Improving Binding Times without Explicit CPS-Conversion

Anders Bondorf *
DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: anders@diku.dk

Abstract

A major obstacle in partial evaluation (program specialization) is the need for *binding time improvements* [HH90]. By reorganizing a source program, the residual programs obtained by specializing the source program may be improved: more computations can be done statically, that is, at specialization time.

One well-known effective reorganization is (manual or automatic) conversion into *continuation passing style* (cps) [Dan91, Jor90, HG91, KS91, CD91, Bon91b, Jor92]. This conversion allows data *consumers* to be propagated through *frozen* expressions to the data *producers*. In this paper we show how such improvements can be obtained without affecting the source program: *by writing the program specializer itself in cps*; traditionally, specialization has been formulated in direct style.

The advantages of avoiding cps-converting source programs are: (1) no cps-conversion phase is needed; (2) the generated residual programs are not in cps; (3) since no source level continuations are added, there is no overhead of manipulating closure representations in the generating extensions (e.g. compilers) obtained by self-application; (4) manual "binding time debugging" is easier since binding time analysis is done on a non-converted program.

We have implemented a cps-based program specializer; it is integrated in the partial evaluator Similix 4.0 [Bon91b].

Using a cps-specializer, *partially static data structures* [Mog88] can be handled safely in a straightforward way. The difficulty is to ensure automatically that residual expressions that become part of a partially static data structure are neither duplicated nor discarded. This is achieved by binding such residual expressions in automatically inserted frozen let-expressions; cps is needed to propagate operations on the partially static data structure through these frozen let-expressions. Based on this idea, we have implemented an extension of Similix 4.0 that handles partially static data structures.

*This work was supported by ESPRIT Basic Research Actions project 3124 "Semantique"

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0001...$1.50

1 Introduction

When using a partial evaluator, a small seemingly innocent change in a source program may have severe effects on the quality of the residual programs obtained by partially evaluating the source program. It thus often occurs that if the source program is written in one way, the specializer happily performs some computation statically — while if the program is formulated in a slightly different way, the specializer has to suspend that computation, leaving it in the residual program.

For an example, consider the following typical piece of code (written in Scheme syntax):

```
(f (let ((val (evalu E ...)))                    (1)
     (lambda (name) (... val ...))))
```

Such a code piece could for instance appear in an interpreter where a value `val` is computed and bound to a name in an environment represented as function. Assume that `val` is dynamic (its value unavailable at specialization time). Then the specializer should not unfold the let-expression: due to the occurrence of `val` under a lambda, unfolding might *duplicate* (or *discard*) the residual version of the expression `(evalu E ...)`. This is unacceptable: for linear time source programs, duplication may lead to exponential time residual programs [Ses88]. Also, discarding a possibly side effecting residual expression is unacceptable [BD91].

We have therefore, using two-level syntax [NN88, GJ91b], annotated the let-expression as "to be frozen" by underlining the keyword `let` in the source code. In a typical specializer [GJ91b, BD91, Bon91a], *freezing a let-expression implies freezing its body expression.* Consequently, the lambda-expression becomes frozen (hence the underlined `lambda`). Function `f` therefore gets a dynamic argument, that is, a piece of residual code: reductions in the definition of `f` depending on `f`'s argument thus cannot be performed (concretely: applications of `f`'s argument cannot be beta reduced by the specializer).

Now let us consider an equivalent piece where `f` is applied directly to the body of the let-expression:

```
(let ((val (evalu E ...)))                       (2)
  (f (lambda (name) (... val ...))))
```

The result of specializing the *whole* body expression `(f (lambda ...))` must be a piece of residual code (a frozen expression), but the frozen let-expression no longer forces

the lambda-expression *itself* to be frozen. Hence f now gets a non-frozen argument rather than a piece of code, and therefore reductions that depend on f's argument can be performed in f's body. (Note: the lambda-expression may still become frozen due to f's internal operations. When that happens, the expressions (1) and (2) give equally bad results.)

The problem with version (1) is that the let-expression blocks the *consumer* f from being propagated to the *producer*, the lambda-expression. Rewriting programs to propagate consumers to producers in general requires more complex *non-local* rewritings. This can be illustrated by the following only slightly more complex example:

```
... (f (g E ...)) ...                                    (3)

(define (g E ...)
  (let ((val (evalu E ...)))
    (lambda (name) (... val ...))))
```

Propagating the consumer f to the lambda-expression now cannot be done locally: it requires changing the definition of g, for instance by passing f as an argument:

```
... (g E ... f) ...

(define (g E ... f)
  (let ((val (evalu E ...)))
    (f (lambda (name) (... val ...)))))
```

Other places calling g then of course also have to be taken into account.

The problem of propagating consumers through frozen let-expressions often appears in practice. A typical example is interpreters that iteratively compute a number of values and store these in an environment, for instance when interpreting function calls in interpreters for functional languages.

Improving binding times by *manual* cps-conversion of parts of programs has been studied by several authors [Dan91, Jor90, HG91, KS91, Bon91b, Jor92]. It was proposed in [CD91] to *automatically* transform source programs into cps, to maximally propagate consumers through frozen expressions. We shall refer to (both manual and automatic) source program cps-conversion as *explicit* cps-conversion.

The key idea of this paper is, instead of explicitly cps-converting *source programs*, to write the *specializer* in cps. What we shall do is the following:

1. Define a traditional direct style specializer $\mathcal{D}$.
2. Define a cps specializer $\mathcal{C}$ and show it *equivalent* to $\mathcal{D}$.
3. Obtain $\mathcal{C}^2$, an improved version of $\mathcal{C}$.

Obtaining $\mathcal{C}^2$ will be done by modifying $\mathcal{C}$ in order to propagate consumers through frozen let-expressions to producers. The modification is based on the following operational Scheme equivalence $\sim$: for all variables V, all expressions E_1, E_2, and all contexts which are "safe" (defined in section 4) E[_]:

$$E[\texttt{(let ((V } E_1\texttt{)) } E_2\texttt{)}] \sim \texttt{(let ((V } E_1\texttt{)) } E[E_2]\texttt{)}$$

Being an equivalence based on distributing contexts, it is not surprising that continuations come into the picture: in direct style, there is no explicit context that can be manipulated; in cps, however, the continuation (context) can be manipulated explicitly.

1.1 Programming language

A program is a set of recursive procedures (functions) written in a subset of Scheme [IEE90], see figure 1 (essentially the Similix core language [Bon91b]). An expression is a constant (atomic value or quoted expression), a variable, a conditional, a let-expression, a primitive operation, a call to a named procedure, a lambda-abstraction, or an application (of an expression evaluating to the value of an anonymous lambda-abstraction). For convenience, the three application forms (O E*), (P E*), and (A E*) are kept syntactically distinct (the distinction is made automatically during parsing). We do not consider side-effects in this paper. However, the ideas presented here are not restricted to side-effect free languages. The Similix system indeed does treat a limited class of side-effects [Bon91b].

```
Π ∈ Program; D ∈ Definition; E, B, A ∈ Expression;
C ∈ Constant; V ∈ Variable; O ∈ PrimopName;
P ∈ ProcName;

Π       ::= D*
D       ::= (define (P V*) B)
E, B, A ::= C | V | (if E1 E2 E3) |
            (let ((V E1)) E2) | (O E*) | (P E*) |
            (lambda (V*) B) | (A E*)
```

Figure 1: Scheme Subset

We shall write annotated source programs to be specialized in a two-level notation, see figure 2. Each compound expression construct thus exists in two forms, as "to be performed" and as "to be frozen" (underlined). We do not in this paper discuss *how* to annotate a program: this is done automatically by a *binding time analysis* (see e.g. [JSS89, NN88, BD91]). When a program is *safely annotated*, no *tag projection error* can ever occur [GJ91b]; injection tags can therefore safely be omitted, one of the main motivations for doing separate binding time analysis.

```
Π ∈ Program; D ∈ Definition; E, B, A ∈ Expression;
C ∈ Constant; V, W ∈ Variable; O ∈ PrimopName;
P ∈ ProcName; M ∈ LambdaId

Π       ::= D*
D       ::= (define (P V*) B)
E, B, A ::= C | V | (if E1 E2 E3) |
            (let ((V E1)) E2) | (O E*) | (P E*) |
            (lambda-M (V*) B[W*]) | (A E*) |
            (lift E) | (if E1 E2 E3) |
            (let ((V E1)) E2) | (O_E*) | (P_E*) |
            (lambda-M (V*) B[W*]) | (A_E*)
```

Figure 2: Two-Level Scheme Subset

The lift-form allows static first order values to be coerced into residual expressions [GJ91b]. This is done by "adding a quote". The syntax of lambda-abstractions is

augmented: each lambda-expression in a program has a unique identification M, and W* is a list of the free variables in the body B.

2 Direct Style Specialization

We now define a direct style memoizing specializer $\mathcal{D}$, see figure 3. Many of the details are not important for understanding the points of this paper, but we have included the full specializer for completeness. The specializer is the one from (the older direct style version of) Similix, but the points we make when improving binding times in section 5 are not specific to Similix.

A residual program is a set of recursive *specialized procedures*, as described in e.g. [JSS89]. During specialization, procedure calls (P E*) are either unfolded or, if the annotation is underlined, a residual call is generated. When a residual call (P' E'*) is generated, a residual procedure definition (`define` (P' ...) ...) is added to the residual program by $memo_{\mathcal{D}}$. Procedure P' is a specialized version of procedure P. To generate the specialized procedure P', $memo_{\mathcal{D}}$ performs a recursive $\mathcal{D}$-call $\mathcal{D}\,\mathsf{B}\,\rho$ where B is the body of procedure definition P.

However, it can happen that P is called with values that are equivalent to values seen in a previous call to P: then the previously generated residual procedure P' is reused [JSS89, Bon91a] (*memoization*/function caching). The specialization process is initiated by processing an initial "goal" call (P_E*). The details of memoization are of no importance here except for one point: to enable the specializer to recognize functional values equivalent to those seen in a previous call, functional values must be represented as first order data objects (*closures*). Details on when values are considered equivalent are described in [Bon91a] for Similix. Consel's Schism system [Con90b] uses similar representations. Lambda-mix performs no memoization and so can represent functional values by functions [GJ91b].

2.1 Notation

$\mathcal{D}$ is written in a functional style in a completely call-by-value (strict) meta-language. $\mathcal{D}$ is written *operationally*: there is no value representing non-termination, and hence all functions are *partial*: an expression $\mathcal{D}\,\mathsf{E}\,\rho$ may not evaluate to any value. The text of the source program is a global constant Π which it is accessible everywhere. The residual program is stored in a globalized "invisible" variable Π' which is accessed and updated only by $memo_{\mathcal{D}}$. We could have written the specializer in a completely functional way by passing Π' around, letting $\mathcal{D}$ take Π' as an additional parameter and letting $\mathcal{D}$ return Π' as an additional result (packaged in a tuple, for instance), but for expository (and implementational) reasons, we prefer to keep Π' globalized. One more non-functional feature is used: the function *gen-var* which simply generates a fresh residual variable.

The value *error* is used to represent all kinds of errors (type errors, tag projection errors, division by zero, etc.). We use D_{error} to denote the domain $D \cup \{error\}$. The ordering of all domains (also domains of form D_{error}) is the flat one; domains are thus simply *sets*. All functions with an argument domain D_{error} are *error-preserving*: when given the value *error*, they return *error*. All non-*error*-values are called *proper* values.

Index ranges are implicitly defined by the context, see e.g. the rule for (O E*) where E_i is an expression in the list E*. We use $[\_\!\!\rightarrowtail\!\!\_]\_$ to denote function updating, and $[\_\!\!\rightarrowtail\!\!\_]$ to denote updating the "initial" function that maps everything to *error*. Subscripted function updating abbreviates nested function updating, see e.g. the rule for (P E*). $\langle\_\rangle\_$ creates a list (cf. *list comprehensions*); see e.g. the rule for (O E*): the list generated there has the same length as the length of E*. $\_\uparrow\_$ and $\_\downarrow\_$ are used for injecting into and projecting out from tagged sum domains. Projecting with a wrong tag gives *error*; *error*$\downarrow\_$ gives *error*. Precedence: application binds stronger than $\uparrow$ and $\downarrow$. Residual syntax domains are denoted by '-symbols; these domains are identical to source syntax domains. The function arrow $\rightarrow$ denotes the domain of (partial error-preserving call-by-value) functions.

OBSERVATION 1 Inspecting the let-rule in the definition of $\mathcal{D}$, we observe that specializing the body expression E_2 *must* result in a piece of residual code. This was exactly the problem pointed out in the introduction: "freezing a let-expression implies freezing its body expression". □

3 Continuation Passing Style Specialization

Figure 4 defines $\mathcal{C}$, a cps-version of $\mathcal{D}$. Although we wrote $\mathcal{C}$ by hand, cps-versions of direct style programs can be derived automatically [Ste78].

The identity continuation $(\lambda v\,.\,v)$ is denoted by ι. Function $memo_{\mathcal{C}}$ is identical to $memo_{\mathcal{D}}$, except that $memo_{\mathcal{D}}$ performs a call $\mathcal{D}\,\mathsf{B}\,\rho$; $memo_{\mathcal{C}}$ instead performs a call $\mathcal{C}\,\mathsf{B}\,\rho\,\iota$. Notice that $\mathcal{C}$'s continuation parameter κ thus is initialized to ι when initiating generation of a new residual procedure.

3.1 Equivalence with direct style

The cps specializer can be proved equivalent to the direct style specializer: whenever $\mathcal{D}$ gives a proper result, $\mathcal{C}$ gives the same result and vice versa.

DEFINITION 2 For all expressions Exp_1 and Exp_2 we use $\mathrm{Exp}_1 \sqsubseteq \mathrm{Exp}_2$ to denote that if Exp_1 evaluates to a proper value V then Exp_2 also evaluates to V. We write $\mathrm{Exp}_1 \equiv \mathrm{Exp}_2$ iff $\mathrm{Exp}_1 \sqsubseteq \mathrm{Exp}_2 \wedge \mathrm{Exp}_1 \sqsupseteq \mathrm{Exp}_2$. We say that Exp is proper iff Exp evaluates to a proper value. □

Correctness of $\mathcal{C}$ with respect to $\mathcal{D}$ can then be stated as $\forall \mathsf{E}, \rho : \mathcal{D}\,\mathsf{E}\,\rho \equiv \mathcal{C}\,\mathsf{E}\,\rho\,\iota$; this follows from the following more general theorem by inserting ι for κ:

THEOREM 3 $\forall \mathsf{E}, \rho, \kappa : \kappa(\mathcal{D}\,\mathsf{E}\,\rho) \equiv \mathcal{C}\,\mathsf{E}\,\rho\,\kappa$.

PROOF Follows from lemma 4 and lemma 5. □

LEMMA 4 $\forall \mathsf{E}, \rho, \kappa : \kappa(\mathcal{D}\,\mathsf{E}\,\rho) \sqsubseteq \mathcal{C}\,\mathsf{E}\,\rho\,\kappa$.

PROOF The proof is by induction on the structure of evaluation trees. We have to prove that if $\kappa(\mathcal{D}\,\mathsf{E}\,\rho)$ evaluates to a proper value v (the assumption), then $\mathcal{C}\,\mathsf{E}\,\rho\,\kappa$ also evaluates to v. Since κ is strict and error-preserving, the assumption implies that $\mathcal{D}\,\mathsf{E}\,\rho$ is proper.

Transform each right-hand side of the $\mathcal{C}$-rules to a form identical to the corresponding $\mathcal{D}$-rule, with the only exceptions that it contains recursive calls of form $\mathcal{C}\,\mathsf{E}_i\,\rho\,\iota$ rather than $\mathcal{D}\,\mathsf{E}_i\,\rho$ and that κ is applied to the result. The transformation is done by using corollary 7 (going from right to left) to move continuations out from argument to apply positions and then beta-reducing the generated applications $(\lambda v_i\,.\,\ldots)\,(\mathcal{C}\,\mathsf{E}_i\,\rho\,\iota)$.

$u, v, w \in 2Val = Ba + Cl + Co$ — two-level values
$\rho \in 2Env = \mathsf{Variable} \to 2Val$ — two-level environments
$ba \in Ba = \ldots$ — base values (integers, booleans, etc.)
$cl \in Cl = 2Val^* \times \mathsf{Lambdald}$ — closures
$co \in Co = \mathsf{Expression}'$ — residual code expressions

$\mathcal{D} : \mathsf{Expression} \to 2Env \to 2Val_{error}$

$\mathcal{D}\,\mathsf{C}\,\rho = \mathsf{C}\uparrow_{Ba}$

$\mathcal{D}\,\mathsf{V}\,\rho = \rho(\mathsf{V})$

$\mathcal{D}\,(\texttt{if}\ \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3)\,\rho = \mathit{if}\ \mathcal{D}\,\mathsf{E}_1\,\rho\downarrow_{Ba}\ \mathit{then}\ \mathcal{D}\,\mathsf{E}_2\,\rho\ \mathit{else}\ \mathcal{D}\,\mathsf{E}_3\,\rho$

$\mathcal{D}\,(\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho = \mathcal{D}\,\mathsf{E}_2\,[\mathsf{V}\mapsto\mathcal{D}\,\mathsf{E}_1\,\rho]\rho$

$\mathcal{D}\,(\mathsf{O}\ \mathsf{E}^*)\,\rho = \mathcal{O}\,\mathsf{O}\,\langle\mathcal{D}\,\mathsf{E}_i\,\rho\downarrow_{Ba}\rangle_i\uparrow_{Ba}$

$\mathcal{D}\,(\mathsf{P}\ \mathsf{E}^*)\,\rho = \mathcal{D}\,\mathsf{B}\,[\mathsf{V}_i\mapsto\mathcal{D}\,\mathsf{E}_i\,\rho]_i$ *where* $\Pi = \ldots(\texttt{define}\ (\mathsf{P}\ \mathsf{V}^*)\ \mathsf{B})\ldots$

$\mathcal{D}\,(\texttt{lambda-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\,\rho = (\langle\rho(\mathsf{W}_j)\rangle_j, \mathsf{M})\uparrow_{Cl}$

$\mathcal{D}\,(\mathsf{A}\ \mathsf{E}^*)\,\rho = \mathcal{D}\,\mathsf{B}\,[\mathsf{V}_i\mapsto\mathcal{D}\,\mathsf{E}_i\,\rho]_i[\mathsf{W}_j\mapsto w_j]_j$
where $(w^*, \mathsf{M}) = \mathcal{D}\,\mathsf{A}\,\rho\downarrow_{Cl}$, $\Pi = \ldots(\texttt{lambda-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\ldots$

$\mathcal{D}\,(\underline{\texttt{lift}}\ \mathsf{E})\,\rho = \mathit{bld\text{-}cst}(\mathcal{D}\,\mathsf{E}\,\rho\downarrow_{Ba})\uparrow_{Co}$

$\mathcal{D}\,(\underline{\texttt{if}}\ \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3)\,\rho = \mathit{bld\text{-}if}(\mathcal{D}\,\mathsf{E}_1\,\rho\downarrow_{Co}, \mathcal{D}\,\mathsf{E}_2\,\rho\downarrow_{Co}, \mathcal{D}\,\mathsf{E}_3\,\rho\downarrow_{Co})\uparrow_{Co}$

$\mathcal{D}\,(\underline{\texttt{let}}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho = \mathit{bld\text{-}let}(\mathsf{V}', \mathcal{D}\,\mathsf{E}_1\,\rho\downarrow_{Co}, \mathcal{D}\,\mathsf{E}_2\,\rho_1\downarrow_{Co})\uparrow_{Co}$ *where* $\mathsf{V}' = \mathit{gen\text{-}var}()$, $\rho_1 = [\mathsf{V}\mapsto\mathsf{V}']\rho$

$\mathcal{D}\,(\mathsf{O}\_\mathsf{E}^*)\,\rho = \mathit{bld\text{-}primop}(\mathsf{O}, \langle\mathcal{D}\,\mathsf{E}_i\,\rho\downarrow_{Co}\rangle_i)\uparrow_{Co}$

$\mathcal{D}\,(\mathsf{P}\_\mathsf{E}^*)\,\rho = \mathit{bld\text{-}pcall}(\mathsf{P}', u^*)\uparrow_{Co}$ *where* $(\mathsf{P}', u^*) = \mathit{memo}_{\mathcal{D}}\ \mathsf{P}\ \langle\mathcal{D}\,\mathsf{E}_i\,\rho\rangle_i$

$\mathcal{D}\,(\underline{\texttt{lambda}}\text{-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\,\rho = \mathit{bld\text{-}lam}(\langle\mathsf{V}'_i\rangle_i, \mathcal{D}\,\mathsf{B}\,[\mathsf{V}_i\mapsto\mathsf{V}'_i]_i\rho\downarrow_{Co})\uparrow_{Co}$ *where* $\mathsf{V}'_i = \mathit{gen\text{-}var}()$

$\mathcal{D}\,(\mathsf{A}\_\mathsf{E}^*)\,\rho = \mathit{bld\text{-}app}(\mathcal{D}\,\mathsf{A}\,\rho, \langle\mathcal{D}\,\mathsf{E}_i\,\rho\downarrow_{Co}\rangle_i)\uparrow_{Co}$

$\mathcal{O} : \mathsf{PrimopName} \to Ba^*_{error} \to Ba_{error}$
$\mathit{memo}_{\mathcal{D}} : \mathsf{ProcName} \to 2Val^*_{error} \to (\mathsf{ProcName}' \times Co^*)_{error}$
$\mathit{bld\text{-}cst} : Ba \to Co$
$\mathit{bld\text{-}if} : (Co \times Co \times Co)_{error} \to Co_{error}$
$\vdots$

Figure 3: Direct Style Specializer

Then apply lemma 4 inductively to each occurrence of expressions of form $\mathcal{D}\,\mathsf{E}_i\,\rho$ in the right-hand sides of the definition of $\mathcal{D}$ (and to the expression $\mathcal{D}\,\mathsf{B}\,\rho$ inside $\mathit{memo}_{\mathcal{D}}$). Validity of induction: by assumption, $\mathcal{D}\,\mathsf{E}\,\rho$ is proper; hence, for any evaluation tree, any expression of the form $\mathcal{D}\,\mathsf{E}_i\,\rho$ in the evaluation subtree is proper. Thus lemma 4 can be applied. □

LEMMA 5 $\forall \mathsf{E}, \rho, \kappa : \kappa(\mathcal{D}\,\mathsf{E}\,\rho) \sqsupseteq \mathcal{C}\,\mathsf{E}\,\rho\,\kappa$.

PROOF By induction on the structure of evaluation trees, assuming that $\mathcal{C}\,\mathsf{E}\,\rho\,\kappa$ is proper. The proof is similar to the proof of lemma 4. □

PROPOSITION 6 $\forall \mathsf{E}, \rho, \kappa, \mu : \mu(\mathcal{C}\,\mathsf{E}\,\rho\,\kappa) \equiv \mathcal{C}\,\mathsf{E}\,\rho\,\mu\circ\kappa$.

PROOF Follows from lemma 8 and lemma 9. □

COROLLARY 7 $\forall \mathsf{E}, \rho, \kappa : \kappa(\mathcal{C}\,\mathsf{E}\,\rho\,\iota) \equiv \mathcal{C}\,\mathsf{E}\,\rho\,\kappa$.

PROOF Follows from proposition 6 by inserting ι for κ. □

LEMMA 8 $\forall \mathsf{E}, \rho, \kappa, \mu : \mu(\mathcal{C}\,\mathsf{E}\,\rho\,\kappa) \sqsubseteq \mathcal{C}\,\mathsf{E}\,\rho\,\mu\circ\kappa$.

PROOF By induction on the structure of evaluation trees, assuming that $\mu(\mathcal{C}\,\mathsf{E}\,\rho\,\kappa)$ is proper. From the assumption, it follows that $\mathcal{C}\,\mathsf{E}\,\rho\,\kappa$ is proper since μ is strict and error-preserving. For each right-hand side in the definition of $\mathcal{C}$, propagate μ inwards so that it is applied to the results of the recursive $\mathcal{C}$-calls. Then use lemma 8 inductively. □

LEMMA 9 $\forall \mathsf{E}, \rho, \kappa, \mu : \mu(\mathcal{C}\,\mathsf{E}\,\rho\,\kappa) \sqsupseteq \mathcal{C}\,\mathsf{E}\,\rho\,\mu\circ\kappa$.

PROOF By induction on the structure of evaluation trees, assuming that $\mathcal{C}\,\mathsf{E}\,\rho\,\mu\circ\kappa$ is proper. Similar to the proof of lemma 8. □

4 Well-behaved continuations

Some of the continuations in figure 4 turn out to be of a form not well-suited for the transformations we shall perform to improve binding times. In this section we rewrite the definition of $\mathcal{C}$ to bring the continuations on the needed form.

2Val, *2Env*, *Ba*, *Cl*, and *Co* as in figure 3

$\kappa, \mu \in 2Cont = 2Val_{error} \to 2Val_{error}$ — continuations

$\mathcal{C}$: Expression $\to$ *2Env* $\to$ *2Cont* $\to$ $2Val_{error}$

$$\mathcal{C}\,\mathsf{C}\,\rho\,\kappa = \kappa(\mathsf{C}\uparrow_{Ba})$$

$$\mathcal{C}\,\mathsf{V}\,\rho\,\kappa = \kappa(\rho(\mathsf{V}))$$

$$\mathcal{C}\,(\texttt{if}\ \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 .\ \textit{if}\ v_1\downarrow_{Ba}\ \textit{then}\ \mathcal{C}\,\mathsf{E}_2\,\rho\,\kappa\ \textit{else}\ \mathcal{C}\,\mathsf{E}_3\,\rho\,\kappa)$$

$$\mathcal{C}\,(\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 .\ \mathcal{C}\,\mathsf{E}_2\,[\mathsf{V}\mapsto v_1]\rho\,\kappa)$$

$$\mathcal{C}\,(\mathsf{O}\ \mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \kappa(\mathcal{O}\,\mathsf{O}\,\langle v_i\downarrow_{Ba}\rangle_i\uparrow_{Ba}))\ldots)$$

$$\mathcal{C}\,(\mathsf{P}\ \mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \mathcal{C}\,\mathsf{B}\,[\mathsf{V}_i\mapsto v_i]_i\,\kappa)\ldots)\ \textit{where}\ \Pi = \ldots(\texttt{define}\ (\mathsf{P}\ \mathsf{V}^*)\ \mathsf{B})\ldots$$

$$\mathcal{C}\,(\texttt{lambda-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\,\rho\,\kappa = \kappa((\langle\rho(\mathsf{W}_j)\rangle_j, \mathsf{M})\uparrow_{Cl})$$

$$\mathcal{C}\,(\mathsf{A}\ \mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{A}\,\rho\,(\lambda u .\ \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \mathcal{C}\,\mathsf{B}\,\rho_1\,\kappa)\ldots))$$
$$\textit{where}\ (w^*, \mathsf{M}) = u\downarrow_{Cl},\ \rho_1 = [\mathsf{V}_i\mapsto v_i]_i[\mathsf{W}_j\mapsto w_j]_j,\ \Pi = \ldots(\texttt{lambda-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\ldots$$

$$\mathcal{C}\,(\underline{\texttt{lift}}\ \mathsf{E})\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}\,\rho\,(\lambda v .\ \kappa(\textit{bld-cst}(v\downarrow_{Ba})\uparrow_{Co}))$$

$$\mathcal{C}\,(\underline{\texttt{if}}\ \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 .\ \mathcal{C}\,\mathsf{E}_2\,\rho\,(\lambda v_2 .\ \mathcal{C}\,\mathsf{E}_3\,\rho\,(\lambda v_3 .\ \kappa(\textit{bld-if}(v_1\downarrow_{Co},\ v_2\downarrow_{Co},\ v_3\downarrow_{Co})\uparrow_{Co}))))$$

$$\mathcal{C}\,(\underline{\texttt{let}}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 .\ \mathcal{C}\,\mathsf{E}_2\,\rho_1\,(\lambda v_2 .\ \kappa(\textit{bld-let}(\mathsf{V}',\ v_1\downarrow_{Co},\ v_2\downarrow_{Co})\uparrow_{Co})))$$
$$\textit{where}\ \mathsf{V}' = \textit{gen-var}(),\ \rho_1 = [\mathsf{V}\mapsto\mathsf{V}']\rho$$

$$\mathcal{C}\,(\mathsf{O}\_\mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \kappa(\textit{bld-primop}(\mathsf{O},\ \langle v_i\downarrow_{Co}\rangle_i)\uparrow_{Co}))\ldots)$$

$$\mathcal{C}\,(\mathsf{P}\_\mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \kappa(\textit{bld-pcall}(\mathsf{P}',\ u^*)\uparrow_{Co}))\ldots)$$
$$\textit{where}\ (\mathsf{P}', u^*) = \textit{memo}_{\mathcal{C}}\ \mathsf{P}\ \langle v_i\rangle_i$$

$$\mathcal{C}\,(\underline{\texttt{lambda}}\text{-M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\,\rho\,\kappa = \mathcal{C}\,\mathsf{B}\,[\mathsf{V}_i\mapsto\mathsf{V}'_i]_i\rho\,(\lambda v .\ \kappa(\textit{bld-lam}(\langle\mathsf{V}'_i\rangle_i,\ v\downarrow_{Co})\uparrow_{Co}))\ \textit{where}\ \mathsf{V}'_i = \textit{gen-var}()$$

$$\mathcal{C}\,(\mathsf{A}\_\mathsf{E}^*)\,\rho\,\kappa = \mathcal{C}\,\mathsf{A}\,\rho\,(\lambda u .\ \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1 . \ldots \mathcal{C}\,\mathsf{E}_n\,\rho\,(\lambda v_n .\ \kappa(\textit{bld-app}(u,\ \langle v_i\downarrow_{Co}\rangle_i)\uparrow_{Co}))\ldots))$$

Figure 4: Continuation Passing Style Specializer

DEFINITION 10 $\mathsf{E}[\_]$ denotes a *context*: an expression with one missing subexpression. $\mathsf{E}[\mathsf{E}_1]$ denotes the expression obtained by *filling* the hole in the context expression $\mathsf{E}[\_]$ with expression E_1. $\mathsf{E}[\_]$ may be the *empty context* $[\_]$ in which case $\mathsf{E}[\mathsf{E}_1] = \mathsf{E}_1$. □

DEFINITION 11 We define the *safe* contexts recursively: if $\mathsf{E}[\_]$ is safe, then the following contexts are also safe:

$[\_]$; (if $\mathsf{E}[\_]$ E_2 E_3) ; (let ((V $\mathsf{E}[\_]$)) E_2) ;

(O E_1,...,$\mathsf{E}[\_]$,...,E_n) ; (P E_1,...,$\mathsf{E}[\_]$,...,E_n) ;

(A$[_]$ E_1,...,E_n) ; (A E_1,...,$\mathsf{E}[\_]$,...,E_n)

We define: *safe*($\mathsf{E}[\_]$) $\iff$ $\mathsf{E}[\_]$ is safe. □

It follows from this definition that the *unsafe* contexts are those that contain a (sub-)context of one of the forms

(if E_1 $\mathsf{E}[\_]$ E_3) ; (if E_1 E_2 $\mathsf{E}[\_]$) ;

(let ((V E_1)) $\mathsf{E}[\_]$) ; (lambda (V*) B$[_]$)

Thus, if the hole in a context $\mathsf{E}[\_]$ lies in some sub-context that is the then- or else-branch of a conditional, the body of a let-expression, or the body of a lambda-expression, then $\mathsf{E}[\_]$ is unsafe.

We use $\sim$ to denote equivalence between expressions:

DEFINITION 12 $\forall \mathsf{E}_1, \mathsf{E}_2 : \mathsf{E}_1 \sim \mathsf{E}_2$ iff E_1 and E_2 are operationally equivalent. □

We state the following property without proof:

PROPERTY 13

$\forall \mathsf{V}, \mathsf{E}_1, \mathsf{E}_2, \mathsf{E}[\_] : \textit{safe}(\mathsf{E}[\_]) \Rightarrow$

$\mathsf{E}[(\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)] \sim (\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}[\mathsf{E}_2])$ □

It is easy to see that the equivalence does *not* hold for unsafe contexts: (1) A conditional is not strict in the then- and else-branches, so E_1 need not be evaluated in $\mathsf{E}[(\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)]$, but is always evaluated in (let ((V E_1)) $\mathsf{E}[\mathsf{E}_2]$). (2) A let-expression introduces a binding that may be used in E_1 in $\mathsf{E}[(\texttt{let}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)]$; this binding is not visible to E_1 in (let ((V E_1)) $\mathsf{E}[\mathsf{E}_2]$). (3) A lambda-expression both introduces bindings and is not not strict in its body-expression.

That the let-equivalence only holds for safe contexts motivates the following definition to distinguish continuations that behave like safe contexts from those that do not:

DEFINITION 14 A continuation κ is *well-behaved* iff $wb(\kappa)$ holds:

$\forall \kappa : wb(\kappa) \iff$

$(\forall co : \kappa(co\uparrow_{Co})$ is not proper) $\vee$

$(\exists \mathsf{E}'[\_] : \textit{safe}(\mathsf{E}'[\_]) \wedge \forall co : \kappa(co\uparrow_{Co}) = \mathsf{E}'[co]\uparrow_{Co})$. □

Thus, when given an argument $co\uparrow_{Co}$, a well-behaved con-

$$
\begin{array}{ll}
\mathcal{C}\,(\underline{\texttt{if}}\ \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3)\,\rho\,\kappa & = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\kappa(\mathit{bld\text{-}if}(v_1{\downarrow_{Co}}\,,\,\mathcal{C}\,\mathsf{E}_2\,\rho\,\iota{\downarrow_{Co}}\,,\,\mathcal{C}\,\mathsf{E}_3\,\rho\,\iota{\downarrow_{Co}}){\uparrow_{Co}})) \\
\mathcal{C}\,(\underline{\texttt{let}}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho\,\kappa & = \mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\kappa(\mathit{bld\text{-}let}(\mathsf{V}'\,,\,v_1{\downarrow_{Co}}\,,\,\mathcal{C}\,\mathsf{E}_2\,\rho_1\,\iota{\downarrow_{Co}}){\uparrow_{Co}})) \\
 & \quad \textit{where}\ \mathsf{V}' = \mathit{gen\text{-}var}()\,,\ \rho_1 = [\mathsf{V}{\mapsto}\mathsf{V}']\rho \\
\mathcal{C}\,(\underline{\texttt{lambda}}\text{-}\mathsf{M}\ (\mathsf{V}^*)\ \mathsf{B}[\mathsf{W}^*])\,\rho\,\kappa & = \kappa(\mathit{bld\text{-}lam}((\mathsf{V}'_i)_i\,,\,\mathcal{C}\,\mathsf{B}\,[\mathsf{V}_i{\mapsto}\mathsf{V}'_i]_i\,\rho\,\iota{\downarrow_{Co}}){\uparrow_{Co}})\ \textit{where}\ \mathsf{V}'_i = \mathit{gen\text{-}var}()
\end{array}
$$

Figure 5: Making the Continuations Well-Behaved

$$
\begin{array}{l}
\mathcal{C}^2\,(\underline{\texttt{let}}\ ((\mathsf{V}\ \mathsf{E}_1))\ \mathsf{E}_2)\,\rho\,\kappa = \mathcal{C}^2\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\mathit{bld\text{-}let}(\mathsf{V}'\,,\,v_1{\downarrow_{Co}}\,,\,\mathcal{C}^2\,\mathsf{E}_2\,\rho_1\,\kappa{\downarrow_{Co}}){\uparrow_{Co}}) \\
\qquad\qquad\qquad\qquad \textit{where}\ \mathsf{V}' = \mathit{gen\text{-}var}()\,,\ \rho_1 = [\mathsf{V}{\mapsto}\mathsf{V}']\rho
\end{array}
$$

Figure 6: Improved let*-rule*

tinuation κ either *always* fails or *always* generates an expression E' which contains *co* as a subexpression; notice that context $\mathsf{E}'[\_]$ is safe and does not depend on *co*. Also notice that continuations expecting an argument $ba{\uparrow_{Ba}}$ or $cl{\uparrow_{Cl}}$ are trivially well-behaved.

We observe from figure 4 that some of the continuations are not well-behaved: the $(\lambda v_2\,.\,\ldots)$- and $(\lambda v_3\,.\,\ldots)$-continuations in the if-rule, the $(\lambda v_2\,.\,\ldots)$-continuation in the let-rule, and the $(\lambda v\,.\,\ldots)$-continuation in the lambda-rule. These continuations all "dump" their argument in an unsafe position, the then- or else-branch of a generated conditional, the body of a generated let-expression, or the body of a generated lambda-expression.

We can make all continuations well-behaved (without changing the meaning of $\mathcal{C}$) by transforming the if-, let-, and lambda-rules: use corollary 7 (going from right to left) to move the non-well-behaved continuations μ out in apply-position, then beta-reduce the application $\mu(\ldots)$. This gives the rules in figure 5. When we refer to $\mathcal{C}$ in the rest of this paper, we mean the well-behaved version with the if-, let-, and lambda-rules from figure 5.

5 Improving binding times

As stated by theorem 3, we so far have not gained anything by going from direct style to cps: $\mathcal{D}$ and $\mathcal{C}$ produce the same results. What we shall do now is to modify $\mathcal{C}$'s let-rule, yielding an *improved* specializer $\mathcal{C}^2$. The improvement is done by propagating the consumer, the continuation κ, through the *bld-let* to the producer, the let-body. This is exactly the distributive consumer-propagation discussed in section 1.

The right-hand side of the let-rule from figure 5 is improved in two steps:

$$
\begin{array}{l}
\mathcal{C}\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\kappa(\mathit{bld\text{-}let}(\mathsf{V}'\,,\,v_1{\downarrow_{Co}}\,,\,\mathcal{C}\,\mathsf{E}_2\,\rho_1\,\iota{\downarrow_{Co}}){\uparrow_{Co}})) \stackrel{\text{step1}}{\Longrightarrow} \\
\mathcal{C}^1\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\mathit{bld\text{-}let}(\mathsf{V}'\,,\,v_1{\downarrow_{Co}}\,,\,\kappa(\mathcal{C}^1\,\mathsf{E}_2\,\rho_1\,\iota){\downarrow_{Co}}){\uparrow_{Co}}) \stackrel{\text{step2}}{\Longrightarrow} \\
\mathcal{C}^2\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\mathit{bld\text{-}let}(\mathsf{V}'\,,\,v_1{\downarrow_{Co}}\,,\,\mathcal{C}^2\,\mathsf{E}_2\,\rho_1\,\kappa{\downarrow_{Co}}){\uparrow_{Co}})
\end{array}
$$

The improved let-rule is given in figure 6. All other $\mathcal{C}^2$-rules are identical to those of $\mathcal{C}$, except that all recursive $\mathcal{C}$-calls should be replaced by $\mathcal{C}^2$-calls. In section 5.2 we come back to *why* the two steps improve the binding times.

The improved let-rule looks very natural, resembling the rule for the non-underlined `let` a lot. If one had originally formulated specialization in cps, not having an equivalent direct style style specializer in mind as we did, it is most likely that one would have written the improved let-rule in the first place.

5.1 Correctness of the steps

In this section we show correctness of $\mathcal{C}^2$ with respect to $\mathcal{C}$ (and thus also with respect to $\mathcal{D}$). Correctness is expressed by corollary 18, which is an instance of theorem 17: whenever $\mathcal{C}$ gives a proper result, $\mathcal{C}^2$ gives an equivalent result.

We need to extend definition 12 to handle values from *2Val*, not just from *Co*:

DEFINITION 15
$\forall co_1, co_2 : co_1{\uparrow_{Co}} \sim co_2{\uparrow_{Co}}$ if $co_1 \sim co_2$
$\forall v_1, v_2 : v_1 \sim v_2$ if $v_1 = v_2$ □

The equivalence $co_1 \sim co_2$ is defined in definition 12; recall that Co = Expression. We also need a slightly modified version of definition 2:

DEFINITION 16 For all expressions Exp_1 and Exp_2 we use $\mathrm{Exp}_1 \preceq \mathrm{Exp}_2$ to denote that if Exp_1 evaluates to a proper value V_1 then Exp_2 evaluates to a proper value V_2 such that $\mathrm{V}_1 \sim \mathrm{V}_2$. □

Notice that V_1 and V_2 need not be equal, they need only be equivalent according to the definitions 12 and 15. Also notice that $\preceq$ is transitive.

The following theorem expresses correctness of $\mathcal{C}^2$ with respect to $\mathcal{C}$:

THEOREM 17 $\forall \mathsf{E}, \rho, \kappa : wb(\kappa) \Rightarrow \mathcal{C}\,\mathsf{E}\,\rho\,\kappa \preceq \mathcal{C}^2\,\mathsf{E}\,\rho\,\kappa$.

PROOF Follows from lemma 21, lemma 23, and transitivity of $\preceq$. □

COROLLARY 18 $\forall \mathsf{E}, \rho : \mathcal{C}\,\mathsf{E}\,\rho\,\iota \preceq \mathcal{C}^2\,\mathsf{E}\,\rho\,\iota$.

PROOF Follows from theorem 17 by inserting ι for κ. ι is trivially well-behaved. □

The central lemma is lemma 19:

LEMMA 19
$$
\begin{array}{l}
\forall \kappa : wb(\kappa) \wedge (\forall co : \kappa(co{\uparrow_{Co}}) \text{ is proper}) \Rightarrow \\
\quad (\forall \mathsf{V}', \mathsf{E}'_1, \mathsf{E}'_2 : \kappa(\mathit{bld\text{-}let}(\mathsf{V}'\,,\,\mathsf{E}'_1\,,\,\mathsf{E}'_2){\uparrow_{Co}}) \sim \\
\qquad\qquad \mathit{bld\text{-}let}(\mathsf{V}'\,,\,\mathsf{E}'_1\,,\,\kappa(\mathsf{E}'_2{\uparrow_{Co}}){\downarrow_{Co}}){\uparrow_{Co}}).
\end{array}
$$

PROOF If the condition holds, then, since κ is well-behaved, there exists a safe context $\mathsf{E}'[\_]$ such that $\kappa(co{\uparrow_{Co}}) = \mathsf{E}'[co]{\uparrow_{Co}}$ for any *co*. We therefore have to prove

$$\forall V', E_1', E_2', E'[\_] : safe(E'[\_]) \Rightarrow E'[\texttt{(let ((}V'\ E_1'\texttt{))}\ E_2')] \sim \texttt{(let ((}V'\ E_1'\texttt{))}\ E'[E_2'])$$

which is stated by property 13. □

The following lemma expresses that $\mathcal{C}$'s continuation argument is never discarded, but always applied:

LEMMA 20
$\forall E, \rho, \kappa : wb(\kappa) \wedge \mathcal{C}\,E\,\rho\,\kappa$ is proper $\Rightarrow$
κ is always applied and the result of the application is proper.

PROOF By induction on the structure of evaluation trees. For any evaluation tree, κ is eventually applied to some argument (by induction, the continuations of form $(\lambda v. \ldots \kappa \ldots)$ are applied, so eventually κ is also applied); the result of this application must be proper, otherwise the inductive assumption would be contradicted. □

LEMMA 21 $\forall E, \rho, \kappa : wb(\kappa) \Rightarrow \mathcal{C}\,E\,\rho\,\kappa \preceq \mathcal{C}^1\,E\,\rho\,\kappa$

PROOF By induction on the structure of evaluation trees, assuming that $\mathcal{C}\,E\,\rho\,\kappa$ is proper. For each syntactic case, induction is valid because if κ is well-behaved then the continuations supplied in the recursive $\mathcal{C}$-calls are also well-behaved ($\mathcal{C}$ only introduces well-behaved continuations, cf. section 4).

The only non-trivial case is then the one for let. Here lemma 19 is used to rewrite the right hand side of $\mathcal{C}$'s let-rule, moving κ inside of *bld-let*. The condition of lemma 19 is fulfilled because: (1) κ is well-behaved by the inductive assumption; (2) by the inductive assumption, by lemma 20, because κ is well-behaved, and by the fact that κ is applied to an argument of form $co{\uparrow_{Co}}$, it holds that $\forall co : \kappa(co{\uparrow_{Co}})$ is proper. □

The converse of lemma 21 ($\mathcal{C}\,E\,\rho\,\kappa \succeq \mathcal{C}^1\,E\,\rho\,\kappa$) does *not* hold. If we were to prove it, we would need to use lemma 19 "the other way around" to move κ outside of *bld-let*. But this is not possible in general: in $\mathcal{C}^1$'s let-rule, κ's argument can be of a form different from $\ldots{\uparrow_{Co}}$ while $\mathcal{C}^1$ may still produce a proper result.

For the same reason, the equivalent of proposition 6 does not hold for $\mathcal{C}^1$. However, we do not need proposition 6 in full generality anymore, a weakened "one-way" form will suffice:

LEMMA 22 $\forall E, \rho, \kappa, \mu : wb(\mu) \Rightarrow \mu(\mathcal{C}^1\,E\,\rho\,\kappa) \preceq \mathcal{C}^1\,E\,\rho\,\mu\circ\kappa.$

PROOF By induction on the structure of evaluation trees, assuming that $\mu(\mathcal{C}^1\,E\,\rho\,\kappa)$ is proper. From the assumption, it follows that $\mathcal{C}^1\,E\,\rho\,\kappa$ is proper since μ is strict and error-preserving. All generated continuations are well-behaved, so induction is always valid. The proof is similar to the proof of lemma 8, but everywhere $\sim$ is used instead of $=$, and lemma 19 is needed additionally for the let-case to propagate μ through *bld-let*. The condition of lemma 19 is fulfilled by a reasoning similar to the one used in the proof of lemma 21. □

LEMMA 23 $\forall E, \rho, \kappa : wb(\kappa) \Rightarrow \mathcal{C}^1\,E\,\rho\,\kappa \preceq \mathcal{C}^2\,E\,\rho\,\kappa$

PROOF By induction on the structure of evaluation trees, assuming that $\mathcal{C}^1\,E\,\rho\,\kappa$ is proper. The proof is similar to the proof of lemma 21, except that lemma 22 is used for the let-case instead of lemma 19. □

5.2 What is the improvement?

As indicated by theorem 17, there exist expressions for which $\mathcal{C}^2$ produces a proper result while $\mathcal{C}$ does not. This happens when processing let-expressions: *there is no requirement to freeze the body expression of a* let *in* $\mathcal{C}^2$*!* This is different from $\mathcal{C}$ (and the equivalent $\mathcal{D}$), cf. observation 1.

When using $\mathcal{C}^2$, we *may* choose to freeze the bodies of let's (in which case we get residual programs equivalent to those produced by $\mathcal{C}$, cf. theorem 17), but we do not *have to* do so. By choosing not to freeze, we get exactly the binding time improvements we are looking for: a consumer can be propagated through let's to a producer. Thus, to obtain such a propagation, *it is not necessary to convert source programs.* For example, specializing the example expressions (1) and (2) from section 1 (with expression (1)'s `lambda` non-underlined) now gives equally good results. Also, example (3) from section 1 can now be specialized with the `lambda` non-underlined; reductions that depend on f's argument can now be performed by the specializer.

5.3 Implementation

We have integrated a $\mathcal{C}^2$-based specializer in the Similix system (the specializer of Similix 4.0 is thus directly based on $\mathcal{C}^2$). Compared to the earlier $\mathcal{D}$-based specializer, the $\mathcal{C}^2$-based specializer allows source programs to be expressed in a much more natural and readable way. Major examples are the Similix specializer itself and the BAWL-interpreter (an interpreter for an Orwell-like lazy functional language) from which a compiler was generated by partial evaluation [Jor92].

6 Discussion and Comparison with Explicit CPS-Conversion

6.1 Residual programs

When specializing explicitly cps-converted source programs, residual programs will always be in cps. This is not always desirable as pointed out in [Dan92] in which automatic transformation back to direct style is described. A specializer written in cps such as $\mathcal{C}^2$ does *not* generate residual programs in cps (unless of course the source programs are in cps in the first place).

6.2 Generating extensions

Explicit cps-conversion of source programs has negative effects on the "generating extensions" generated by self-applying the specializer mix, for instance the compilers generated by specializing mix with respect to interpreters: comp = *mix*(mix, int). In a memoizing specializer, functions in int, including continuations, must be represented as closures by mix (cf. section 2). Operations on these closures are dynamic at self-application time and thus cannot be reduced when generating comp; these relatively costly operations therefore remain in the code of comp.

These problems do not occur with our approach: since there is no cps-conversion of source programs, int contains no additional continuations that would give an overhead in the code of comp. Note: compared to a direct style mix, we do get an overhead in the compiler generator cogen = *mix*(mix, mix) since the second mix is in cps.

Compiler comp is a specialized version of mix (which is in cps) and so will itself be written in cps; but since the

continuations in comp are not represented as closures, this is not a problem. In [CD91], the mix being specialized to generate comp need not be cps-converted (mix is a special case there since it only operates on cps-programs, not arbitrary programs); hence their generated compilers will not themselves be written in cps.

6.3 Manual debugging of binding time analysed source code

It is being considered increasingly important in the partial evaluation community to provide useful feedback from the binding time analysis. With explicit cps-conversion, the user sees a binding time analysed *converted* program which he/she then has to relate to the hand-written input code (unless the binding time analysed program is converted back to direct style [Dan92]). With our approach there is no conversion into cps, so binding time analysis is done on the user written code (modulo other source transformations such as macro-expansion). Hence the user sees his/her own hand-written code with added binding time information.

6.4 Specialization points

Explicit cps-conversion allows context information to be propagated through specialization points (calls (P_E*) to be specialized), but our $memo_C$ (see figure 4) does *not* specialize with respect to κ. Instead, κ is *applied* in the (P_E*)-rule. This means that $\mathcal{C}$ (and $\mathcal{C}^2$ as well) does *not* propagate context information through specialization points!

There is nothing that prevents us from rewriting $\mathcal{C}/\mathcal{C}^2$ to propagate context information through specialization points: rather than applying κ in the (P_E*)-rule, we could give it as an argument to $memo_C$. Then $memo_C$ should perform a recursive call $\mathcal{C}\,\mathsf{B}\,\rho\,\kappa$ instead of the current $\mathcal{C}\,\mathsf{B}\,\rho\,\iota$ (cf. the beginning of section 3).

However, there would be some disadvantages with this approach. In addition to identifying when *values* are equivalent to values seen in an earlier call (cf. section 2), $memo_C$ should now also identify when the *continuation* κ were equivalent to a continuation seen in an earlier call. In order to perform such a comparison, κ would need to be represented explicitly as a first order data structure, a closure; κ could no longer simply be a function in the implementation language (in our case Scheme). Representing κ as a closure would have negative effects on the efficiency of mix, and the code of the generating extensions would be severely worsened: there would be a large overhead of code for closure manipulations.

Specializing with respect to continuations also increases termination problems: explicit cps-conversion introduces an additional source of non-terminating specialization [CD91]; continuations sometimes need to be *generalized* to ensure termination. Similarly, if $memo_C$ always memoized with respect to κ, specialization would hardly ever terminate: larger and larger continuations κ would be built during specialization. Therefore, to ensure termination, κ would sometimes need to be reset, for instance by using the (P_E*)-rule of figure 4; resetting κ parallels the effect of generalizing continuations in explicitly cps-converted source programs [CD91].

6.5 Dynamic choice of static values

We can binding time improve the treatment of if by using the following distributive Scheme equivalence: for all expressions E_1, E_2, E_3, and all safe contexts $\mathsf{E}[\_]$:

$$\mathsf{E}[\texttt{(if } \mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3\texttt{)}] \sim \texttt{(if } \mathsf{E}_1\ \mathsf{E}[\mathsf{E}_2]\ \mathsf{E}[\mathsf{E}_3]\texttt{)}$$

By rewriting the right-hand side of the if-rule in figure 5 using steps similar to step 1 and step 2 in section 5, we obtain

$$\mathcal{C}^2\,\mathsf{E}_1\,\rho\,(\lambda v_1\,.\,\mathit{bld\text{-}if}(v_1{\downarrow_{Co}}, \mathcal{C}^2\,\mathsf{E}_2\,\rho\,\kappa{\downarrow_{Co}}, \mathcal{C}^2\,\mathsf{E}_3\,\rho\,\kappa{\downarrow_{Co}}){\uparrow_{Co}})$$

Now $\mathcal{C}^2$ supports *dynamic choice of static values* [Mog89]: even though the test E_1 is frozen, the branches E_2 and E_3 need not be frozen. Notice a problem of code duplication, though: κ is duplicated in the above expression (κ is duplicated nowhere else in $\mathcal{C}^2$, cf. figure 4).

Dynamic choice of static values is needed to handle the pattern matcher example of [CD91]. Another "classical" example of dynamic choice of static values is dynamic indexing in a static environment (called "finitely dynamic values" in [GJ91a]; see also [HH90, HG91, Bon91b]):

```
... (f (lookup n ns vs)) ...

(define (lookup n ns values)
  (if (null? ns)
      error
      (if (equal? n (car ns))
          (car vs)
          (lookup n (cdr ns) (cdr vs)))))
```

Assume that ns and vs are static, but n dynamic. Since the static ns decreases for every recursive call to lookup, it is safe to unfold completely; termination is guaranteed. The result will be a piece of non-recursive residual code, essentially a case-expression that compares n to all values in ns and in each case returns the corresponding value in vs. Function f will be propagated through the tests and is applied to each of the possible (static) values in vs: f's argument becomes static.

We have not included dynamic choice of static values in the Similix 4.0 implementation of $\mathcal{C}^2$. The reason is that Similix 4.0 uses a simple specialization point insertion strategy that inserts specialization points at *all* dynamic conditionals [BD91]. Similix 4.0 does not propagate κ though specialization points (because of the negative effects described in section 6.4), so any dynamic conditional effectively blocks propagation of κ. To obtain dynamic choice of static values when using Similix 4.0, explicit cps-conversion of source program as described in e.g. [HG91] is therefore needed. A better, less conservative, specialization point insertion strategy could be imagined: it would for instance neither insert the unneeded specialization points in the pattern matcher example of [CD91] nor in the dynamic indexing example. Given such a specialization point insertion strategy, Similix would be able to support dynamic choice of static values.

6.6 Other expression forms

Could we hope to propagate contexts through other underlined expression forms than let and if? The answer is no: let and if are the only expression forms in our Scheme subset which have distributive equivalences that allow us

to propagate the context into the subexpressions, and thus these expression forms are the only ones that allow us to propagate κ through the corresponding *bld-...* form.

7 Partially Static Data Structures

Treating *partially static data structures* [Mog88] safely is difficult due to the risk of duplicating/discarding residual code expressions that become part of a partially static data structure. Discarding is generally unsafe for languages with side-effects [BD91]; it is also unsafe for side-effect free languages due to the risk of not preserving termination properties [BD91].

For instance, consider the expression E_0 =

```
(let ((V (cons E1 E2))) E3)
```

where E_1 is static and E_2 dynamic. Partially static data structures enable operations on the data structure created by the cons, so the result of specializing E_0 will for instance be

$$E_1' \textit{ if } E_3 = \texttt{(car V)}$$

and

$$\texttt{(+ } E_2' \ E_2'\texttt{)} \textit{ if } E_3 = \texttt{(+ (cdr V) (cdr V))}$$

(we use E_x' to denote the result of specializing E_x).

However, notice that these reductions unfortunately respectively discard and duplicate E_2'! We can get around this problem by rewriting constructor expressions, wrapping them into let-expressions that bind the arguments:

```
(cons E1 E2) ⟹
(let ((V1 E1)) (let ((V2 E2)) (cons V1 V2)))
```

Note: cons evaluates its arguments in an arbitrary order; in the transformation, we have arbitrarily chosen left-to-right evaluation. Rewriting the example expression E_0 would thus give E_{00} =

```
(let ((V (let ((V1 E1))
           (let ((V2 E2))
             (cons V1 V2)))))
  E3)
```

The binding of the dynamic E_2 has been underlined to prevent unfolding.

Specializing E_{00} with a traditional direct style specializer such as $\mathcal{D}$ (extended to handle operations on partially static data structures such as constructors and selectors) will give poor results: V becomes dynamic due to the frozen let-expression binding V2. Hence operations in E_3 on the data structure created by the cons will not be performed at partial evaluation time: the cons will be frozen and so will the car and cdr operations in E_3 that operate on V.

But with a cps specializer such as $\mathcal{C}^2$ (extended to handle operations on partially static data structures), the context is propagated through the frozen let-expression binding V2. The frozen let-expression does *not* cause the cons to be frozen. Operations such as car and cdr in the context can now be reduced at partial evaluation time. For example, the result of specializing E_{00} will be

$$\texttt{(let ((V2 } E_2'\texttt{)) } E_1'\texttt{)} \textit{ if } E_3 = \texttt{(car V)}$$

and

$$\texttt{(let ((V2 } E_2'\texttt{)) (+ V2 V2))} \textit{ if } E_3 = \texttt{(+ (cdr V) (cdr V))}$$

Thus we can at the same time provide partially static data structures and *still preserve safety*: no residual expression that becomes part of a partially static data structure is ever duplicated or discarded!

Based on this idea, we have implemented an extension of Similix 4.0 that handles partially static data structures. Instead of rewriting the source program, the extension inserts the needed additional let-expressions during specialization, thus not burdening the user with transformed source code (cf. section 6.3).

The extension handles arbitrary n-ary user defined constructors, not just cons. This enables handling "disjoint sum of product" values where the binding times of arguments to different constructors do not get intermingled: the tag of a constructed value determines the binding times of the components. This kind of binding time information is also treated in [Lau91], but not in [Mog88, Con90a] where only the constructor cons is handled.

7.1 Related work

In [Mog88], safety (ensuring that no residual expression that becomes part of a partially static data structure is ever duplicated or discarded) was obtained by keeping let-bindings in the partially static structures. Manipulating these let-bindings was extremely complex, and the operations were all dynamic when self-applying the specializer. Hence, the operations remained in the generated generating extensions.

In Fuse [WCRS91], the specializer generates residual code *graphs* rather than residual code expressions; a post-phase generates code expressions from the graph. Safety is obtained by inserting appropriate bindings during the post-phase. Generating residual code graphs works well for a non-self-applicable specializer such as Fuse, but is not well-suited for self-applicable specializers (such as Similix): the (costly) operation of generating code from the graph would be completely dynamic at self-application time and so would be present in all generated generating extensions.

The safety problem is not addressed in [Lau91, Con90a].

8 Conclusion

We have shown how one of the most important binding time improvement problems, that of propagating consumers to producers, can be solved by using a cps-based specializer. There is still a need for addressing binding time improvements, for instance polyvariant closure and binding time analyses.

Acknowledgements

Thanks to Fritz Henglein, Jesper Jørgensen, and Torben Æ. Mogensen who all contributed to this work; to the rest of the DIKU "TOPPS" group and to Peter Sestoft. Fritz Henglein and Olivier Danvy also gave useful suggestions for improving the presentation.

References

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global

variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[Bon91a] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming, LNCS 432)(1-3):3–34, December 1991.

[Bon91b] Anders Bondorf. *Similix Manual, system version 4.0.* DIKU, University of Copenhagen, Denmark, September 1991.

[CD91] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523*, pages 495–519, Springer-Verlag, August 1991.

[Con90a] Charles Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Languages. Nice, France*, pages 264–272, June 1990.

[Con90b] Charles Consel. *The Schism Manual.* Yale University, New Haven, Connecticut, USA, 1990. Version 1.0.

[Dan91] Olivier Danvy. Semantics-directed compilation of nonlinear patterns. *Information Processing Letters*, 37(6):315–322, 1991.

[Dan92] Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *ESOP'92, 4th European Symposium on Programming, Rennes, France. Lecture Notes in Computer Science 582*, pages 130–150, Springer-Verlag, February 1992.

[GJ91a] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[GJ91b] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[HG91] Carsten Kehler Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale University, New Haven, Connecticut. SIGPLAN Notices, volume 26, 9*, pages 223–233, ACM Press, June 1991.

[HH90] Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 83–100, Springer-Verlag, August 1990.

[IEE90] IEEE standard for the Scheme programming language. May 1990. IEEE Std 1178-1990.

[Jor90] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 177–195, Springer-Verlag, August 1990.

[Jor92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, January 1992.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[KS91] Siau Cheng Khoo and R.S. Sundaresh. Compiling inheritance using partial evaluation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale University, New Haven, Connecticut. SIGPLAN Notices, volume 26, 9*, pages 211–222, ACM Press, June 1991.

[Lau91] John Launchbury. *Projection Factorisations in Partial Evaluation. Distinguished Dissertations in Computer Science*, Cambridge University Press, 1991.

[Mog88] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347, North-Holland, 1988.

[Mog89] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989.

[NN88] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. San Diego, California*, pages 98–106, 1988.

[Ses88] Peter Sestoft. Automatic call unfolding in a partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506, North-Holland, 1988.

[Ste78] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Technical Report AI-TR-474, MIT, Cambridge, Massachusetts, Cambridge, Massachusetts, May 1978.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523*, pages 165–191, Springer-Verlag, August 1991.