sigma)\sigma \\
\mathcal{E}[\![\text{pred } e_1]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_1]\!]\rho(\lambda y.\lambda\sigma.k(y-1)\sigma)\sigma \\
\mathcal{E}[\![\lambda x.e]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.k(\lambda y.\mathcal{E}[\![e]\!]\rho[y/x])\sigma \\
\mathcal{E}[\![e_1 @ e_2]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_1]\!]\rho(\lambda f.\lambda\sigma.\mathcal{E}[\![e_2]\!]\rho(\lambda a.\lambda\sigma.fak\sigma)\sigma)\sigma \\
\mathcal{E}[\![\text{if0 } e_0\ e_1\ e_2]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_0]\!]\rho(\lambda y.\lambda\sigma'.\text{if0 } y\ (\mathcal{E}[\![e_1]\!]\rho k\sigma')\ (\mathcal{E}[\![e_2]\!]\rho k\sigma'))\sigma \\
\mathcal{E}[\![\text{rec } f(x).e]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.k(\mathbf{fix}\ \lambda g.\lambda y.\mathcal{E}[\![e]\!]\rho[g/f][y/x])\sigma \\
\mathcal{E}[\![\text{ref } e]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e]\!]\rho(\lambda y.\lambda\sigma'.k\alpha(\sigma'[\alpha\mapsto y]))\sigma \\
 & & \qquad\qquad\qquad\qquad \text{where } \sigma'\alpha = \text{unused} \\
\mathcal{E}[\![\ !e]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e]\!]\rho(\lambda\alpha.\lambda\sigma'.k(\sigma'\alpha)\sigma)\sigma \\
\mathcal{E}[\![e_1 := e_2]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{E}[\![e_1]\!]\rho(\lambda\alpha.\lambda\sigma.\mathcal{E}[\![e_2]\!]\rho(\lambda y.\lambda\sigma'.ky(\sigma'[\alpha\mapsto y]))\sigma)\sigma
\end{array}
$$

Figure 4: Semantics of Λ^{ref}

$$
\begin{array}{llll}
\text{binding times} & b & ::= & S \mid D \\
\text{annotated expressions} & E & ::= & x \mid \lambda^b x.E \mid E @^b E \mid \text{rec}^b\ f(x).E \mid \text{ref}^b\ E \mid\ !^b E \mid E :=^b E \mid \\
 & & & \text{lift } E \mid c \mid \text{succ}^b\ E \mid \text{pred}^b\ E \mid \text{if0}^b\ E\ E\ E
\end{array}
$$

Figure 5: Syntax of the annotated source language Λ^{ref}_{ann}

3 Specialization

In this section, we extend the source language to an annotated language by adding a binding-time annotation to each construct. Its semantics is an extension of the standard semantics. The semantics also serves as a functional program implementing an interpreter for the annotated language. The translation to ML or Scheme is straightforward.

3.1 Syntax

Figure 5 defines the syntax of the annotated source language Λ^{ref}_{ann}. It adds a binding-time annotation to each construct of Λ^{ref}, where a binding time b is either S for "static" or D for "dynamic". There is an extra construct "lift E" to propagate numbers from binding time S to binding time D. For a Λ^{ref}_{ann}-expression E, define $|E|$ to be its stripped version ($\in \Lambda^{ref}$) after removing all annotations and all lifts.

The annotated language is not intended for programming: the binding-time analysis automatically derives annotated expressions from Λ^{ref}-expressions. The analysis is subject of Section 4.

3.2 Semantic domains

Our domains need a slight revision (see Fig. 6) to become suitable for specialization: The sub-domain Val$'$ of semantic value has an additional summand RExpr to model residual expressions. There is an obvious embedding from Val into Val$'$.

The domain equations leave RExpr unspecified for several reasons. First, it does not add to understanding the specializer. Second, generating residual expressions involves generating fresh variable names. It is a research topic on its own to investigate a satisfactory model that includes name generation [55]. Third, we may want to parameterize over the residual syntax anyway to replace the syntax constructors, for example, by compiling functions [69].

For these reasons, we treat RExpr as an abstract datatype with an interface reminiscent of higher-order abstract syntax [59], to sidestep the issue of name generation. For convenience, RExpr includes a let-expression with the usual meaning. The specializer requires the following interface for RExpr:

- **quote**() : Int $\to$ RExpr converts a number to its representation as a residual expression. It inserts specialization-time values of type integer into the residual program.
- $\underline{\text{let}}$: RExpr $\times$ (Var $\to$ RExpr) $\to$ RExpr builds a residual let-expression. Roughly, $\underline{\text{let}}\ (\lambda x.e_2, \lambda e_1.)$ can be thought of as building "let $x = e_1$ in e_2". $\underline{\text{lam}}$: (Var $\to$ RExpr) $\to$ RExpr builds a residual lambda expression. $\underline{\text{rec}}$: (Var $\to$ Var $\to$ RExpr) $\to$ RExpr builds a residual rec expression. All three have a functional argument, so that the necessary generation of fresh variables is hidden in RExpr: one possible interpretation is to have the implementation of RExpr apply these functions to freshly generated variables.
- $\underline{\text{succ}}, \underline{\text{pred}}$: RExpr $\to$ RExpr builds a residual successor (predecessor) expression.
- Similarly for $\underline{@}, \underline{:=}$: RExpr $\times$ RExpr $\to$ RExpr, which we write infix, $\underline{\text{if}}$: RExpr $\times$ RExpr $\times$ RExpr $\to$ RExpr, and $\underline{\text{ref}}, \underline{!}$: RExpr $\to$ RExpr.

3.3 Semantic equations

The semantics of Λ^{ref}_{ann} extends the semantics of Λ^{ref} in a natural way. The idea is that the semantics of a completely static expression E (where all annotations are S and which does not contain lift) is identical to the Λ^{ref}-semantics of the stripped expression $|E|$, up to the embedding mentioned in Sec. 3.2.

Therefore, the specialization semantics $\mathcal{S}$ inherits all the defining equations from $\mathcal{E}$ by putting S annotations on all constructs except constants c and variables x (which are copied as is) and then replacing all occurrences of $\mathcal{E}$ by $\mathcal{S}$. Figure 7 defines the additional equations for the constructs annotated with D and for the lift-expression.

Next, we consider each equation in turn, discuss it, and identify constraints that the binding-time analysis must enforce later on. To begin with, it is useful to realize that the continuation k is the function that specializes the context of the current expression. So passing a value to k means to make it available to the context and to start specializing it.

3.3.1 lift

$$\mathcal{S}[\![\text{lift } E]\!] \quad = \quad \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma.k(\mathbf{quote}(y))\sigma)\sigma$$

$$
\begin{array}{rlcl}
 & \text{Comp}' & = & \text{Cont}' \to \text{Store}' \to \text{Val}' \\
k \in & \text{Cont}' & = & \text{Val}' \to \text{Store}' \to \text{Val}' \\
\rho \in & \text{Env}' & = & \text{Var} \to \text{Val}' \\
y \in & \text{Val}' & = & \text{Int}_\perp \oplus \text{Loc}_\perp \oplus \text{RExpr}_\perp \oplus (\text{Val}' \to \text{Comp}')_\perp \oplus \{\text{error}\}_\perp \\
\sigma \in & \text{Store}' & = & \text{Loc} \to (\text{Val}' \oplus \{\text{unused}\}_\perp) \\
 & \text{Loc} & = & \text{unspecified infinite set of store locations} \\
 & \text{RExpr} & = & \text{unspecified domain of residual expressions}
\end{array}
$$

Figure 6: Semantic domains for specialization

$\mathcal{S} : \text{Expr} \to \text{Env}' \to \text{Comp}'$

$$
\begin{array}{lcl}
\mathcal{S}[\![\text{lift } E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma.k(\mathbf{quote}(y))\sigma)\sigma \\
\mathcal{S}[\![\text{succ}^D\ E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma'.\underline{\text{let}}\ (\underline{\text{succ}}\ y, \lambda n.kn\sigma'))\sigma \\
\mathcal{S}[\![\text{pred}^D\ E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma'.\underline{\text{let}}\ (\underline{\text{pred}}\ y, \lambda n.kn\sigma'))\sigma \\
\mathcal{S}[\![\lambda^D x.E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma'.\underline{\text{let}}\ (\underline{\text{lam}}\ \lambda n.\mathcal{S}[\![E]\!]\rho[\overline{n/x}](\lambda y.\lambda\sigma'.y)\sigma_{\text{empty}}, \lambda n.kn\sigma') \\
\mathcal{S}[\![E_1 @^D E_2]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\lambda\sigma.\mathcal{S}[\![E_2]\!]\rho(\lambda y_2.\lambda\sigma'.\underline{\text{let}}\ (y_1 \underline{@} y_2, \lambda n.kn\sigma'))\sigma)\sigma \\
\mathcal{S}[\![\text{if0}^D\ E_1\ E_2\ E_3]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\lambda\sigma'.\underline{\text{if0}}\ y_1\ (\mathcal{S}[\![E_2]\!]\rho k\sigma')\ (\mathcal{S}[\![E_3]\!]\rho k\sigma'))\sigma \\
\mathcal{S}[\![\text{rec}^D\ f(x).E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma'.\underline{\text{let}}\ (\underline{\text{rec}}\ \lambda g.\lambda y.\mathcal{S}[\![E]\!]\rho[g,y/f,x](\lambda y.\lambda\sigma'.y)\sigma_{\text{empty}}, \lambda n.kn\sigma') \\
\mathcal{S}[\![\text{ref}^D\ E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma'.\underline{\text{let}}\ (\underline{\text{ref}}\ y, \lambda n.kn\sigma'))\sigma \\
\mathcal{S}[\![\ !^D E]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma'.\underline{\text{let}}\ (\underline{!}y, \lambda n.kn\sigma'))\sigma \\
\mathcal{S}[\![E_1 :=^D E_2]\!] & = & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\lambda\sigma.\mathcal{S}[\![E_2]\!]\rho(\lambda y_2.\lambda\sigma'.\underline{\text{let}}\ (y_1 \underline{:=}\ y_2, \lambda n.kn\sigma'))\sigma)\sigma
\end{array}
$$

Figure 7: Specializer using continuation-passing and store-passing

The specializer converts a specialization-time number to residual code. After specializing E, the resulting number is converted to code and passed to the continuation. The state is simply passed on.

3.3.2 succ and pred

$$\mathcal{S}[\![\mathrm{succ}^D\ E]\!] \quad = \quad \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E]\!]\rho(\lambda y.\lambda\sigma'.\underline{\mathrm{let}}\ (\underline{\mathrm{succ}}\ y, \lambda n.kn\sigma'))\sigma$$

The specializer has to create a residual successor expression. It takes care not to discard, duplicate, or reorder the residual computation succ y. Therefore, the specializer first specializes E. Next, it constructs the residual expression succ y from the resulting expression y. To make sure that the generated succ y expression gets executed exactly once, the specializer inserts a let-expression and starts specializing the context *in the body of the let-expression* by invoking the continuation. The specializer passes only the let-bound variable n to the continuation. This variable can be discarded, duplicated, or reordered without affecting the number of times that succ y gets computed in the residual program.

In this case, the construction of the let-expression on-the-fly is not required for soundness, because "succ E" has no side effects. However, it avoids code duplication.

The specialization of $\mathrm{pred}^D\ E$ works the same way.

3.3.3 Lambda abstraction and rec

$$\mathcal{S}[\![\lambda^D x.E]\!] \quad = \quad \lambda\rho.\lambda k.\lambda\sigma'.\underline{\mathrm{let}}\ (\underline{\mathrm{lam}}\ \lambda n.\mathcal{S}[\![E]\!]\rho[n/x](\lambda y.\lambda\sigma'.y)\sigma_{\mathrm{empty}}, \lambda n.kn\sigma')$$

The specializer has to construct a residual lambda abstraction. Since there is no way for the specializer to predict the continuation or the contents of the static store for the body of the lambda, it starts afresh, with the empty continuation $\lambda y.\lambda\sigma'.y$ and the empty store σ_{empty}. The lifetime of the new store is confined to the body of the lambda: the empty continuation discards the final store.

In consequence, the body of a dynamic lambda cannot assign to or dereference a static reference that is defined outside the scope of the lambda. The binding-time analysis must ensure that every such external reference is dynamic. In addition, both the argument and the result of the dynamic lambda can never be static.

Again, wrapping the constructed lambda in a let-expression only avoids code duplication. There is no semantical problem with discarding, duplicating, or reordering for this expression, because the evaluation of a lambda expression always terminates and never causes side effects.

The rationale and the rule for specializing $\mathrm{rec}^D\ f(x).E$ is analogous to that for $\lambda^D x.E$.

3.3.4 Application

$$\mathcal{S}[\![E_1 @^D E_2]\!] \quad = \quad \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\lambda\sigma.\mathcal{S}[\![E_2]\!]\rho(\lambda y_2.\lambda\sigma'.\underline{\mathrm{let}}\ (y_1\underline{@}y_2, \lambda n.kn\sigma'))\sigma)\sigma$$

The specializer has to construct a residual application which might have computational effects. To ensure soundness, the specializer must make sure that the residual application gets executed exactly once. The mechanism to implement this is on-the-fly let-insertion as for $\mathrm{succ}^D\ E$. Here its use is mandatory and failure to use it can change the semantics [49].

3.3.5 if0

$$\mathcal{S}[\![\text{if0}^D\ E_1\ E_2\ E_3]\!] \quad = \quad \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y.\lambda\sigma'.\underline{\text{if0}}\ y\ (\mathcal{S}[\![E_2]\!]\rho k\sigma')\ (\mathcal{S}[\![E_3]\!]\rho k\sigma'))\sigma$$

The specializer has to construct a residual conditional expression. In designing this case, there is a choice to make between potential code duplication and unsatisfactory specialization. With the present definition, the specializer may duplicate code because it specializes both branches E_2 and E_3 of the conditional using the same continuation (which performs the specialization of the context of the conditional, hence the potential for code duplication) and the same store. This definition leads to a liberal binding-time analysis.

Quite often, this is the required behavior to obtain satisfactory specialization. For example, consider the expression

$$\lambda d.\text{let}\ x = \text{ref}\ 0\ \text{in pair}\ (\text{if0}\ d\ (x := 1)\ (x := 2))\ (\ !x)$$

which uses a primitive operator "pair" to construct a pair. With our choice of duplicating the continuation, the annotated expression

$$\lambda^D d.\text{let}\ x = \text{ref}^S\ 0\ \text{in pair}^D\ (\text{lift}\ (\text{if0}^D\ d\ (x :=^S 1)\ (x :=^S 2)))\ (\text{lift}\ (\ !^S x))$$

specializes satisfactorily to

$$\lambda d.\text{if0}\ d\ (1, 1)\ (2, 2).$$

The alternative to duplicating the continuation and the store is to cut off both at a dynamic conditional and start afresh in both branches:

$$\begin{aligned}\mathcal{S}[\![\text{if0}^D\ E_1\ E_2\ E_3]\!] \quad = \quad & \lambda\rho.\lambda k.\lambda\sigma.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\lambda\sigma.k(\underline{\text{if0}}\ y_1\ (\mathcal{S}[\![E_2]\!]\rho(\lambda y.\lambda\sigma.y)\sigma_{\text{empty}})\\ & \qquad (\mathcal{S}[\![E_3]\!]\rho(\lambda y.\lambda\sigma.y)\sigma_{\text{empty}}))\sigma)\sigma\end{aligned}$$

However, with this rule in place, the above expression requires a more restrictive annotation:

$$\lambda^D d.\text{let}\ x = \text{ref}^D\ (\text{lift}\ 0)\ \text{in pair}\ (\text{if0}^D\ d\ (x :=^D \text{lift}\ 1)\ (x :=^D \text{lift}\ 2))\ (\ !^D x)$$

No specialization can take place because every construct is annotated dynamic.

In both alternatives, the specializer discards the final static store after processing a branch completely. Why does it work? Well, for the first alternative, the continuation k performs specialization up to the next enclosing dynamic lambda. Since the dynamic lambda cannot not have an effect on the static store, as explained above, its contents are local to the lambda and can safely be discarded. For the second, non-duplicating alternative, the binding-time analysis must ensure that a dynamic conditional does not have effects on the static store, so that it can be discarded, too.

As a consequence, the problem of merging the two different final stores in the context of a dynamic conditional, which is discussed elsewhere [58], disappears in our approach.

Finally, specialization points—as introduced for program point specialization (see Sec. 5.1)—also delimit the continuation and thus limit the amount of code duplication that occurs in practice. The potential code duplication introduced by the dynamic conditional does not affect the complexity of the residual programs in any way.

3.3.6 Operations on references

The operations on references do not yield new insights: $\text{ref}^D\ E$, $!^D E$, and $E_1 :=^D E_2$ all generate residual code which depends on the state or has an effect on it, hence the specializer *must* construct a let-expression. Interestingly, the dynamic store that they operate on only exists at runtime of the residual program, the specializer just keeps the operations on it in the correct order.

3.4 Remarks

Lawall and Thiemann [49] have proved the soundness of this on-the-fly let-insertion algorithm for specializers that deal with arbitrary side effects. Moreover, the residual code is in direct style restricted to *A-normal form* which facilitates compilation [30, 38, 69].

4 Binding-Time Analysis

In this section, we describe the aims and objectives of a binding-time analysis for Λ^{ref}, specify the analysis, and sketch a polynomial-time algorithm for it. The binding-time analysis maps a Λ^{ref}-expression to a Λ^{ref}_{ann}-expression E, with as many constructs as possible annotated as static S, given that no static operation or value should depend on dynamic data. This property of the analysis—its correctness—is proved elsewhere [73].

4.1 Outline

Conceptually, the binding-time analysis has three phases: region inference, binding-time decoration, and binding-time inference. We illustrate these phases with the example from the previous section.

$$\lambda d.\text{let } x = \text{ref } 0 \text{ in pair } (\text{if0 } d\ (x := 1)\ (x := 2))\ (\ !x)$$

In the first phase (Sections 4.2, 4.3, 4.4, and 4.5), a monomorphic region and effect inference system translates a Λ^{ref}-expression into an expression of a language Λ^{ref}_r which is an extension of Λ^{ref} with explicit region annotations. A region inference system divides the store into disjoint regions and assigns each reference to one of them. An effect system approximates the effect of each expression, i.e., the set of regions that may be accessed by evaluation of e. The simplification with respect to other published region inference systems [45, 50, 70, 76] is the omission of polymorphism, leading to an analysis which is monovariant with respect to types, effects, and regions. This restriction leads to a polynomial-time algorithm, as opposed to the exponential algorithm for polymorphic region and effect inference (see Sec. 4.4).

In the example expression only one region is necessary to hold the single reference.

$$\lambda d.\text{letregion } \rho \text{ in let } x = \text{ref}_\rho\ 0 \text{ in pair } (\text{if0 } d\ (x :=_\rho 1)\ (x :=_\rho 2))\ (\ !_\rho x)$$

The region ρ is confined to the body of the letregion. Each operation on a reference carries an annotation that mentions the region involved.

The second phase (Sections 4.6 and 4.7) decorates the derivation of the region inference translation judgement with formal binding-time annotations, i.e., binding-time variables. There are three tasks in this phase.

1. Assign a binding-time variable to each subexpression;
2. assign a binding-time variable to each region;
3. insert lift-expressions where appropriate—on top of every expression of type Int which occurs as an argument of an application, a primitive operation, or on top of the branches of a conditional).

The result is an expression in $\Lambda^{ref}_{r,\,ann}$, the annotated version of Λ^{ref}_{r}. The example expression now reads as follows (simplified for clarity):

$$\begin{array}{l}\lambda^{\beta_1} d.\ \text{letregion}^{\beta_2}\ \rho\ \text{in}\\ \qquad \text{let}^{\beta_3}\ x = \text{ref}^{\beta_4}_{\rho}\ 0\ \text{in}\\ \qquad \text{pair}^{\beta_9}\ (\text{lift}^{\gamma_1}\ (\text{if0}^{\beta_5}\ d\ (x :=^{\beta_6}_{\rho} 1)\ (x :=^{\beta_7}_{\rho} 2)))\ (\text{lift}^{\gamma_2}\ (\ !^{\beta_8}_{\rho} x))\end{array}$$

In the third and final phase (Sec. 4.9 and 4.8), a well-formedness condition gives rise to a set of constraints on the possible values of the binding-time annotations. These constraints ensure that the binding time of a region is equal to the binding time of all references that live in that region: if the region is allocated at specialization time then references to it are static; if the region is allocated at runtime then references to it must be static. In addition, each operation on a reference has the same binding time as the reference itself, and if a dynamic function has an effect on a particular region then the binding time of that region must be dynamic, as well. Finally, the result of the top-level expression is always dynamic.

For the example, these constraints amount to:

		constraint	explanation
β_1	$=$	D	top-level expression
β_2	$=$	$\beta_4 = \beta_6 = \beta_7 = \beta_8$	operate on same region
β_1	$\leq$	β_5	argument of function
β_1	$\leq$	β_9	result of function
β_9	$\leq$	γ_1	first component of pair
β_9	$\leq$	γ_2	second component of pair

Such a set of constraints always has a least (read: as static as possible) solution and we find that $\beta_1 = \beta_5 = \beta_9 = \gamma_1 = \gamma_2 = D$ and $\beta_2 = \beta_3 = \beta_4 = \beta_6 = \beta_7 = \beta_8 = S$ define the least solution. It corresponds to the annotation used in the previous section:

$$\lambda^D d.\text{let}\ x = \text{ref}^S\ 0\ \text{in pair}^D\ (\text{lift}\ (\text{if0}^D\ d\ (x :=^S 1)\ (x :=^S 2)))\ (\text{lift}\ (\ !^S x))$$

We have omitted the $\text{letregion}^S\ \rho$ in ... and the region annotations—the specializer ignores them anyway. The region analysis only provides extra structure for the store, so that the analysis can keep static and dynamic regions apart. The static store in the specializer simply bundles together all the static regions which are currently active.

4.2 Region Language

In this section, we define the syntax of expressions, types, and effects of the region language Λ^{ref}_{r}, which is the target language of our first translation step from Λ^{ref}. This step introduces region annotations on operations on references as well as on reference types and it introduces effect annotations on function types.

$$
\begin{array}{llll}
\text{expressions} & E & ::= & c \mid \text{succ } E \mid \text{pred } E \mid \text{if0 } E\ E\ E \mid \\
 & & & x \mid \lambda x.E \mid E@E \mid \text{rec } f(x).E \mid \\
 & & & \text{ref}_\rho\ E \mid\ !_\rho E \mid E :=_\rho E \mid \text{letregion } \rho \text{ in } E \\
\text{region variables} & \rho & \in & \text{RegVar}
\end{array}
$$

Figure 8: Syntax of Λ_r^{ref}

$$
\begin{array}{llll}
\text{effects} & \epsilon & \subseteq & \{\text{init}(\rho), \text{read}(\rho), \text{write}(\rho) \mid \rho \in \text{RegVar}\} \\
\text{types} & \theta & ::= & \text{int} \mid \text{ref}_\rho\ \theta \mid \theta \xrightarrow{\epsilon} \theta
\end{array}
$$

Figure 9: Effects and region-annotated types

Λ_r^{ref} is essentially the language considered by Talpin and Jouvelot [70] extended by effect masking [50], but using the more elegant notation of Tofte and Talpin [76]. Figure 8 defines the syntax of Λ_r^{ref}-expressions.

RegVar is an infinite set of *region variables*. "letregion ρ in E" binds ρ to a newly allocated empty region of memory, "$\text{ref}_\rho\ E$" allocates a new cell in region ρ, "$!_\rho E$" dereferences a reference from region ρ, and "$E :=_\rho E$" assigns to a cell in region ρ. "letregion $\rho_1, \dots, \rho_n$ in E" is an abbreviation for "letregion ρ_1 in $\dots$ letregion ρ_n in E"; for $n = 0$, letregion in E is the same as E. $\text{frv}(E)$ is the set of region variables that appear free in E, its definition is analogous to the standard notion of free variables.

Figure 9 defines a type language with region annotations on reference types and effect annotations on function arrows. An effect ϵ is a description of a potential side effect of the evaluation of an expression. It is a set of atomic effects: the allocation of a cell in region ρ ($\text{init}(\rho)$), dereferencing a cell in region ρ ($\text{read}(\rho)$), and assignment to a cell in region ρ ($\text{write}(\rho)$). For an effect ϵ and a set $R \subseteq \text{RegVar}$ we define the intersection

$$\epsilon \cap R = \bigcup_{\rho \in R} (\epsilon \cap \{\text{read}(\rho), \text{write}(\rho), \text{init}(\rho)\}).$$

We define set difference $\epsilon \setminus R$ analogously.

On top of that we build the type language. The region annotation ρ in the type $\text{ref}_\rho\ \theta$ of references to values of type θ indicates the memory region in which a cell of that type resides. The type $\theta_1 \xrightarrow{\epsilon} \theta_2$ is the type of functions that map values of type θ_1 to values of type θ_2 and have a latent effect described by ϵ. Type assumptions A are defined as usual.

Effects, types, and type assumptions can have free region variables as defined in Figure 10.

4.3 Region Translation

In this section, we define the translation from Λ^{ref} to Λ_r^{ref}. A Λ^{ref}-expression e translates to a Λ_r^{ref}-expression E if the translation judgement $A \vdash e \rightsquigarrow E : \theta, \epsilon$ is derivable. The judgement reads "under type assumptions A source expression e translates into region expression E which has type θ and effect ϵ." Figure 11 shows the translation rules (cf. [70, 76]). The rules

$$
\begin{array}{lcl}
\mathrm{frv}(\epsilon) & = & \{\rho \mid \mathrm{init}(\rho) \in \epsilon \vee \mathrm{read}(\rho) \in \epsilon \vee \mathrm{write}(\rho) \in \epsilon\} \\
\mathrm{frv}(\mathrm{int}) & = & \emptyset \\
\mathrm{frv}(\mathrm{ref}_\rho\ \theta) & = & \{\rho\} \cup \mathrm{frv}(\theta) \\
\mathrm{frv}(\theta_1 \xrightarrow{\epsilon} \theta_2) & = & \mathrm{frv}(\epsilon) \cup \mathrm{frv}(\theta_1) \cup \mathrm{frv}(\theta_2) \\
\mathrm{frv}(A) & = & \bigcup\{\mathrm{frv}(\theta) \mid x : \theta \text{ in } A\}
\end{array}
$$

Figure 10: Free region variables

are syntax-directed, because our main interest is the derivation of an inference algorithm. For the same reason, we have made explicit the fact that there is an effect (variable) for each sub-expression and that the effects are related by set inclusion and—in rules (r-abs) and (r-rec)—by set intersection. Next we consider some of the rules in detail.

The rules (r-var) and (r-cst) show that the translation does not affect variables and constants. These expressions do not have an effect and we would expect the constraint $\epsilon = \emptyset$: Instead we have $\epsilon \subseteq \emptyset$. This is because the rules—like all others—integrate *subeffecting*. Subeffecting allows us to make the effect information less precise by increasing it. This can be necessary if two expressions must have the same type (including the latent effects) but may evaluate to different functions. The rules (r-if), (r-app), and (r-assn) require such type equalities.

The effect part of the rules (r-succ) and (r-pred) just passes on the effect of the subexpression E. Additionally, subeffecting may happen.

The first really interesting rule is (r-abs). It makes use of a new concept, *effect masking*. Effect masking [50] formalizes the encapsulation of all accesses to a particular region ρ. If $A \vdash e \rightsquigarrow E : \theta, \epsilon$ and ρ does not occur free in A and θ then Lucassen and Gifford have shown that ρ is local to the evaluation of E in the sense that the rest of the computation will not access any value stored in ρ.

Back to rule (r-abs). Once the rule has determined the effect ϵ_0 of the body of the lambda, it computes in ϵ_1 the "sub-effect" of ϵ_0 that affects regions mentioned either in the environment or in the types θ_1 or θ_2. The remaining regions $\mathrm{frv}(\epsilon_0 \setminus \epsilon_1)$ are considered to be local to the body of the lambda. They are masked out by the "letregion $\rho_1, \ldots, \rho_n$ in E" expression. Then the rule applies subeffecting to the remaining effect ϵ_1 in $\epsilon_2 \supseteq \epsilon_1$. The resulting effect ϵ_2 is the latent effect of the function. The effect of the entire lambda expression is $\emptyset$, modulo subeffecting.

The (r-app) rule collects the effects of the subexpressions and includes the latent effect of the function that is applied here.

The (r-if) rule only collects the effects of the subexpressions.

The (r-rec) rule is similar in concept to the (r-abs) rule, because it also performs effect masking. However, it does *not* perform subeffecting for the latent effect ϵ on the function arrow. Without this restriction, we would not be able to prove our upcoming Lemma 1 which establishes that our system is essentially a restricted version of system considered by Talpin and Jouvelot [70].

The remaining rules (r-ref), (r-deref), and (r-assn) just collect the effects of their subexpressions and include their own atomic effect in the returned effect. They also include subef-

$$\text{(r-var)}\ \frac{\epsilon \supseteq \emptyset}{A\{x:\theta\} \vdash x \leadsto x : \theta, \epsilon}$$

$$\text{(r-cst)}\ \frac{\epsilon \supseteq \emptyset}{A \vdash c \leadsto c : \mathrm{int}, \epsilon}$$

$$\text{(r-succ)}\ \frac{\begin{array}{c} A \vdash e \leadsto E : \mathrm{int}, \epsilon_0 \\ \epsilon \supseteq \epsilon_0 \end{array}}{A \vdash \mathrm{succ}\ e \leadsto \mathrm{succ}\ E : \mathrm{int}, \epsilon} \qquad \text{(r-pred)}\ \frac{\begin{array}{c} A \vdash e \leadsto E : \mathrm{int}, \epsilon_0 \\ \epsilon \supseteq \epsilon_0 \end{array}}{A \vdash \mathrm{pred}\ e \leadsto \mathrm{pred}\ E : \mathrm{int}, \epsilon}$$

$$\text{(r-abs)}\ \frac{\begin{array}{c} A\{x:\theta_1\} \vdash e \leadsto E : \theta_2, \epsilon_0 \\ \{\rho_1, \ldots, \rho_n\} = \mathrm{frv}(\epsilon_0 \setminus \epsilon_1) \\ \epsilon_1 = \epsilon_0 \cap (\mathrm{frv}(A) \cup \mathrm{frv}(\theta_1) \cup \mathrm{frv}(\theta_2)) \qquad \epsilon_2 \supseteq \epsilon_1 \qquad \epsilon_3 \supseteq \emptyset \end{array}}{A \vdash \lambda x.e \leadsto \lambda x.\mathrm{letregion}\ \rho_1, \ldots, \rho_n\ \mathrm{in}\ E : \theta_1 \xrightarrow{\epsilon_2} \theta_2, \epsilon_3}$$

$$\text{(r-app)}\ \frac{\begin{array}{c} A \vdash e_1 \leadsto E_1 : \theta_2 \xrightarrow{\epsilon_0} \theta_1, \epsilon_1 \quad A \vdash e_2 \leadsto E_2 : \theta_2, \epsilon_2 \\ \epsilon \supseteq \epsilon_0 \cup \epsilon_1 \cup \epsilon_2 \end{array}}{A \vdash e_1 @ e_2 \leadsto E_1 @ E_2 : \theta_1, \epsilon}$$

$$\text{(r-if)}\ \frac{\begin{array}{c} A \vdash e_0 \leadsto E_0 : \mathrm{int}, \epsilon_0 \quad A \vdash e_1 \leadsto E_1 : \theta, \epsilon_1 \quad A \vdash e_2 \leadsto E_2 : \theta, \epsilon_2 \\ \epsilon \supseteq \epsilon_0 \cup \epsilon_1 \cup \epsilon_2 \end{array}}{A \vdash \mathrm{if0}\ e_0\ e_1\ e_2 \leadsto \mathrm{if0}\ E_0\ E_1\ E_2 : \theta, \epsilon}$$

$$\text{(r-rec)}\ \frac{\begin{array}{c} A\{f : \theta_1 \xrightarrow{\epsilon} \theta_2, x : \theta_1\} \vdash e \leadsto E : \theta_2, \epsilon_0 \\ \{\rho_1, \ldots, \rho_n\} = \mathrm{frv}(\epsilon_0 \setminus \epsilon) \\ \epsilon = \epsilon_0 \cap (\mathrm{frv}(A) \cup \mathrm{frv}(\theta_1 \xrightarrow{\epsilon} \theta_2)) \qquad \epsilon_1 \supseteq \emptyset \end{array}}{A \vdash \mathrm{rec}\ f(x).e \leadsto \mathrm{rec}\ f(x).\mathrm{letregion}\ \rho_1, \ldots, \rho_n\ \mathrm{in}\ E : \theta_1 \xrightarrow{\epsilon} \theta_2, \epsilon_1}$$

$$\text{(r-ref)}\ \frac{\begin{array}{c} A \vdash e \leadsto E : \theta, \epsilon_0 \\ \epsilon \supseteq \epsilon_0 \cup \{\mathrm{init}(\rho)\} \end{array}}{A \vdash \mathrm{ref}\ e \leadsto \mathrm{ref}_\rho\ E : \mathrm{ref}_\rho\ \theta, \epsilon}$$

$$\text{(r-deref)}\ \frac{\begin{array}{c} A \vdash e \leadsto E : \mathrm{ref}_\rho\ \theta, \epsilon_0 \\ \epsilon \supseteq \epsilon_0 \cup \{\mathrm{read}(\rho)\} \end{array}}{A \vdash\ !e \leadsto\ !_\rho E : \theta, \epsilon}$$

$$\text{(r-assn)}\ \frac{\begin{array}{c} A \vdash e_1 \leadsto E_1 : \mathrm{ref}_\rho\ \theta, \epsilon_1 \quad A \vdash e_2 \leadsto E_2 : \theta, \epsilon_2 \\ \epsilon \supseteq \epsilon_1 \cup \epsilon_2 \cup \{\mathrm{write}(\rho)\} \end{array}}{A \vdash e_1 := e_2 \leadsto E_1 :=_\rho E_2 : \theta, \epsilon}$$

Figure 11: Region translation

fecting.

4.4 Algorithm

This section outlines an algorithm to compute a derivation of a region translation judgement in polynomial time. Given a Λ^{ref} expression of size n, the algorithm first constructs the standard type derivation. This takes almost-linear time in n using a term graph representation of the types [40]. This representation immediately yields a suitable annotation with region variables: we can simply use the identity of the nodes denoting reference types.

Next, the algorithm attaches an effect variable to each judgement in the type derivation and to each function type constructor therein. The translation rules give rise to a system of recursive set inequations with union and intersection (the latter is due to rules (r-abs) and (r-rec)) on effects. From the inference rules, it is clear that there are no inequations which mention a variable defined by intersection on their left side. Unions can be eliminated from the right sides, so that there are only inequations of the form $X \supseteq C$, where C is a constant, $X \supseteq Y$, where Y is another variable, and equations of the form $X = Y \cap Z$, where Y and Z are variables.

The least solution of such a system can be computed by fixpoint iteration: Each system gives rise to a function $F : S^n \to S^n$ where S is the finite subset of the region variables RegVar that is mentioned in the analyzed expression and n is the number of effect variables X_i. The ith component F_i of F is defined by

1. $F_i(X_1, \dots, X_n) = C_1 \cup \dots \cup C_r \cup X_{j_1} \cup \dots \cup X_{j_s}$ if $X_i \supseteq C_1, \dots, X_i \supseteq C_r$ and $X_i \supseteq X_{j_1}, \dots, X_i \supseteq X_{j_s}$ are all inequations with left side X_i;

2. $F_i(X_1, \dots, X_n) = X_j \cap X_k$ if $X_i = X_j \cap X_k$ is an equation;

3. $F_i(X_1, \dots, X_n) = \emptyset$ if there is no equation or inequation with left side X_i.

Obviously, each F_i is monotone and (since S is finite) continuous with respect to set inclusion. By Tarksi's fixpoint theorem, F has a least fixpoint and it can be computed by iterating F starting with $(\emptyset, \dots, \emptyset)$. The fixpoint gives rise to a derivation of the desired region translation judgement by substituting the resulting sets. This derivation is minimal in a sense that we will not formalize here.

If n is the size of the program then the computation of the least fixpoint takes $O(n^4)$ time in the worst case: one occurrence of an (r-abs) or an (r-rec) rule generates $O(n^2)$ inequations in the worst case, since an assumption can have $O(n)$ entries and each type may have size $O(n)$. Therefore, we generate $n \cdot O(n^2) = O(n^3)$ constraints for the entire expression. To solve the constraints, each region variable (there are $O(n)$ of them) is propagated at most once through each constraint.

4.5 Simplified Translation Rules

In this section, we simplify the system of the previous section. The previous system had syntax-directed rules that helped to construct the inference algorithm and argue about its complexity. Now, —working towards a binding-time decoration of a region derivation—we will use a slightly modified set of rules to simplify the definition of the binding-time decoration.

In the modified system, effect masking and subeffecting are separate rules. We do not lose anything in that transition, because for every judgement derivable in the original system there will be a corresponding judgement in the modified system.

$$\text{(r-abs')} \quad \frac{A\{x:\theta_1\} \vdash e \rightsquigarrow E : \theta_2, \epsilon}{A \vdash' \lambda x.e \rightsquigarrow \lambda x.E : \theta_1 \xrightarrow{\epsilon} \theta_2, \emptyset}$$

$$\text{(r-rec')} \quad \frac{A\{f:\theta_1 \xrightarrow{\epsilon} \theta_2, x:\theta_1\} \vdash' e \rightsquigarrow E : \theta_2, \epsilon}{A \vdash' \text{rec } f(x).e \rightsquigarrow \text{rec } f(x).E : \theta_1 \xrightarrow{\epsilon} \theta_2, \emptyset}$$

$$\text{(r-esub')} \quad \frac{A \vdash' e \rightsquigarrow E : \theta, \epsilon}{A \vdash' e \rightsquigarrow E : \theta, \epsilon'} \; \epsilon \subseteq \epsilon'$$

$$\text{(r-mask')} \quad \frac{\begin{array}{c} A \vdash' e \rightsquigarrow E : \theta, \epsilon \\ \rho \in \text{frv}(\epsilon) \quad \rho \notin \text{frv}(A) \cup \text{frv}(\theta) \quad \epsilon' = \epsilon \setminus \{\rho\} \end{array}}{A \vdash' e \rightsquigarrow \text{letregion } \rho \text{ in } E : \theta, \epsilon'}$$

Figure 12: Simplified region translation rules

Definition 1 The judgement for the simplified region translation $A \vdash' e \rightsquigarrow E : \theta, \epsilon$ is defined by the following rules.

1. The rules (r-abs'), (r-rec'), (r-esub'), and (r-mask') defined in Fig. 12.
2. The rules from Fig. 11, except (r-abs) and (r-rec), after replacing each occurrence of $\epsilon \supseteq Y$ by $\epsilon = Y$ and each occurrence of $\vdash$ by $\vdash'$.

Lemma 1

$$A \vdash e \rightsquigarrow E : \theta, \epsilon \qquad \textit{implies} \qquad A \vdash' e \rightsquigarrow E : \theta, \epsilon.$$

Proof: By induction on the derivation of $A \vdash e \rightsquigarrow E : \theta, \epsilon$. The only interesting cases are those where the last rule in the derivation is (r-abs) or (r-rec). The other cases are immediate by appeal to the inductive hypothesis, applying the corresponding rule for $\vdash'$, and applying the rule (r-esub'). We demonstrate the case for (r-abs), the case for (r-rec) works analogously.

Suppose the last rule in the derivation is

$$\text{(r-abs)} \quad \frac{\begin{array}{c} A\{x:\theta_1\} \vdash e \rightsquigarrow E : \theta_2, \epsilon_0 \\ \{\rho_1, \ldots, \rho_n\} = \text{frv}(\epsilon_0 \setminus \epsilon_1) \\ \epsilon_1 = \epsilon_0 \cap (\text{frv}(A) \cup \text{frv}(\theta_1) \cup \text{frv}(\theta_2)) \qquad \epsilon_2 \supseteq \epsilon_1 \qquad \epsilon_3 \supseteq \emptyset \end{array}}{A \vdash \lambda x.e \rightsquigarrow \lambda x.\text{letregion } \rho_1, \ldots, \rho_n \text{ in } E : \theta_1 \xrightarrow{\epsilon_2} \theta_2, \epsilon_3}$$

By induction, there is a derivation for

$$A\{x:\theta_1\} \vdash' e \rightsquigarrow E : \theta_2, \epsilon_0 \tag{1}$$

An auxiliary induction shows that for all $0 \leq i \leq n$ there exists $\epsilon \subseteq \epsilon_0$ such that

$$A\{x:\theta_1\} \vdash' e \rightsquigarrow \text{letregion } \rho_{n-i+1}, \ldots, \rho_n \text{ in } E : \theta_2, \epsilon$$

and $\text{frv}(\epsilon) \cap \{\rho_1, \ldots, \rho_n\} = \{\rho_1, \ldots, \rho_{n-i}\}$.

- Case $i = 0$: with $\epsilon = \epsilon_0$ we have

$$A\{x : \theta_1\} \vdash' e \rightsquigarrow \text{letregion} \;\; \text{in } E : \theta_2, \epsilon$$

by (1) and also $\text{frv}(\epsilon) \cap \{\rho_1, \ldots, \rho_n\} = \{\rho_1, \ldots, \rho_{n-i}\}$ because $\{\rho_1, \ldots, \rho_n\} \subseteq \text{frv}(\epsilon) = \text{frv}(\epsilon_0)$ since rule (r-abs) is applicable.

- Case $0 < i \leq n$: by the auxiliary inductive hypothesis, there exists an $\epsilon \subseteq \epsilon_0$ such that

$$A\{x : \theta_1\} \vdash' e \rightsquigarrow \text{letregion } \rho_{n-(i-1)+1}, \ldots, \rho_n \text{ in } E : \theta_2, \epsilon$$

and $\text{frv}(\epsilon) \cap \{\rho_1, \ldots, \rho_n\} = \{\rho_1, \ldots, \rho_{n-(i-1)}\}$. Since $\rho_{n-(i-1)} \in \text{frv}(\epsilon)$ (by the preceding equation) and $\rho_{n-(i-1)} \notin \text{frv}(A) \cup \text{frv}(\theta_1) \cup \text{frv}(\theta_2)$ (since $\{\rho_1, \ldots, \rho_n\} = \text{frv}(\epsilon_0) \setminus (\text{frv}(A) \cup \text{frv}(\theta_1) \cup \text{frv}(\theta_2))$ by applicability of (r-abs)) the rule (r-mask') is applicable and yields

$$A\{x : \theta_1\} \vdash' e \rightsquigarrow \text{letregion } \rho_{n-(i-1)}, \rho_{n-(i-1)+1}, \ldots, \rho_n \text{ in } E : \theta_2, \epsilon \setminus \{\rho_{n-(i-1)}\}$$

where $\epsilon \setminus \{\rho_{n-(i-1)}\} \subseteq \epsilon_0$ because of the assumption $\epsilon \subseteq \epsilon_0$. Furthermore,

$$\begin{aligned} \text{frv}(\epsilon \setminus \{\rho_{n-(i-1)}\}) \cap \{\rho_1, \ldots, \rho_n\} &= (\text{frv}(\epsilon) \cap \{\rho_1, \ldots, \rho_n\}) \setminus \{\rho_{n-(i-1)}\} \\ &= \{\rho_1, \ldots, \rho_{n-(i-1)}\} \setminus \{\rho_{n-(i-1)}\} \\ &= \{\rho_1, \ldots, \rho_{n-i}\}. \end{aligned}$$

For $i = n$ we obtain an $\epsilon \subseteq \epsilon_0$ such that

$$A\{x : \theta_1\} \vdash' e \rightsquigarrow \text{letregion } \rho_1, \ldots, \rho_n \text{ in } E : \theta_2, \epsilon \tag{2}$$

and $\text{frv}(\epsilon) \cap \{\rho_1, \ldots, \rho_n\} = \{\rho_1, \ldots, \rho_{n-n}\} = \emptyset$.

Hence the final ϵ is exactly ϵ_1 from rule (r-abs). Applying the rule (r-esub') to (2) for $\epsilon_2 \supseteq \epsilon_1$ from rule (r-abs) yields

$$A\{x : \theta_1\} \vdash' e \rightsquigarrow \text{letregion } \rho_1, \ldots, \rho_n \text{ in } E : \theta_2, \epsilon_2. \tag{3}$$

Apply (r-abs') to get

$$A \vdash' \lambda x.e \rightsquigarrow \lambda x.\text{letregion } \rho_1, \ldots, \rho_n \text{ in } E : \theta_1 \xrightarrow{\epsilon_2} \theta_2, \emptyset \tag{4}$$

and finally (r-esub') for $\emptyset \subseteq \epsilon_3$ to obtain the result

$$A \vdash' \lambda x.e \rightsquigarrow \lambda x.\text{letregion } \rho_1, \ldots, \rho_n \text{ in } E : \theta_1 \xrightarrow{\epsilon_2} \theta_2, \epsilon_3. \tag{5}$$

4.6 Annotated Region Language

The next step adds binding-time annotations to the region language, both to the expression language and to the type language. Figure 13 defines the syntax of the expressions and the types of the binding-time-annotated region language $\Lambda^{ref}_{r,\,ann}$. In the expression language, there are binding-time annotated versions of all Λ^{ref}_r-expressions (except variables and constants) and a lift-expression. The construct "letregion^b ρ in E" binds ρ to a new region of binding time b for the execution of E.

In the type language, every type carries a binding-time annotation that denotes the binding time when a value of this type is available for computation. $\text{Top}(\sigma)$ denotes the top-level binding-time annotation of σ, for example $\text{Top}(\sigma_1 \xrightarrow{\epsilon}{}^b \sigma_2) = b$. Furthermore, we define stripped expressions, types, and type assumptions.

$$
\begin{array}{llll}
\text{annotated expressions} & E & ::= & \text{lift } E \mid c \mid \text{succ}^b\ E \mid \text{pred}^b\ E \mid \text{if0}^b\ E\ E\ E \mid \\
 & & & x \mid \lambda^b x.E \mid E @^b E \mid \text{rec}^b\ f(x).E \mid \\
 & & & \text{ref}^b_\rho\ E \mid\ !^b_\rho E \mid E :=^b_\rho E \mid \text{letregion}^b\ \rho \text{ in } E \\
\text{binding times} & b & \in & BT = \{S, D\} \\
\text{annotated types} & \sigma & ::= & \text{int}^b \mid \text{ref}^b_\rho\ \sigma \mid \sigma \xrightarrow{\epsilon}{}^b \sigma
\end{array}
$$

Figure 13: Annotated region language $\Lambda^{ref}_{r,\ ann}$

Definition 2 1. $|\sigma|$ is the region-annotated type that is obtained by stripping all binding-time annotations from σ.

2. $|A|$ is defined by $x : |\sigma|$ in $|A|$ iff $x : \sigma$ in A.

3. $|E|$ is the Λ^{ref}_r expression that is obtained from E by stripping all binding-time annotations and all lift constructs.

Not all binding-time annotated types are acceptable. Using the ordering $S < D$ on binding times and our knowledge from the design of the specializer, we derive some constraints that lead to the definition of a well-formed binding-time annotated type.

- The binding time of a region is equal to the binding time of all references (addresses) into this region. Therefore, *the top-level binding time of a reference type must be equal to the binding time of its region.*

- The value stored in a cell depends on its address. If the address is dynamic then the value cannot be static. Therefore, *the binding time of the value stored in a reference needs to be greater than or equal to the binding time of the reference to that cell.* However, a static reference may contain dynamic values.

- A dynamic function must neither take static parameters nor deliver static results [26, 35, 40]: *the binding time of a function must be less than or equal to the binding time of the argument type and less than or equal to the binding time of the result type.*

- A dynamic function must not have an effect on a static region. This coincides precisely with the observation that the specializer cannot pass the static store to a dynamic abstraction, but rather starts a new local static store in the scope of the abstraction. The letregion-expression introduced in rule (r-abs) identifies those regions that are candidates for the local static store inside the body of the function. Therefore, *the binding times of the regions mentioned in the latent effect of a function must be greater than or equal to the binding time of the function itself.*

  Of course, a static function may have an effect on static and dynamic regions. The effects on static regions happen at specialization time while the effects on dynamic regions yield residual code.

The above description of acceptable annotated types leads to the following definition [73] of well-formed binding-time-annotated types.

$$B \vdash \mathrm{int}^b \text{ wft}$$

$$\frac{B \vdash \sigma \text{ wft} \quad b \leq \mathrm{Top}(\sigma) \quad B(\rho) = b}{B \vdash \mathrm{ref}_\rho^b\ \sigma \text{ wft}}$$

$$\frac{B \vdash \sigma_1 \text{ wft} \quad B \vdash \sigma_2 \text{ wft} \quad b \leq \mathrm{Top}(\sigma_1) \quad b \leq \mathrm{Top}(\sigma_2) \quad \forall\rho \in \mathrm{frv}(\epsilon).b \leq B(\rho)}{B \vdash \sigma_1 \xrightarrow{\epsilon}{}^b\ \sigma_2 \text{ wft}}$$

Figure 14: Well-annotated region types

Definition 3 A binding-time assumption B is a finite mapping from region variables to binding times. $\mathrm{dom}(B)$ is the domain of this mapping.

Suppose B is a binding-time assumption. A binding-time-annotated type σ is *well-formed* with respect to B if $B \vdash \sigma$ wft can be derived using the rules in Fig 14. A type assumption A is well-formed with respect to B, $B \vdash A$ wft, if $B \vdash \sigma$ wft for all $x : \sigma$ in A.

The notion of well-formedness is the main source of constraints for the binding-time analysis. The typing part of the system provides additional equalities.

4.7 Binding-Time Annotation

In this section, we give the typing rules for $\Lambda^{\mathit{ref}}_{r,\ \mathit{ann}}$. They are built on top of the rules for the translation judgement $A_0 \vdash' e \rightsquigarrow E_0 : \theta, \epsilon$, by decorating the expression and the types with binding-time annotations. To avoid notational clutter, we omit the translation part $e \rightsquigarrow$ of the judgement and show only the resulting $\Lambda^{\mathit{ref}}_{r,\ \mathit{ann}}$-expression. The judgement has the form $A, B \vdash E : \sigma, \epsilon$: "with type assumption A and binding-time assumption B the $\Lambda^{\mathit{ref}}_{r,\ \mathit{ann}}$-expression E has binding-time-annotated type σ and effect ϵ." Figure 15 shows the inference rules.

The new parts in the rules lies in the introduction of binding-time annotations and in the insistence on well-formedness in the rules that describe the construction of data, i.e., abstraction (b-abs), recursion (b-rec), and reference creation (b-ref). The assumption of the (b-var) rule guarantees the well-formedness of the type assumption A. Constants (b-cst) are always static and must be lifted if they are used in a dynamic context. This is captured by the additional binding-time coercion rule (b-lift) that lifts static integers to code. This rule is standard in binding-time analyses.

One rule that is different from other presentations of binding-time analysis is (b-if). The rule states that a conditional expression is well-formed if the types of the branches are identical. Contrary to the standard rules, the binding time of the result type σ is independent of the binding time of the condition. This is possible because the specializer propagates the full context to both branches of the conditional (cf. 3.3.5). If we used the alternative specialization rule mentioned there, we would have to add the constraint $b \leq \mathrm{Top}(\sigma)$.

Lemma 2 *If $A, B \vdash E : \sigma, \epsilon$ is derivable then $B \vdash \sigma$ wft and $B \vdash A$ wft.*

Proof: Induction on the derivation of $A, B \vdash E : \sigma, \epsilon$.

Therefore, in rule (b-deref) $\mathrm{ref}_\rho^b\ \sigma$ is well-formed and will be accessed by a dereference operation annotated with b. The same holds for the assignment operation (b-assn). If the

$$\text{(b-var)}\ \frac{B \vdash A \text{ wft} \quad x : \sigma \text{ in } A}{A, B \vdash x : \sigma, \emptyset}$$

$$\text{(b-cst)}\ \frac{B \vdash A \text{ wft}}{A, B \vdash c : \text{int}^S, \emptyset}$$

$$\text{(b-succ)}\ \frac{A, B \vdash E : \text{int}^b, \epsilon}{A, B \vdash \text{succ}^b\ E : \text{int}^b, \epsilon}$$

$$\text{(b-pred)}\ \frac{A, B \vdash E : \text{int}^b, \epsilon}{A, B \vdash \text{pred}^b\ E : \text{int}^b, \epsilon}$$

$$\text{(b-lift)}\ \frac{A, B \vdash E : \text{int}^b, \epsilon \quad b < b'}{A, B \vdash \text{lift}\ E : \text{int}^{b'}, \epsilon}$$

$$\text{(b-abs)}\ \frac{A\{x : \sigma_1\}, B \vdash E : \sigma_2, \epsilon \quad B \vdash \sigma_1 \xrightarrow{\epsilon}{}^{b} \sigma_2 \text{ wft}}{A, B \vdash \lambda^b x.E : \sigma_1 \xrightarrow{\epsilon}{}^{b} \sigma_2, \emptyset}$$

$$\text{(b-app)}\ \frac{A, B \vdash E_1 : \sigma_2 \xrightarrow{\epsilon}{}^{b} \sigma_1, \epsilon_1 \quad A, B \vdash E_2 : \sigma_2, \epsilon_2}{A, B \vdash E_1 @^b E_2 : \sigma_1, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

$$\text{(b-if)}\ \frac{A, B \vdash E_0 : \text{int}^b, \epsilon_0 \quad A, B \vdash E_1 : \sigma, \epsilon_1 \quad A, B \vdash E_2 : \sigma, \epsilon_2}{A, B \vdash \text{if0}^b\ E_0\ E_1\ E_2 : \sigma, \epsilon_0 \cup \epsilon_1 \cup \epsilon_2}$$

$$\text{(b-rec)}\ \frac{A\{f : \sigma, x : \sigma_1\}, B \vdash E : \sigma, \epsilon \quad B \vdash \sigma \text{ wft}}{A, B \vdash \text{rec}^b\ f(x).E : \sigma, \emptyset}\ \sigma = \sigma_1 \xrightarrow{\epsilon}{}^{b} \sigma_2$$

$$\text{(b-ref)}\ \frac{A, B \vdash E : \sigma, \epsilon \quad B \vdash \text{ref}^b_\rho\ \sigma \text{ wft}}{A, B \vdash \text{ref}^b_\rho\ E : \text{ref}^b_\rho\ \sigma, \epsilon \cup \{\text{init}(\rho)\}}$$

$$\text{(b-deref)}\ \frac{A, B \vdash E : \text{ref}^b_\rho\ \sigma, \epsilon}{A, B \vdash\ !^b_\rho E : \sigma, \epsilon \cup \{\text{read}(\rho)\}}$$

$$\text{(b-assn)}\ \frac{A, B \vdash E_1 : \text{ref}^b_\rho\ \sigma, \epsilon_1 \quad A, B \vdash E_2 : \sigma, \epsilon_2}{A, B \vdash E_1 :=^b_\rho E_2 : \sigma, \epsilon_1 \cup \epsilon_2 \cup \{\text{write}(\rho)\}}$$

$$\text{(b-esub)}\ \frac{A, B \vdash E : \sigma, \epsilon}{A, B \vdash E : \sigma, \epsilon'}\ \epsilon \subseteq \epsilon'$$

$$\text{(b-mask)}\ \frac{\begin{array}{c} A, B\{\rho : b\} \vdash E : \sigma, \epsilon \\ \rho \in \text{frv}(\epsilon) \quad \rho \notin \text{frv}(A) \cup \text{frv}(\sigma) \quad \epsilon' = \epsilon \setminus \{\rho\} \end{array}}{A, B \vdash \text{letregion}^b\ \rho \text{ in } E : \sigma, \epsilon'}$$

Figure 15: Decorated region inference

reference has well-formed type $\text{ref}_\rho^b\ \sigma$ and the argument has type σ then the assignment will be performed at the time indicated by b. The binding time $\text{Top}(\sigma)$ may be greater than or equal to b, it may be dynamic even if b, the binding time of the address, is static. Likewise, in (b-app) the type of E is well-formed.

4.8 Properties of Binding-Time Annotations

We are now interested in finding the "most static" binding-time annotation for an expression, once we have chosen a certain derivation in Λ_r^{ref} as the basis (for example, using the algorithm in Sec. 4.4). To formalize this, we define the notion of a completion of a Λ_r^{ref}-derivation.

Definition 4 $\Delta' = A, B \vdash E : \sigma, \epsilon$ is a *completion* of $A_0 \vdash' e \leadsto E_0 : \theta, \epsilon$ if Δ' is derivable, $|A| = A_0$, $\text{dom}(B) = \text{frv}(E_0)$, $|E| = E_0$, and $|\sigma| = \theta$.

However, we are not interested in any completion, but in the "most static" one, so we set out to define an ordering on the set of completions of a particular derivation. First, we need an auxiliary definition to define greatest lower bounds of binding times, types, type assumptions, binding-time assumptions, and judgements.

Definition 5 1. For $b_1, b_2 \in BT$, $b_1 \sqcap b_2 = \begin{cases} b_1 & \text{if } b_1 = b_2 \vee b_1 < b_2 \\ b_2 & \text{otherwise.} \end{cases}$

2. Let σ_1, σ_2 be such that $|\sigma_1| = |\sigma_2|$. Define $\sigma_1 \sqcap \sigma_2$ inductively by

$$\begin{array}{lcl} \text{int}^{b_1} \sqcap \text{int}^{b_2} & = & \text{int}^{b_1 \sqcap b_2} \\ (\text{ref}_\rho^{b_1}\ \sigma_1) \sqcap (\text{ref}_\rho^{b_2}\ \sigma_2) & = & \text{ref}_\rho^{b_1 \sqcap b_2}\ (\sigma_1 \sqcap \sigma_2) \\ (\sigma_1 \xrightarrow{\epsilon}{}^{b_1} \sigma_1') \sqcap (\sigma_2 \xrightarrow{\epsilon}{}^{b_2} \sigma_2') & = & (\sigma_1 \sqcap \sigma_2) \xrightarrow{\epsilon}{}^{b_1 \sqcap b_2} (\sigma_1' \sqcap \sigma_2') \end{array}$$

3. For type assumptions A_1, A_2 with $|A_1| = |A_2|$ define $A_1 \sqcap A_2$ by:
$x : \sigma_1 \sqcap \sigma_2$ in $A_1 \sqcap A_2$ iff $x : \sigma_1$ in A_1 and $x : \sigma_2$ in A_2.

4. For binding-time assumptions B_1, B_2 with $\text{dom}(B_1) = \text{dom}(B_2)$ define

$$B_1 \sqcap B_2 = \{\rho : B_1(\rho) \sqcap B_2(\rho) \mid \rho \in \text{dom}(B_1)\}.$$

5. For judgements $\Delta' = A', B' \vdash E' : \sigma', \epsilon$ and $\Delta'' = A'', B'' \vdash E'' : \sigma'', \epsilon$ which are completions of $\Delta = A_0 \vdash' e \leadsto E_0 : \theta, \epsilon$, define the judgement $\Delta' \sqcap \Delta''$ by induction on Δ and the number of lift-expressions in E' and E''.

   $\Delta' \sqcap \Delta''$ has the form $A' \sqcap A'', B' \sqcap B'' \vdash E : \sigma' \sqcap \sigma'', \epsilon$ (all this is well-defined, because Δ' and Δ'' are completions of Δ) where is E is defined as follows:

   If $E' = \text{lift}\ E_1'$ and $E'' = \text{lift}\ E_1''$ then $\sigma' = \sigma'' = \text{int}^D$ and $E = \text{lift}\ E_1$ where E_1 is determined inductively by the result of $(A', B' \vdash E_1' : \text{int}^S, \epsilon) \sqcap (A'', B'' \vdash E_1'' : \text{int}^S, \epsilon) = (A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^S, \epsilon)$.

   If $E' = \text{lift}\ E_1'$, $E'' \neq \text{lift}\ E_1''$, $\sigma' = \text{int}^D$ and $\sigma'' = \text{int}^D$ then $E = \text{lift}\ E_1$ where E_1 is determined by the result of $(A', B' \vdash E_1' : \text{int}^S, \epsilon) \sqcap (A'', B'' \vdash E'' : \text{int}^D, \epsilon) = (A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^S, \epsilon)$.

If $E' = \text{lift } E'_1$, $E'' \neq \text{lift } E''_1$, $\sigma' = \text{int}^D$ and $\sigma'' = \text{int}^S$ then $E = E_1$ where E_1 is determined by the result of $(A', B' \vdash E'_1 : \text{int}^S, \epsilon) \sqcap (A'', B'' \vdash E'' : \text{int}^S, \epsilon) = (A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^S, \epsilon)$.

If $E' \neq \text{lift } E'_1$ and $E'' = \text{lift } E''_1$: analogous to the previous two cases.

For the remaining cases, we can assume that neither E' nor E'' is a lift-expression. They are straightforward and we present just two samples. Basically, if b' and b'' are the top-level annotations in E' and E'', we need to apply the same syntax constructor annotated with $b' \sqcap b''$ to the expressions that we obtain inductively.

If $E' = x$ then $E'' = x$ and $E = x$, too.

If $E' = \text{succ}^{b'}\ E'_1$ and $E'' = \text{succ}^{b''}\ E''_1$ then $E = \text{succ}^{b' \sqcap b''}\ E_1$ where E_1 is obtained by induction as above.

Lemma 3 *The greatest lower bound operations on binding times, types, type assumptions, binding-time assumptions, and judgements are commutative and associative.*

Proof: Straightforward, with a tedious case analysis in the case for judgements involving lift-expressions.

In addition, well-formedness of types and type assumptions is not affected by taking greatest lower bounds.

Lemma 4 *Suppose* $\text{dom}(B_1) = \text{dom}(B_2)$.

1. *For all* $\theta = |\sigma_1| = |\sigma_2|$ *with* $frv(\theta) \subseteq \text{dom}(B_1)$, *if* $B_1 \vdash \sigma_1$ wft *and* $B_2 \vdash \sigma_2$ wft *then* $B_1 \sqcap B_2 \vdash \sigma_1 \sqcap \sigma_2$ wft.
2. *For all* $A_0 = |A_1| = |A_2|$ *with* $frv(A_0) \subseteq \text{dom}(B_1)$, *if* $B_1 \vdash A_1$ wft *and* $B_2 \vdash A_2$ wft *then* $B_1 \sqcap B_2 \vdash A_1 \sqcap A_2$ wft.

Proof:
1. Induction on θ.
2. Induction on the size of A_0, then apply part 1.

Next we can prove that the greatest lower bound of two completions is itself a completion, which includes its derivability.

Lemma 5 *If* Δ' *and* Δ'' *are completions of some* Λ_r^{ref}*-judgement* Δ *then* $\Delta' \sqcap \Delta''$ *is a completion of* Δ.

Proof: By induction on the definition of $\Delta' \sqcap \Delta''$.

Let $\Delta' = A', B' \vdash E' : \sigma', \epsilon$, $\Delta'' = A'', B'' \vdash E'' : \sigma'', \epsilon$, $\Delta = A_0 \vdash' e \rightsquigarrow E_0 : \theta, \epsilon$, and $\Delta' \sqcap \Delta'' = A' \sqcap A'', B' \sqcap B'' \vdash E : \sigma' \sqcap \sigma'', \epsilon$.

Now we follow the cases of the definition:

- If $E' = \text{lift } E'_1$ and $E'' = \text{lift } E''_1$ then $\sigma' = \sigma'' = \text{int}^D$ and $E = \text{lift } E_1$ where E_1 is determined by the result of $(A', B' \vdash E'_1 : \text{int}^S, \epsilon) \sqcap (A'', B'' \vdash E''_1 : \text{int}^S, \epsilon) = (A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^S, \epsilon)$.
  By induction, this judgement is derivable and we apply (b-lift) to obtain $A' \sqcap A'', B' \sqcap B'' \vdash \text{lift } E_1 : \text{int}^D, \epsilon)$, but that is exactly $\Delta' \sqcap \Delta''$.
  Trivially, $|E| = |\text{lift } E_1| = E_0$.

- The case $E' = \text{lift } E_1'$, $E'' \neq \text{lift } E_1''$, $\sigma' = \text{int}^D$ and $\sigma'' = \text{int}^D$ works by exactly the same reasoning.
- If $E' = \text{lift } E_1'$, $E'' \neq \text{lift } E_1''$, $\sigma' = \text{int}^D$ and $\sigma'' = \text{int}^S$ then $E = E_1$ where E_1 is determined by the result of $(A', B' \vdash E_1' : \text{int}^S, \epsilon) \sqcap (A'', B'' \vdash E'' : \text{int}^S, \epsilon) = (A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^S, \epsilon)$.

  Again, by induction this judgement is derivable, but this time it is already equal to $\Delta' \sqcap \Delta''$.

  Trivially, $|E| = |E_1| = E_0$.
- If $E' = x$ then $E'' = x$ and $E = x$, too.

  By assumption, $A', B' \vdash x : \sigma', \epsilon$ which must be due to rule (b-var). Hence $x : \sigma'$ in A' and $\epsilon = \emptyset$. By the same reasoning, $x : \sigma''$ in A'' so that $x : \sigma' \sqcap \sigma''$ in $A' \sqcap A''$, by definition. Since $B' \sqcap B'' \vdash A' \sqcap A''$ wft by Lemma 4, (b-var) proves $A' \sqcap A'', B' \sqcap B'' \vdash x : \sigma' \sqcap \sigma'', \emptyset$.

  Trivially, $|E| = x = E_0$.
- If $E' = \text{succ}^{b'}\ E_1'$ and $E'' = \text{succ}^{b''}\ E_1''$ then $E = \text{succ}^{b' \sqcap b''}\ E_1$ where E_1 is obtained as indicated in the definition of $\sqcap$.

  In this case, we have $\sigma' = \text{int}^{b'}$, $\sigma'' = \text{int}^{b''}$, and E_0 is succ E_0'. The last rule in both cases must have been (b-succ) applied to $A', B' \vdash E_1' : \text{int}^{b'}, \epsilon$ and $A'', B'' \vdash E_1'' : \text{int}^{b''}, \epsilon$, respectively.

  The greatest lower bound of these is by definition $A' \sqcap A'', B' \sqcap B'' \vdash E_1 : \text{int}^{b' \sqcap b''}, \epsilon$ and it is a completion of E_0', by induction. Applying (b-succ) to it yields $A' \sqcap A'', B' \sqcap B'' \vdash \text{succ}^{b' \sqcap b''}\ E_1 : \text{int}^{b' \sqcap b''}, \epsilon$ which is exactly $\Delta' \sqcap \Delta''$.

  Furthermore, $|E_1| = E_0'$ by induction, so $|E| = |\text{succ}^{b' \sqcap b''}\ E_1| = \text{succ } E_0' = E_0$.
- The remaining cases are essentially straightforward appeals to the inductive hypothesis.

Thus armed, we set out to show that each Λ_r^{ref} derivation Δ can be completed to a $\Lambda_{r,\,ann}^{ref}$-derivation. Furthermore, the set of completions of Δ forms a finite lower semi-lattice, i.e., a partial order with greatest lower bounds and a smallest element.

Proposition 1 *Suppose the translation judgement $\Delta = A_0 \vdash' e \rightsquigarrow E_0 : \theta, \epsilon$ is derivable.*

1. *There exist A, B, E, and σ such that $A, B \vdash E : \sigma, \epsilon$ is a completion of Δ.*
2. *Define the relation $\leq$ on the set of completions of Δ by $\Delta' \leq \Delta''$ iff $\Delta' \sqcap \Delta'' = \Delta'$. This relation is a partial order.*
3. *The partial order $\leq$ on the set of completions of Δ has a smallest element.*

Proof: 1. Define A such that for each $x_i : \theta_i$ in A_0 there is $x_i : \sigma_i$ in A with $\theta_i = |\sigma_i|$ and all annotations in σ_i are D. Likewise for σ. For each free region variable ρ in E_0 there is $\rho : D$ in B. Obviously, $B \vdash A$ wft and $B \vdash \sigma$ wft. Define E such that $|E| = E_0$, all annotations in E are D, and a lift is placed on every constant. This assignment fulfills all well-formedness requirements, and by induction on the derivation of Δ we can prove that $A, B \vdash E : \sigma, \epsilon$ is derivable and conforms to Δ.

2. We have to show that $\leq$ is reflexive, transitive, and antisymmetric. Reflexivity is trivial by definition of $\sqcap$. Transitivity is immediate using the definition of $\leq$ and the associativity of $\sqcap$. Antisymmetry follows by commutativity of $\sqcap$.
3. Since BT and E_0 are finite, there is only a finite number of completions of Δ. Since the set of completions is non-empty by part 1 and it is closed under greatest lower bounds $\sqcap$, the greatest lower bound of *all* completions is a completion which is $\leq$ to every other completion.

This result sets our goal: The algorithm should find the annotation corresponding to the smallest completion of a fixed Λ_r^{ref}-derivation.

4.9 Binding-Time Reconstruction

In this section, we use the well-formedness constraints on the binding-time annotation to determine the smallest possible annotation. This reconstruction of the binding-time annotations boils down to solving inequations over natural numbers.

To compute the minimal well-formed binding-time decoration for the derivation of a translation judgement $\Delta = A_0 \vdash' e \rightsquigarrow E_0 : \theta, \epsilon$, we first attach a binding-time variable to each construct of E_0 and to each type constructor mentioned in the derivation of Δ. Binding-time variables range over BT. The type equalities imposed by the inference rules imply equalities between binding-time variables. These equalities are resolved by unification during type inference. The well-formedness constraints in the decorated region inference rules give rise to a system of inequations between binding-time variables. Such a system can be solved in time linear in the number of inequations [68]. To facilitate the insertion of lift-expressions, expressions of type int carry two binding-time variables b_1 and b_2 constrained by $b_1 \leq b_2$, as proposed by Henglein [40].

The number of inequations is at most quadratic in n, the size of the expression e, because we need at most $2 + n$ inequations for each type constructor and there is a (term graph) representation of all types mentioned in Δ that needs at most n type constructor nodes [40]. Hence, the worst case time bound for this step is $O(n^2)$.

5 Implementation Issues

The implementation of the specializer is mostly straightforward and follows the specialization semantics given in Sec. 3. This section deals with two implementation issues that we have ignored so far: program point specialization and static store management.

5.1 Program Point Specialization

Program point specialization is an essential feature of realistic specializers [2,8,9,17,51,58]. It implements the fold and define rules that are well-known from program transformation [16], thus avoiding code duplication and many cases of infinite specialization. Folding is the process of using already known definitions.

A program point specializer tries to fold at *specialization points*, marked by "memo E" in the annotated language. How to fold is determined by the *static fingerprint*, a projection of the current state of the specializer. Compared to the other specializers, we have an additional problem with computing the static fingerprint due to the presence of sharing and cycles in the store. The static fingerprint of $\mathcal{S}[\![\text{memo } E]\!]\rho k\sigma$ depends on ρ and the reachability graph of the store, $R(\sigma)$, which is defined as follows:

$R(\sigma)$ is a labeled digraph. Its nodes are the used locations of the store $\{\alpha \mid \sigma(\alpha) \neq \text{unused}\}$ and the label of node α is $\sigma(\alpha)$. There is an edge $\alpha \to \alpha'$ iff $\sigma(\alpha)$ contains the address α'.

The static fingerprint of $\mathcal{S}[\![\text{memo } E]\!]\rho k\sigma$ consists of

- the static values in the environment ρ and
- the statically reachable part of the store σ, i.e., the subgraph of the graph $R(\sigma)$ that is reachable from the static references in the environment.

Whenever the specializer encounters a specialization point "memo E" it creates a residual procedure call. The current static fingerprint determines which residual procedure is called. When the specializer encounters a particular fingerprint for the first time it creates a new procedure definition and specializes E to construct its body. The new definition is cached and indexed with the fingerprint. The next time, the specializer only constructs the residual procedure call and adjusts the static store according to the effect that specializing E would have.

The implementation uses a linearized representation of the reachable static store, which is constructed during a depth-first traversal of $R(\sigma)$. This representation only reflects the structure of the graph, but not the actual store addresses. Otherwise it would be virtually impossible to encounter the same fingerprint again. Therefore, the address used in a fingerprint is the depth-first number of the node in the traversal of $R(\sigma)$.

The binding time of a specialization point memo E is dynamic, because the specializer generates a residual function call for it. The specializer processes the body E with the empty continuation $\lambda y.\lambda\sigma.y$ and—in contrast to the dynamic abstraction—the static store at the memo E expression. In addition, it stores a log of the static side effects performed while specializing E.

This log causes a problem, because it is only complete after the specializer is finished with E. However, the specializer may recursively try to process the same specialization point with respect to the same static fingerprint before the log for this specialization point is complete. There are basically two solutions to this problem.

1. The online strategy [58] "freezes" the log when it encounters such a recursive call. Then it proceeds with its current contents. The specializer signals an error on an attempt to modify the "frozen" log later on.
2. The offline strategy disallows static write and init effects at specialization points.

Our system implements the offline strategy. It avoids unexpected errors at specialization time and the binding-time analysis can easily enforce it. More sophisticated offline strategies are possible, but we have yet to encounter examples where they are required.

5.2 Static Store Management

The specializer duplicates the static store at a dynamic conditional (see Fig. 7). Taken literally, the specializer would hold on to a copy of the static store while specializing the then-branch and use the copy to specialize the else-branch afterwards (as in early versions of C-Mix [2]). This is clearly inefficient.

In our implementation, a stack (which happens to be identical to the above-mentioned log) keeps track of all modifications, similar to the trail stack in a Prolog implementation. Every static assignment pushes the address α and the previous contents $\sigma(\alpha)$ on the stack. At the dynamic conditional, the specializer pushes a mark on the stack before entering the then-branch. After this specialization is finished and before specializing the else-branch, the specializer pops and undoes all entries in the stack up to (and including) the mark. Thus it establishes a static store which is equivalent to the static store before entering the then-branch.

6 Applications

In preceding sections, we have seen that two factors aggravate the problem of performing assignments at specialization time with respect to a traditional imperative language: first-class references complicate the data flow and higher-order functions complicate the control flow. With the applications below, we show that our specializer handles both in a satisfactory manner. None of the existing specializers for Scheme and ML [7, 9, 51] can satisfactorily specialize these examples in the presented form.

The first application demonstrates program point specialization with respect to cyclic data structures. All previous program point specializers explicitly forbid cyclic data since it results in non-termination. Specializers for traditional languages avoid this problem by restricting the use of pointers. The second application is the specialization of DAG unification where variables are implemented by pointers. To achieve similar results with another specializer, requires major rewriting of the source program [19]. In contrast, our source program is straightforward and needs only slight modifications to achieve good specialization. For the third application we specialize an interpreter for a first-order lazy functional language which implements updatable closures using thunks and references. The result is online specialization for the lazy language. This is a particular instance of specializer generation [33] which has not been achieved before. For this example, the ability of the specializer to process thunks and references at specialization time is crucial.

The examples use a non-standard construct to declare algebraic datatypes. For example

```
(define-data xlist (xnil) (xcons xcar xcdr))
```

defines a new datatype `xlist` with a nullary constructor `xnil` and a binary constructor `xcons` with selectors `xcar` and `xcdr`.

6.1 Cyclic Data

The program below constructs a cyclic list of ones and combines it with an unknown list `d`. The function `main` maps a list (x_1 ... x_n) to the list of pairs ((x_1 . 1) ... (x_n . 1)).

```
(define-data xlist (xnil) (xcons xcar xcdr))
(define (main d)
  (let ((cycle (xcons 1 (make-cell (xnil)))))
    (cell-set! (xcdr cycle) cycle)
    (zip d cycle)))
(define (zip d s)
  (if (null? d)
      '()
      (cons (cons (car d) (xcar s))
            (zip (cdr d) (cell-ref (xcdr s))))))
```

The binding-time analysis determines that all operations on references can be performed at specialization time, given that `d` is dynamic. Specializing the function `zip` requires taking the static fingerprint of the cyclic data structure bound to `cycle`. The resulting residual program is:

```
(define ($goal-1 yyy-1)
  (define (zip-4-2 yyy-1-1)
    (let ((mlet-2 (null? yyy-1-1)))
      (if mlet-2
        '()
        (let* ((mlet-5 (car yyy-1-1))
               (mlet-4 (cons mlet-5 1))
               (mlet-6 (cdr yyy-1-1))
               (mlet-7 (zip-4-2 mlet-6)))
          (cons mlet-4 mlet-7)))))
  (zip-4-2 yyy-1))
```

The specializer terminates despite the cyclic structure which vanishes on specialization. The construction of the pair `(x . 1)` is implemented by `(cons mlet-5 1)`.

6.2 Unification

We have implemented a unification algorithm `(unify s t)` which works on terms where variables are implemented by references:

```
(define-data maybe
  (just one)
  (nothing))
(define-data term
  (make-var ref)                       ; make-var : ref (maybe term) -> term
  (make-cst num)
  (make-bin term1 term2)
  (make-dyn dynterm))
```

A value of type `maybe X` is either `(nothing)` or `(just X)`, where `X` has type `X`. A variable is represented by `(make-var ref)` where `ref` is a reference which either contains `(nothing)` (an unbound variable) or a binding `(just term)` where `term` is a term. A constant is simply `(make-cst num)` where `num` is a number. The single binary constructor `make-bin` applies to two terms. Finally, a term may also be a `dynterm` which is explained below.

The following code is written for `s` static and `t` dynamic.

```
(define (unify s t)
  (cond
   ((make-var? s)
    (let ((ref-maybe-s (ref s)))
      (if (just? (cell-ref ref-maybe-s))
          (unify (one (cell-ref ref-maybe-s)) t)
          (begin
            (cell-set! ref-maybe-s (just (make-dyn t)))
            SUCCESS))))
   ((make-cst? s)
    (cond ((make-var? t)
           (let ((ref-maybe-t (ref t)))
```

```
              (if (just? (cell-ref ref-maybe-t))
                  (unify s (one (cell-ref ref-maybe-t)))
                  (begin
                    (cell-set! ref-maybe-t (just (make-cst (num s))))
                    SUCCESS))))
           ((make-cst? t)
            (= (num s) (num t)))
           (else
            FAIL)))
    ((make-bin? s)
     (cond ((make-var? t)
            (let ((ref-maybe-t (ref t)))
              (if (just? (cell-ref ref-maybe-t))
                  (unify s (one (cell-ref ref-maybe-t)))
                  (begin
                    (cell-set! ref-maybe-t (just (make-bin (coerce (term1 s))
                                                           (coerce (term2 s)))))
                    SUCCESS))))
           ((make-bin? t)
            (and (unify (term1 s) (term1 t))
                 (unify (term2 s) (term2 t))))
           (else
            FAIL)))
    ((make-dyn? s)
     (dynamic-unify (dynterm s) t))
    (else
     FAIL)))
```

`SUCCESS` and `FAIL` are just names for `#t` and `#f`. We consider the case where `s` is `make-cst` in depth.

If the dynamic term `t` is a variable then the code checks (dynamically) whether the variable is bound. If it is bound it recursively unifies `s` with the term bound to the variable. If the variable is not bound then it is dynamically bound to the constant term and `SUCCESS` is returned.

Otherwise, if `t` is a constant then the values of the constants are compared and the result of the comparison is the result of the unification. Otherwise, for any other kind of term, failure is signalled.

The `make-dyn` constructor appears in static terms when a static variable is bound to a dynamic term. Consequently, if `s` is such a dynamic term, `unify` extracts the term and dynamically unifies it with `t`. Therefore, a completely dynamic copy of `unify` is necessary. This duplication of `unify` (not shown) is a standard binding-time improvement, which can be automated [75].

Finally, `coerce` transforms a static term into a dynamic copy of the same term. It is necessary whenever a static term is bound to a dynamic variable. If the code would simply assign the static term to the dynamic variable the binding-time analysis would classify the intended static term as dynamic. Therefore, `coerce` first copies the static term and the code assigns the copy to the variable. The copy procedure `coerce` serves as a binding-time coercion because its argument term is static and its result term is dynamic. `Coerce` must implement a graph copy algorithm to preserve sharing.

```
(define ($goal-1 yyy-1)
  (define (unify_1-56-2 yyy-1-1)
    (let ((mlet-2 (make-var? yyy-1-1)))
      (if mlet-2
        (let* ((mlet-3 (ref yyy-1-1))
               (mlet-4 (cell-ref mlet-3)))
          (unify_1-57-3 mlet-4 mlet-3))
        (unify_1-58-4 yyy-1-1))))
  (define (unify_1-58-4 yyy-1-1-1)
    (let ((mlet-2 (make-bin? yyy-1-1-1)))
      (if mlet-2 (unify_1-59-5 yyy-1-1-1) #f)))
  (define (unify_1-59-5 yyy-1-1-1-1)
    (let* ((mlet-2 (term1 yyy-1-1-1-1))
           (mlet-3 (term2 yyy-1-1-1-1)))
      (dynamic-unify mlet-2 mlet-3)))
  (define (unify_1-57-3 mlet-4-2 mlet-3-1)
    (let ((mlet-3 (just? mlet-4-2)))
      (if mlet-3
        (let ((mlet-4 (one mlet-4-2)))
          (unify_1-56-2 mlet-4))
        (let* ((mlet-10 (nothing))
               (mlet-9 (make-cell mlet-10))
               (mlet-11 (make-var mlet-9))
               (mlet-12 (make-var mlet-9))
               (mlet-8 (make-bin mlet-11 mlet-12))
               (mlet-7 (just mlet-8))
               (mlet-6 (cell-set! mlet-3-1 mlet-7)))
          #t))))
  (unify_1-56-2 yyy-1))
```

Figure 16: `unify` specialized with respect to `(make-bin (make-var r) (make-var r))`

Our binding-time analysis annotates all operations on static terms `s` as static. Only the operations depending on `t` are classified dynamic.

We have specialized this algorithm with respect to various terms `s`. In each case the specializer is able to perform all operations that involve the processing of `s`. It handles non-linear variables correctly due to the fact that `coerce` preserves sharing. Figure 16 shows the result of specializing `unify` with respect to `(make-bin (make-var r) (make-var r))` where the variable is not bound. The function `unify_1-59-5` handles the non-linear occurrence of the variable `(make-var r)`. The last part of function `unify_1-57-3` constructs a dynamic version of the static input term, again respecting sharing (viz. `mlet-9`).

The speedup obtained by specialization of `unify` varies between 1.65 and 2.25. This is a good result for a realistic algorithm.

6.3 Specializer Generation

Specializer generation is a challenging application [32,33]. Basically, it allows the automatic construction of a specializer from a suitable interpreter. This is an extension of the classic application of partial evaluation to compilation. The theoretical foundations are the specializer projections [32], which are generalizations of Futamura's projections (which show how to achieve compilation) [31].

Using the specializer projections we have generated an online specializer for a lazy first-order functional programming language from a two-level interpreter. A two-level interpreter accepts its input data in two parts. The first part is considered the known part of the input while the second part is unknown. The first part is put in a standard environment, while the second part is put into a *configuration environment*. The known inputs may refer to the unknown inputs through *configuration variables* (pointers to unknown input). The interpreter represents values using the datatype `desc`:

```
(define-data desc
  (const const->value)
  (cvar  cvar->number)
  (static-susp static-susp->ref-sum)    ; (value + unit -> value) ref
  (dyn-susp dyn-static->ref-sum dyn-susp->ref-sum))
(define-data sum
  (make-value sum->value)
  (make-thunk sum->thunk))
```

The interpreter tries to perform a computation with the known values first before it backtracks and uses unknown values from the configuration environment. A value can be a constant (`const`), a configuration variable (`cvar`) that points into the dynamic configuration environment, a static suspension, or a dynamic suspension. A static suspension is implemented by a reference that either holds the value (if it was already computed) or a thunk that is invoked when the value is first demanded. The interpreter generates a dynamic suspension if it is not able to determine *a priori* whether the value of the suspension will be known or unknown. Therefore, the `dyn-susp` contains both, a static and a dynamic suspension. The interpreter first tries the static one and only falls back to using the dynamic one if the static suspension fails to deliver a result.

The binding-time analysis classifies all references and thunks in the interpreter as static. This is impossible to achieve with an overly conservative partial evaluator like Similix. The specialized programs have the property that the value of each source expression of the program is computed at most once.

In our experiments, we have found that the speedup due to specialization varies extremely depending on the particular first-order program. We have measured speedups between 70 and 150, which reflects the massive overhead in the two-level interpreter.

For example, specialization of the program (first-order recursive equations)

```
(f x y z = (g x (+ y z) (- y z)))
(g x y z = (if x (+ y y) (* z z)))
```

with respect to `(f` cv_1 cv_2 `13)` (where cv_i denotes a configuration variable) yields the following residual program.

```
(define ($goal-1 dyn-1)
  (define (dyn-eval_1-11-2 mlet-4-6 mlet-3-5 mlet-4-4 mlet-3-3 mlet-4-2 mlet-3-1)
    (if mlet-3-1
      (let ((mlet-7 (+ mlet-4-4 13)))
        (+ mlet-7 mlet-7))
      (let ((mlet-9 (- mlet-4-6 13)))
        (* mlet-9 mlet-9))))
```

```
  (let* ((mlet-2 (cdr dyn-1))
         (mlet-3 (car dyn-1))
         (mlet-4 (car mlet-2)))
    (dyn-eval_1-11-2 mlet-4 mlet-3 mlet-4 mlet-3 mlet-4 mlet-3)))
```

The parameter `dyn` represents the configuration environment. It is a list of the values of the configuration variables. So `mlet-3` and `mlet-4` hold the values of cv_1 and cv_2, respectively. It is clearly visible that each operation in the source program is executed at most once in the residual program. In addition, the known value `13` is propagated to its uses.

If the input additionally specifies the value of the condition, i.e., `(f #f` cv_2 `13)` the residual program reduces to one slice of the above residual program.

```
(define ($goal-1 dyn-1)
  (let* ((mlet-2 (car dyn-1))
         (mlet-3 (- mlet-2 13)))
    (* mlet-3 mlet-3)))
```

7 Related Work

7.1 Online Specialization

The early work on specialization mostly considers online specialization for imperative languages. For example, Futamura [31] and Ershov and his group [14, 28, 29] consider fragments of Algol. Building on work by Ershov's group, Meyer [53] defines and proves correct an online specializer for Pascal that performs side effects at specialization time. Marquard and Steensgaard [52] describe an online partial evaluator for an object-oriented imperative language. The REDFUN group [6, 37] developed specializers for impure Lisp using an ad-hoc approach to handling side effects. Ørbæk's POPE [64] specializes Scheme with mutable variables, but always residualizes operations that affect variables. Asai and others [4] describe an online specializer for a subset of Scheme which handles dynamic side effects using pre-actions. Pre-actions have a purpose similar to automatically inserted let-expressions. Their specializer does not handle side effects at specialization time. There are also versions of the FUSE specializer [77] which deal with assignments.

7.2 Offline Specialization

The first offline specializer for an imperative flow-chart language is flow-chart mix [36]. A similar language is studied by Das and others [24] who provide a semantic notion of binding-time analysis for a tiny imperative language based on a notion of value sequences. Today there are partial evaluators for realistic languages, for example C-mix and Tempo for a subset of ANSI C [2, 20] , F-spec for a subset of Fortran 77 [5], and M2MIX for Modula-2 [15]. All of them perform some side effects statically, but all of them deal with first-order languages and some do not include pointers (flow-chart mix and F-spec). In contrast, we specialize a higher-order language with first-class mutable references.

The system of Nirkhe and Pugh [58] is interesting, because it requires programming in an annotated language. They describe partial evaluation for a block-structured imperative language. They perform full memoization, but using the online technique described in Sec. 5.1. Furthermore, they cut down the static fingerprint by only including those parts of the store that are actually read. This feature could also be included in our specializer using an additional analysis or by starting from a polymorphically typed language.

Like our system, recent versions of C-Mix duplicate the store only conceptually at dynamic conditionals and include sophisticated static memory management techniques [3]. Bulyonkov and Kochetov [15] define a static analysis that determines the part of the store that is affected by specialization of the then-branch. The result of this analysis can drive a more efficient partial save/restore scheme than ours. Our approach to static memory management draws on ideas from the implementation of first-class stores [25, 43, 56]. A language supporting first-class stores provides operations to reify the current store as a first-class object and to install such a store again as the current one later in the computation.

The specializers of Bondorf and Danvy [9, 12] deal with first-order and higher-order recursive equations with global variables. Neither specializer performs side effects at specialization time. SML-mix [7] and Pell-Mell [51] employ the same conservative strategy for partial evaluation of SML. Pell-Mell wraps dynamic computations in dynamic let-expressions to obtain "lightweight symbolic values." This is similar to the introduction of let-expressions in our specializer.

7.3 Specialization with Continuations

Continuation-based partial evaluation started of with Consel and Danvy's improvement of static data flow by CPS-transforming the source program before specializing it [18]. Bondorf [10] avoids CPS in residual programs by writing the specializer itself using continuations and liberalizing the binding-time analysis in the same way as Consel and Danvy. Lawall and Danvy [48] reexpress Bondorf's specializer in direct style plus control operators to gain efficiency. Moura, Consel, and Lawall [57] suggest an approach to static analysis of imperative programs by transforming them to a sophisticated variant of store-passing style (and to static single assignment form [21]) and apply analysis techniques developed for pure functional programs. This parallels Consel and Danvy's use of the CPS transformation [18]. In our work we use Consel and Danvy's approach as a proof device. In practice, our specializer rephrases Bondorf's specializer [10] in the style of Consel and Danvy [18] using A-normal form instead of CPS for the residual code and, of course, adding store passing. Our actual implementation generalizes Lawall and Danvy's direct-style specializer.

Interestingly, Hatcliff and Danvy [39] have specified and proved correct a partial evaluator that performs let-insertion automatically. Their let-insertion results from a preceding transformation into Moggi's computational metalanguage [54] whereas our specializer does it on the fly. Another difference is that their specializer only propagates static contexts whereas our specializer propagates dynamic contexts, too.

Our binding-time analysis is inspired by binding-time specifications using non-standard and annotated type systems [26, 34, 40]. The supporting analyses are based on effect systems [50, 70, 76].

7.4 Type-based Specialization

Hughes [42] has discovered a novel framework for specialization of typed higher-order languages. His type specialization is a variation of type inference and is thus able to overcome some limitations of partial evaluators for typed languages. Hughes and the authors [27] have extended type specialization to Moggi's computational metalanguage (instantiated to the state monad) so that it covers a similar range of applications as the present work. However, whereas the focus of the type specialization work lies on the exploitation of (and struggling with) the superior information flow granted by unification, the present work targets efficient partial evaluation for realistic languages. Due to the unification a type specializer constructs its output out of order, whereas our specializer adheres to the evaluation order. Furthermore, the partial evaluator presented here is fully automated: the binding-time analysis constructs well-annotated programs which the specializer processes without producing errors. In contrast, the type specializer must be fed an annotated program and it may report errors during specialization even on well-formed programs.

Type-directed partial evaluation [22] is also applicable to specialization with static state, in principle. However, if sum types are handled as indicated in the paper [22] then the static store will not be duplicated at dynamic conditionals. This is due to the use of versions of the control operators shift and reset that are ignorant of the store, leading to incorrect results. In addition, type-directed partial evaluation does not perform program point specialization.

8 Conclusions and Further Work

We have developed an offline partial evaluator for a call-by-value lambda calculus with first-class references. Although presented for a simple core language, the techniques scale up to full Scheme or Standard ML. It is straightforward to include partially static data, to construct program-generator generators (cogens), and to extend the specializer to multiple levels of binding times. All these extensions are implemented along with the techniques presented in this work in the PGG system which applies to the full Scheme language [71, 72, 74].

The specializer is applicable to programs that are otherwise hard to specialize in a satisfactory manner: Specializers like Similix explicitly disallow the construction of cyclic data structures; specializing unification without static references requires non-trivial rewriting of the algorithm [19]; specialization of programs in message-passing style is impossible without a proper treatment of static references as demonstrated in Fig. 1; specializer generation for lazy functional languages has not been achieved before.

Specialization time and the efficiency of the binding-time analysis were never a problem in practice. Preliminary experiments suggest that the analysis runs in linear time in the typical case, i.e., for programs that do not use side effects or that use them sparingly. We have never observed the worst-case $O(n^4)$ behavior in practice.

The specialization algorithm is based on continuation-passing and store-passing style. It generalizes continuation-based partial evaluation. The novelties are the treatment of the static store using store-passing style and the automatic let-insertion. In the implementation we have rephrased our algorithm in a similar way as Lawall and Danvy rephrased continuation-based partial evaluation [48]. Furthermore, we have some evidence that our approach can be generalized to deal with other computational effects, such as exceptions.

The novel element of our binding-time analysis is its reliance on a region inference system in place of a type system. We regard it as a natural extension of the type-based analyses that are used in other partial evaluators [13]. Our analysis can be regarded as improving the results of earlier analyses by exploiting region information. There is a correctness proof for the region-based binding-time analysis [73].

The main emphasis of our approach is on simplicity and efficiency. For these reasons we have chosen a monovariant, context-insensitive binding-time analysis. This yields a tractable polynomial analysis which is fast in practice. We are fully aware that there are more precise and more expensive analysis methods that exploit polyvariance or polymorphism, but the examples show that our approach already leads to satisfactory results. Our conclusion is that the additional precision of other analysis methods may not be required for many applications in mostly functional languages like Scheme or ML. In another context, for example to specialize C or Modula-2, different choices may be preferable.

The basic techniques presented in this work are generally applicable. For example, our techniques for analysis and static memory management are immediately applicable to any procedural language, i.e., Modula-2 or C. Beyond that, we believe that the analysis and implementation techniques can be extended to object-oriented programming languages.

Acknowledgements

This work was initiated during a visit of Dirk Dussart at Tübingen University, funded by a Ministerium für Wissenschaft und Forschung grant, in February 1996. It was further developed during his stay at BRICS, Aarhus, in 1996. Special thanks are due to Olivier Danvy, Simon Helsen, Julia Lawall, Torben Mogensen, Claus Reinke, and Michael Sperber for valuable feedback in various stages of this work and to S. Doaitse Swierstra for support.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Peter Holst Andersen. Static memory management in C-Mix. available at URL http://www.diku.dk/research-groups/topps/activities/cmix/memory.ps.gz, December 1996.

[4] Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial evaluation of call-by-value λ-calculus with side-effects. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 12–21, Amsterdam, The Netherlands, June 1997. ACM Press.

[5] Romana Baier, Robert Glück, and Robert Zöchling. Partial evaluation of numerical programs in Fortran. In Peter Sestoft and Harald Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 119–132, Orlando, Fla., June 1994. ACM.

[6] L. Beckman, A. Haraldsson, Ö. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[7] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.

[8] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '94)*, volume 844

of *Lecture Notes in Computer Science*, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.

[9] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[10] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, California, USA, June 1992.

[11] Anders Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, May 1993.

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

[13] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[14] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[15] Mikhail A. Bulyonkov and Dmitrij V. Kochetov. Practical aspects of specialization of Algol-like programs. In Danvy et al. [23], pages 17–32.

[16] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[17] Charles Consel. Polyvariant binding-time analysis for applicative languages. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.

[18] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [41], pages 496–519.

[19] Charles Consel and Siau-Cheng Khoo. Semantics directed generation of a Prolog compiler. In Jan Maluszynski and Martin Wirsing, editors, *Proc. Programming Language Implementation and Logic Programming '91*, pages 135–146, Passau, Germany, August 1991. Springer-Verlag. LNCS 528.

[20] Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. In POPL1996 [63], pages 145–156.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[22] Olivier Danvy. Type-directed partial evaluation. In POPL1996 [63], pages 242–257.

[23] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, volume 1110 of *Lecture Notes in Computer Science*, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[24] Manuvir Das, Thomas Reps, and Pascal Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 100–110, La Jolla, CA, June 1995. ACM Press.

[25] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth ACM Symposium on Theory of Computing*, pages 109–121, May 1986.

[26] 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.

[27] Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, pages 204–216, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[28] Andrei P. Ershov. On the essence of compilation. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[29] Andrei P. Ershov. On mixed computation: Informal account of the strict and polyvariant computational schemes. In Manfred Broy, editor, *Control Flow and Data Flow: Consepts of Distributed Programming*, pages 107–120. Springer-Verlag, 1984.

[30] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[31] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[32] Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, October 1994.

[33] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[34] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 282–287, Nice, France, 1990. ACM Press.

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

[36] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: A case study. *Structured Programming*, 12:123–144, 1991.

[37] Anders Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[38] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In POPL1994 [62], pages 458–471.

[39] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[40] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [41], pages 448–472.

[41] John Hughes, editor. *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Cambridge, MA, 1991. Springer-Verlag.

[42] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [23], pages 183–215.

[43] G. F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In POPL1988 [61], pages 158–168.

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

[45] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.

[46] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language scheme. Technical report, 1998.

[47] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.

[48] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.

[49] Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.

[50] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In POPL1988 [61], pages 47–57.

[51] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 112–119, Orlando, Florida, June 1994.

[52] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, April 1992.

[53] Uwe Meyer. Techniques for partial evaluation of imperative languages. In Paul Hudak and Neil D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 94–105, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).

[54] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of the 4th Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, June 1989. IEEE Computer Society Press.

[55] Eugenio Moggi. Functor categories and two-level languages. In M. Nivat and A. Arnold, editors, *Foundations of Software Science and Computation Structures, FoSSaCS'98*, Lecture Notes in Computer Science, Lisbon, Portugal, April 1998.

[56] J. Gregory Morrisett. Generalizing first-class stores. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, pages 73–87, Copenhagen, Denmark, June 1993. Yale University, Department of Computer Science, New Haven, CT. Technical Report YALEU/DCS/RR-968.

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

[58] Vivek Nirkhe and William Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 269–279, Albuquerque, New Mexico, January 1992. ACM Press.

[59] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. Conference on Programming Language Design and Implementation '88*, pages 199–208, Atlanta, July 1988. ACM.

[60] Gordon D. Plotkin. $\mathbf{T}^\omega$ as a universal domain. *Journal of Computer and System Sciences*, 17:209–236, 1978.

[61] *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.

[62] *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OG, January 1994. ACM Press.

[63] *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Fla., January 1996. ACM Press.

[64] Peter Ørbæk. POPE: An on-line partial evaluator. ftp://ftp.daimi.aau.dk/pub/empl/poe/pope.ps.gz, June 1994.

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

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

[67] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[68] Helmut Seidl. Least solutions of equations over $\mathcal{N}$. In *Proc. International Conference of Automata, Languages and Programming, ICALP '94*, volume 820 of *Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, 1994.

[69] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 215–225, Las Vegas, NV, USA, June 1997. ACM Press.

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

[71] Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.

[72] Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.

[73] Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: http://www.elsevier.nl/locate/entcs/volume6.html.

[74] Peter Thiemann. *The PGG System—User Manual.* University of Nottingham, Nottingham, England, June 1998. Available from `ftp://ftp.informatik.uni-tuebingen.de/pub/PU/thiemann/software/pgg/`.

[75] Peter Thiemann and Michael Sperber. Polyvariant expansion and compiler generators. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 285–296, Novosibirsk, Russia, June 1996. Springer-Verlag.

[76] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In POPL1994 [62], pages 188–201.

[77] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In Hughes [41], pages 165–191.