Call-By-Name CPS-Translation as a Binding-Time Improvement

Kristian Nielsen & Morten Heine Sørensen

DIKU (Department of Computer Science, University of Copenhagen)
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.
E-mail: {bombadil,rambo}@diku.dk

Abstract. Much attention has been given to the call-by-value continuation passing style (CBV CPS) translation as a tool in partial evaluation, but the call-by-name (CBN) CPS translation has not been investigated. We undertake a systematic investigation of the effect of CBN CPS in connection with partial evaluation and deforestation.

First, we give an example where CBN CPS translation acts as a binding time improvement to achieve the effects of deforestation using partial evaluation. The same effect cannot be achieved with CBV CPS. Second, we prove formally that the CBN CPS translation together with partial evaluation has the power to achieve *all* the effects of deforestation.

The consequence of these results is a practical tool (the CBN CPS) for improving the results of partial evaluation, as well as an improved understanding of the relation between partial evaluation and deforestation.

1 Introduction

Partial evaluation [Jon93] is a program transformation technique that can automatically perform a variety of program improvements. However, partial evaluation has had only partial success in the elimination of intermediate data structures. For this particular problem deforestation [Wad90] is more applicable.

For a standard example of elimination of intermediate data structures, let a be the function for appending two lists. The expression $a\ (a\ xs\ ys)\ zs$ constructs an intermediate list for the result of $a\ xs\ ys$. A more efficient, equivalent expression is $a\ xs\ (a\ ys\ zs)$, which does not construct the intermediate list. Deforestation can transform the former into the latter, whereas partial evaluation cannot.

The CBV CPS translation [Plo75] has been used to partially solve this shortcoming of partial evaluation. Consel and Danvy [Con91, Dan91] show that CBV CPS translating the source program before applying partial evaluation can improve specialization. For instance, a typical partial evaluator will return the term $(\textbf{if}\ x = 0\ \textbf{then}\ 9\ \textbf{else}\ 13) + 7$ unchanged if x is dynamic. CBV CPS translation of the term gives roughly $\textbf{if}\ x = 0\ \textbf{then}\ (\lambda v.v + 7)\ 9\ \textbf{else}\ (\lambda v.v + 7)\ 13$ which partial evaluation can transform to $\textbf{if}\ x = 0\ \textbf{then}\ 16\ \textbf{else}\ 20$. A similar effect is obtained if the partial evaluator adopts an explicit *context distribution* rule that moves the context $+7$ into the branches of the conditional. Bondorf [Bon92]

shows how such manipulations can be incorporated elegantly by writing the specializer itself in CPS. Lawall and Danvy adopt a language with control operators to include such manipulations in a direct style (DS) specializer [Law94].

However, these methods "cannot handle dynamic, recursive data structures" [Con91], so they do not achieve the full power of deforestation. In the example they work because the intermediate structure is a number (9 and 13), but in the double-append example they fail, because the intermediate structure is a *list*.

In this paper, we investigate the effect of the CBN CPS translation on partial evaluation, and its connection to deforestation. We show that by using CBN, rather than CBV CPS, it is possible to eliminate intermediate data with a partial evaluator, even dynamic, recursive ones. Section 2 presents an example use of the CBN CPS translation with the partial evaluator Similix to obtain non-trivial deforestation of a program. The remaining sections prove in detail that CBN CPS-translation followed by partial evaluation gives the same result as deforestation of the original program.

2 CBN CPS as a Binding-Time Improvement

In this section we present a case study[1] of the use of CBN CPS with partial evaluation to obtain better specialization. All experiments have been performed using Similix [Bon90] which is a partial evaluator for Scheme. For readability we present programs in a Miranda/ML-like syntax. The section is informal in style and assumes some familiarity with partial evaluation and CPS-translation corresponding to for instance [Jon93] or [Con91].

2.1 Compiler and Interpreter for Algebraic Expressions

We consider the language E of *algebraic expressions* as given by the grammar

$$e ::= n \mid e + e \mid -e$$

(n denotes a numerical constant) and a *stack machine* language S:

$$s ::= \mathsf{push}\ n \mid \mathsf{add} \mid \mathsf{neg} \mid s_1; s_2$$

with instructions for pushing constants onto the stack, adding the two top elements of the stack, negating the top of the stack, and for sequencing.

Figure 1 defines a compiler *comp* from expressions to stack-machine code (ie. from E to S), and an interpreter *int-stack* for S. The function *comp* is a simple syntax-directed translation. The function *int-stack* executes each instruction in turn while maintaining the stack, and returns the top of the stack as the result of evaluation.

Using *comp* and *int-stack* we can define a stack-based interpreter *int-exp* for E as the composition of the two functions (Fig. 1). However, this interpreter is inefficient, since it first constructs an intermediate compiled program which

[1] The example is inspired by work by Dussart [Dus95].

```
comp (ConstE n)   = Push n
comp (NegE e)     = Seq (comp e) Neg
comp (AddE e1 e2) = Seq (comp e1) (Seq (comp e2) Add)

int-stack ι = hd (int' ι [])
  where
    int' (Push n) s       = n : s
    int' Neg (n : s)      = (−n) : s
    int' Add (n : m : s)  = (m + n) : s
    int' (Seq ι1 ι2) s    = int' ι2 (int' ι1 s)

int-exp e = int-stack (comp e)
```

Fig. 1. Stack machine compiler and interpreter.

is then interpreted by *int-stack*. *Our goal* is to use partial evaluation to automatically transform the expression *int-exp* e with dynamic e into a one-pass interpreter that does not construct an intermediate program.

2.2 The Failure of CBV Techniques

Applying Similix to the program in Fig. 1 yields the program back unchanged — no improvement is achieved. Consel and Danvy [Con91] explain this by saying that the producer (*comp*) and the consumer (*int-stack*) are separated by a dynamic conditional (the dynamic cases on the expression syntax in *comp*), and they describe how the CBV CPS translation can solve such problems by bringing the producer and the consumer together, similarly to the example in the introduction. However, they also note that the technique cannot handle recursive, dynamic data structures (such as the one in *comp*), since they cause infinite specialization with respect to a growing continuation.

Indeed, when Similix is applied to a CBV CPS version of the program, the binding time analysis indicates that the intermediate data structure can now be eliminated, but infinite specialization is the result. In order to ensure termination, Similix would have to generalize the growing continuation, in which case the program is not improved.

2.3 The Effect of CBN Techniques

In [Con91] the CBV CPS translation was used exclusively; similar results were *conjectured* for the CBN CPS. In this subsection we show that, in fact, the key to success in our example is the use of CBN, rather than CBV CPS.

If we want to apply the CBN CPS translation to a program with a CBV semantics we must make sure that the translated program is equivalent to the original. The problem here is that a program p and its CBN CPS translated counter-part $\underline{p}$ are not necessarily equivalent under a CBV semantics: (*i*) side effects may happen in a different order in $\underline{p}$ than in p, (*ii*) $\underline{p}$ may be less efficient

than p because of duplicated computation, and (*iii*) p may be non-terminating while $\underline{p}$ is terminating.

Of these, (*iii*) is probably not a problem in practice; for instance, some transformers choose to ignore the possibility of changing non-terminating programs into terminating ones, *e.g.* Fuse [Wei91] and the supercompiler [Tur86]. (*ii*) is well-known in deforestation.

Figure 2 gives a CBN CPS version of the program in Fig. 1. The idea in CBN CPS is that every expression translates into a function $\lambda\kappa.t$ that expects a *continuation* argument. Applying the function to a continuation κ evaluates the expression to weak head normal form and applies κ to the result. By keeping all parts that will appear in the residual program in direct style (the input variables, as well as the stack and the result of *int-stack*) we obtain a residual program in DS.

```
comp (ConstE n) κ    = κ (Push n)
comp (NegE e) κ      = κ (Seq (comp e) (λκ.κ Neg))
comp (AddE e1 e2) κ  = κ (Seq (comp e1) (λκ.κ (Seq (comp e2) (λκ.κ Add))))

int-stack ι = hd (int' ι [])
  where
    int' ι s = ι (λv.
      case (v, s) of
        (Push n, s)      = n : s
        (Neg, n : s)     = (−n) : s
        (Add, n : m : s) = (m + n) : s
        (Seq ι1 ι2, s)   = int' ι2 (int' ι1 s))

int-exp e = int-stack (comp e)
```

Fig. 2. CBN CPS version of two-pass interpreter.

Applying Similix to the program in Fig. 2 gives the desired, efficient, one-pass stack-based interpreter (Fig. 3). By rewriting the program into CBN CPS we have succeeded in eliminating a dynamic intermediate data structure using Similix, even though the data structure is recursive.

```
int-exp e = hd (int' e [])
  where
    int' (ConstE n) s             = n : s
    int' (NegE e) s               = let (n : s') = int' e s in (−n) : s'
    int' (AddE e1 e2) (n : m : s) = let (n : m : s') = int' e2 (int' e1 s) in (m + n) : s'
```

Fig. 3. Efficient, specialized interpreter.

2.4 Summary and Overview

We have seen that is some cases where it is safe to apply the CBN CPS translation, better results are obtained with partial evaluation than when using eg. CBV CPS. In the following sections we characterize this phenomenon more generally by proving that partial evaluation of a CBN CPS translated program is equivalent to deforestation of the original program.

The structure of the proof is as follows. We first define *unfolding relations* $\Rightarrow_n$ and $\Rightarrow_v$ for a CBN and a CBV transformer corresponding to deforestation and partial evaluation, respectively (Sects. 4 and 6). For these we prove *simulation and indifference* (Sects. 5 and 7); that unfolding for either transformer on programs in CBN CPS is the same as CBN unfolding on programs in DS. To simplify the presentation, simulation is proved only for a first-order language; the proof extends to the higher order case (deforestation is usually restricted to first-order programs anyway). Finally (Sect. 8) we introduce *folding* and extend our simulation and indifference results to deal with folding.

3 Language and Notational Conventions

We use the language in Fig. 4; it is the lambda-calculus extended with recursive function definitions, constructors, and case expressions on simple patterns (written with "?" for brevity). We assume denumerable, disjoint sets of variable names, fixed arity constructor names, and fixed arity function names. The *first-order fragment* of our language is the language obtained by omitting lambda abstractions and applications in Fig. 4.

$$
\begin{array}{l}
x \in \mathsf{TermVar}, c \in \mathsf{Constr}, f \in \mathsf{FName}, t \in \mathsf{Term} \\
t ::= x \mid \lambda x.t \mid c\ t_1 \cdots t_n \mid f\ t_1 \cdots t_n \mid t_1\ t_2 \mid t\,?\,\{p_i \to t_i\}_{i=1}^{n} \\
p ::= c\ x_1 \cdots x_n \\
d ::= f\ x_1 \cdots x_m \triangleq t
\end{array}
$$

Fig. 4. Language

A program is a set of definitions $\{d_1 \ldots d_n\}$. As usual we require that no variable occurs more than once in a definition's left hand side or in a pattern. We also require that all variables in a definitions right hand side be present in its left side. We require that each function in a program have only one definition, and that the patterns in a case-expression be distinct. Finally, constructors and function calls must be supplied with the correct number of arguments.

We use the vector notation $\mathbf{t}$ for a list of terms $t_1 \cdots t_n$. For instance $c\ \mathbf{t}$ means $c\ t_1 \cdots t_n$. In case expressions $\mathbf{p} \to \mathbf{t}$ abbreviates $p_1 \to t_1 \cdots p_n \to t_n$. We use $t'\{x := t\}$ to denote the result of replacing free occurrences of x in t' by t,

and $t\{\mathbf{x} := \mathbf{t}\}$ denotes simultaneous substitution. Application associates to the left, so $x\ t_1 \cdots t_n$ abbreviates $(\cdots(x\ t_1)\cdots t_n)$.

For simplicity we do not consider "error terms" — terms of form $(c\ \mathbf{t})\ t'$ (application of a constructor term), $(\lambda x.t)\,?\,\{p_i \to t_i\}_{i=1}^n$ (case on a lambda-abstraction), and $(c\ \mathbf{t})\,?\,\{p_i \to t_i\}_{i=1}^n$ where no pattern has form $c\ \mathbf{x}$. In such cases our transformers simply halt. Our development extends to the general case without these restrictions without any fundamental changes, but at the cost of some rather more complicated definitions.

The CBN CPS translation, based on the translation in [Plo75], is defined in Fig. 5 for the higher order language. Programs are CPS translated by translating the right hand sides of function definitions:

$$\underline{f\ x_1 \cdots x_n \triangleq t} \quad = \quad f\ x_1 \cdots x_n \triangleq \underline{t}$$

$$\begin{aligned}
\underline{x} &= x \\
\underline{\lambda x.t} &= \lambda\kappa.\kappa\ (\lambda x.\underline{t}) \\
\underline{c\ t_1 \cdots t_n} &= \lambda\kappa.\kappa\ (c\ \underline{t_1} \cdots \underline{t_n}) \\
\underline{f\ t_1 \cdots t_n} &= \lambda\kappa.(f\ \underline{t_1} \cdots \underline{t_n})\ \kappa \\
\underline{t_1\ t_2} &= \lambda\kappa.\underline{t_1}\ (\lambda v.v\ \underline{t_2}\ \kappa) \\
\underline{t\,?\,\{p_i \to t_i\}_{i=1}^n} &= \lambda\kappa.\underline{t}\ (\lambda v.v\,?\,\{p_i \to \underline{t_i}\ \kappa\}_{i=1}^n)
\end{aligned}$$

Fig. 5. CBN CPS translation

4 CBN Unfolding

In this section we introduce CBN unfolding both for our higher-order language and for the first-order fragment. For the first-order fragment the unfolding mechanism is essentially the unfolding in Wadler's deforestation [Wad90]. The higher-order unfolding is similar to the one higher-order deforestation [Mar92, Ham94, San95]; we use a presentation similar to the one in [San95].

We introduce the unfolding mechanism as a rewrite relation $\Rightarrow_n$ on $\mathsf{Term} \times \mathsf{Term}$ that generalizes the normal CBN rewrite semantics on closed terms; unknown (dynamic) values are represented by free variables.

The sets $\mathsf{Ctxt_n}$ and $\mathsf{Redex_n}$ are traditional CBN evaluation contexts and CBN redexes, respectively; $t \equiv e[r]$ means that r is the next CBN redex in t. If t does not have form $e[r]$, ie. normal CBN evaluation has terminated, then t has an outermost λ or constructor, or t is a stuck application or case expression. These may still have redexes. $\mathsf{DVal_n}$ is the set of terms without any redexes. If a term does not have form $e[r]$ but is not in $\mathsf{DVal_n}$ it will have form $\mathcal{E}[e[r]]$ where $\mathcal{E} \in \mathsf{DCtxt_n}$ is a dead context; these allow us to get hold of subterms $e[r]$ under

$$
\begin{array}{l}
e \in \mathsf{Ctxt_n}, r \in \mathsf{Redex_n}, \mathcal{V} \in \mathsf{DVal_n}, \mathcal{E} \in \mathsf{DCtxt_n} \\
e \ ::= [\,] \mid e\ t \mid e\,?\,\{\mathbf{p} \to \mathbf{t}\} \\
r \ ::= (\lambda x.t_1)\ t_2 \mid c\ \mathbf{t'}\,?\,\{\mathbf{p} \to \mathbf{t}\} \mid f\ \mathbf{t} \mid (x\ t_1 \cdots t_m\,?\,\{\mathbf{p} \to \mathbf{t}\})\ t' \\
\quad \mid \ (x\ t_1 \cdots t_n\,?\,\{\mathbf{p} \to \mathbf{t}\})\,?\,\{\mathbf{p'} \to \mathbf{t'}\} \\
\mathcal{V} ::= x\ \mathcal{V}_1 \cdots \mathcal{V}_n \mid \lambda x.\mathcal{V} \mid c\ \boldsymbol{\mathcal{V}} \mid (x\ \mathcal{V}'_1 \cdots \mathcal{V}'_n)\,?\,\{\mathbf{p} \to \boldsymbol{\mathcal{V}}\} \\
\mathcal{E} ::= [\,] \mid \lambda x.\mathcal{E} \mid c\ \boldsymbol{\mathcal{V}}\ \mathcal{E}\ \mathbf{t} \mid x\ \mathcal{V}_1 \cdots \mathcal{V}_m\ \mathcal{E}\ t_1 \cdots t_n \mid (x\ \mathcal{V}_1 \cdots \mathcal{V}_m\ \mathcal{E}\ t_1 \cdots t_n)\,?\,\{\mathbf{p} \to \mathbf{t}\} \\
\quad \mid \ (x\ \mathcal{V}_1 \cdots \mathcal{V}_l)\,?\,\{p_1 \to \mathcal{V}_1; \ldots; p_m \to \mathcal{V}_m; p \to \mathcal{E}; p_1 \to t_1; \ldots; p_n \to t_n\}
\end{array}
$$

Fig. 6. Higher-order CBN contexts

a lambda, under a constructor, under stuck applications and in the branch of a stuck case expression. This is all formalized in Fig. 6.

The definition satisfies the following central *unique decomposition property:*

Lemma 1. *For any* $t \in \mathsf{Term}$*, either* $t \in \mathsf{DVal_n}$*; or there exists a unique triple* $(\mathcal{E}, e, r) \in \mathsf{DCtxt_n} \times \mathsf{Ctxt_n} \times \mathsf{Redex_n}$ *such that* $t = \mathcal{E}[e[r]]$

Proof. Use induction on the structure of the term t.□

Lemma 1 allows us to make definitions by induction on the structure of the unique decomposition into $\mathcal{V}$ respectively $\mathcal{E}[e[r]]$. The CBN unfolding relation is defined in Fig. 7. The relation $\mapsto_n$ defines pure evaluation; $\Rightarrow_n$ contains $\mapsto_n$, but extends it with operations to evaluate in normal as well as dead contexts. Note that both $\mapsto_n$ and $\Rightarrow_n$ are deterministic by virtue of Lemma 1.

$$
\begin{array}{lll}
e[(\lambda x.t_1)\ t_2] & \mapsto_n e[t_1\{x := t_2\}] & \\
e[c\ \mathbf{t'}\,?\,\{p_i \to t_i\}_{i=1}^n] & \mapsto_n [t_k\{\mathbf{x} := \mathbf{t'}\}] & \text{if } p_k = c\ \mathbf{x} \\
e[f\ \mathbf{t}] & \mapsto_n e[t'\{\mathbf{x} := \mathbf{t}\}] & \text{if } f\ \mathbf{x} \triangleq t' \text{ is a definition}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[e[(x\ t_1 \cdots t_m\,?\,\{p_i \to t_i\}_{i=1}^n)\ t']] \Rightarrow_n \mathcal{E}[x\ t_1 \cdots t_m\,?\,\{p_i \to e[t_i\ t']\}_{i=1}^n] \\
\mathcal{E}[e[(x\ t_1 \cdots t_l\,?\,\{p_i \to t_i\}_{i=1}^n)\,?\,\{p'_j \to t'_j\}_{j=1}^m]] \\
\qquad \Rightarrow_n \mathcal{E}[x\ t_1 \cdots t_l\,?\,\{p_i \to e[t_i\,?\,\{p'_j \to t'_j\}_{j=1}^m]\}_{i=1}^n] \\
\mathcal{E}[e[r]] \Rightarrow_n \mathcal{E}[e[t]] \qquad \text{if } e[r] \mapsto_n e[t]
\end{array}
$$

Fig. 7. Higher-order CBN unfolding

Since we state the simulation result for the first-order language subset, we need the restriction of $\Rightarrow_n$ to first-order terms. This is defined in Figs. 8 and 9. The definition is justified by a unique decomposition property in a similar way to the higher-order case.

$$
\begin{array}{l}
e \in \mathsf{Ctxt}_1, r \in \mathsf{Redex}_1, \mathcal{V} \in \mathsf{DVal}_1, \mathcal{E} \in \mathsf{DCtxt}_1 \\
e \ ::= [\,] \mid e\,?\,\{\mathbf{p} \to \mathbf{t}\} \\
r \ ::= c\ \mathbf{t}'\,?\,\{\mathbf{p} \to \mathbf{t}\} \mid f\ \mathbf{t} \mid (x\,?\,\{\mathbf{p} \to \mathbf{t}\})\,?\,\{\mathbf{p}' \to \mathbf{t}'\} \\
\mathcal{V} ::= x \mid c\ \boldsymbol{\mathcal{V}} \mid x\,?\,\{\mathbf{p} \to \boldsymbol{\mathcal{V}}\} \\
\mathcal{E} ::= [\,] \mid c\ \boldsymbol{\mathcal{V}}\ \boldsymbol{\mathcal{E}}\ \mathbf{t} \mid x\,?\,\{p_1 \to \mathcal{V}_1; \ldots; p_m \to \mathcal{V}_m; p \to \mathcal{E}; p_1 \to t_1; \ldots; p_n \to t_n\}
\end{array}
$$

Fig. 8. First-order CBN contexts

$$
\begin{array}{lll}
e[c\ \mathbf{t}'\,?\,\{p_i \to t_i\}_{i=1}^n] \mapsto_1 e[t_k\{\mathbf{x} := \mathbf{t}'\}] & & \text{if } p_k = c\ \mathbf{x} \\
e[f\ \mathbf{t}] \mapsto_1 e[t'\{\mathbf{x} := \mathbf{t}\}] & & \text{if } f\ \mathbf{x} \triangleq t' \text{ is a definition} \\
\\
\mathcal{E}[e[(x\,?\,\{p_i \to t_i\}_{i=1}^n)\,?\,\{p'_j \to t'_j\}_{j=1}^m]] \Rightarrow_1 \mathcal{E}[x\,?\,\{p_i \to e[t_i\,?\,\{p'_j \to t'_j\}_{j=1}^m]\}_{i=1}^n] & & \\
\mathcal{E}[e[r]] \Rightarrow_1 \mathcal{E}[e[t]] & & \text{if } e[r] \mapsto_1 e[t]
\end{array}
$$

Fig. 9. First-order CBN unfolding

5 Simulation

In this section we prove that $\Rightarrow_n$ on CBN CPS translated terms simulates $\Rightarrow_1$ on DS terms. Letting $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$, we prove that

$$\text{if } t_1 \Rightarrow_1^* t_2 \text{ then } \underline{t_1} \Rightarrow_n^* u$$

where u is essentially equivalent to $\underline{t_2}$.

For this, we classify reductions in the CPS world in two classes: administrative reductions and proper reductions. Each reduction in direct style corresponds in CPS to zero or more administrative reductions followed by a proper reduction. Intuitively, the proper reductions perform the "real" computation, while the administrative reductions form an "overhead" introduced by the CPS translation. Using so-called colon-translations to associate with each DS redex a proper CPS redex, the definition of "essentially equivalent" can be made precise.

The proof is in two parts, corresponding to the normal and dead contexts of the unfolding mechanisms. The first part concerns evaluation, and is a restatement of Plotkin's proof for the language used here [Plo75]. The second part extends the proof to dead contexts.

The first part uses the *first colon translation* in Fig. 10. It consists of the following three lemmas (ℓ ranges over terms):

Lemma 2. *For all terms t and t', $\underline{t}\{x := \underline{t'}\} = \underline{t\{x := t'\}}$.*

Lemma 3. *For all terms t and ℓ, $\underline{t}\ \ell \mapsto_n^* t : \ell$.*

Lemma 4. *For all terms t_1 and ℓ, if $t_1 \mapsto_1 t_2$ then $t_1 : \ell \mapsto_n^* t_2 : \ell$.*

The proofs of these lemmas are all straightforward inductions (the last lemma uses the two first ones), see [Plo75].

$$
\begin{aligned}
x : \ell &= x\ \ell \\
c\ t_1 \cdots t_n : \ell &= \ell\ (c\ \underline{t_1} \cdots \underline{t_n}) \\
f\ t_1 \cdots t_n : \ell &= (f\ \underline{t_1} \cdots \underline{t_n})\ \ell \\
c\ t_1 \cdots t_n\ ?\ \{p_i \to t_i\}_{i=1}^n : \ell &= (c\ \underline{t_1} \cdots \underline{t_n})\ ?\ \{p_i \to \underline{t_i}\ \ell\}_{i=1}^n \\
t\ ?\ \{p_i \to t_i\}_{i=1}^n : \ell &= t : (\lambda v.v\ ?\ \{p_i \to \underline{t_i}\ \ell\}_{i=1}^n) \qquad t \neq c\ t_1 \cdots t_n
\end{aligned}
$$

Fig. 10. First colon translation

The first colon translation assigns to each DS term t a unique term $t : \ell$ which is the result of reducing administrative redexes in $\underline{t}\ \ell$. To extend simulation to the unfolding relation $\Rightarrow_1$ the second colon translation must instead assign a (finite) *set* of possible terms with administrative redexes reduced.

For example, $c\ ?\ \{c \to x\} \Rightarrow_1 x$ but $c\ ?\ \{c \to x\} \Rightarrow_1 \lambda\kappa.x\ \kappa$, so the set of terms assigned to x must include both x and $\lambda\kappa.x\ \kappa$; likewise, administrative redexes in branches of stuck case expressions may be reduced in different orders. This complicates the second part of the proof somewhat. Also, the residual program may contain subexpressions $\lambda\kappa.x\ \kappa$ instead of just x, so simulation only holds up to η-conversion of variables (this problem is also present in Plotkin's translation proof in [Plo75], as was recently discovered by Hatcliff and Danvy [Hat95]).

The second colon translation is defined in Fig. 11. Three different mappings are used (we use, eg. $c\ T_1 \cdots T_n$ to denote the set $\{c\ t_1 \cdots t_n \mid t_i \in T_i\}$):

- $t \star \ell$ for the set of terms corresponding to the first proper redex in $\underline{t}\ \ell$.
- $\widetilde{t}$ for the set of terms corresponding to the first proper redex in $\underline{t}$.
- $t \cdot \ell$ for the set of terms that can appear instead of $\underline{t}\ \ell$ in residual case expressions.

$$
\begin{aligned}
&e[r] \star \kappa = \{e[r] : \kappa\} \\
&x \star \kappa = \{x\ \kappa\} \\
&(c\ \mathcal{V}_1 \cdots \mathcal{V}_m\ t_1 \cdots t_n) \star \kappa = \kappa\ (c\ \widetilde{\mathcal{V}}_1 \cdots \widetilde{\mathcal{V}}_m\ \widetilde{t_1}\ \underline{t_2} \cdots \underline{t_n}) \\
&(x\ ?\ \{p_1 \to \mathcal{V}_1; \ldots; p_m \to \mathcal{V}_m; p'_1 \to t_1; \ldots; p'_n \to t_n\}) \star \kappa \\
&= x\ (\lambda v.v\ ?\ \{p_1 \to \mathcal{V}_1 \star \kappa; \ldots; p_m \to \mathcal{V}_m \star \kappa; p'_1 \to t_1 \star \kappa; p'_2 \to t_2 \cdot \kappa; \ldots; p'_n \to t_n \cdot \kappa\})
\end{aligned}
$$

(Here, $t_1 \notin \mathsf{DVal}_1$ if $n > 1$.)

$$
\widetilde{t} = \begin{cases} \{x, \lambda\kappa.x\ \kappa\} & \text{if } t = x \\ \lambda\kappa.t \star \kappa & \text{otherwise} \end{cases}
$$

$$
t \cdot \ell = \begin{cases} \{\underline{t}\ \ell\} \cup (t' \cdot \lambda v.v\ ?\ \{p_i \to \underline{t_i}\ \ell\}_{i=1}^n) & \text{if } t = t'\ ?\ \{p_i \to t_i\}_{i=1}^n \\ \{\underline{t}\ \ell\} & \text{otherwise} \end{cases}
$$

Fig. 11. Second colon translation.

First we need to prove a number of properties about the colon translations.

Lemma 5. *For any* $\mathcal{V}$, $\widetilde{\mathcal{V}} \subset \mathsf{DVal_n}$ *and* $\mathcal{V} \star \kappa \subset \mathsf{DVal_n}$.

Proof. A simple induction on $\mathcal{V}$.□

Lemma 6. $e[x\,?\,\{p_i \to t_i\}_{i=1}^n] : \ell = x(\lambda v.v\,?\,\{p_i \to u_i\}_{i=1}^n)$ *for some* $u_i \in e[t_i] \cdot \ell$.

Proof. Induction on e.□

Lemma 7. *For all* $u \in t \cdot \ell$, $u \mapsto_n^* t : \ell$.

Proof. Induction on the definition of $t \cdot \ell$, using Lemma 3.□

We can now prove the simulation property:

Lemma 8. *For all* t,

1. $t : \kappa \Rightarrow_n^* u$ *for some* $u \in t \star \kappa$
2. $\underline{t} \Rightarrow_n^* u$ *for some* $u \in \widetilde{t}$
3. $\underline{t}\ \kappa \Rightarrow_n^* u$ *for some* $u \in t \star \kappa$
4. *If* $u \in t \cdot \kappa$ *then* $u \Rightarrow_n^* u'$ *for some* $u' \in t \star \kappa$.

Proof. Simultaneous induction on t.

For part 1, a typical case is when $t = c\ \mathcal{V}_1 \cdots \mathcal{V}_m\ t_1 \cdots t_n$, with $t_1 \notin \mathsf{DVal_1}$ if $n > 1$. By induction hypothesis 2 we have $\underline{\mathcal{V}_i} \Rightarrow_n^* u_i \in \widetilde{\mathcal{V}_i}$ and $\underline{t_1} \Rightarrow_n^* u' \in \widetilde{t_1}$. By Lemma 5 $u_i \in \mathsf{DVal_n}$. Now

$$\begin{aligned} t : \kappa\ &=\ \kappa\ (c\ \underline{\mathcal{V}_1} \cdots \underline{\mathcal{V}_m}\ \underline{t_1} \cdots \underline{t_n}) \\ &\Rightarrow_n^* \kappa\ (c\ u_1 \cdots u_m\ u'\ \underline{t_2} \cdots \underline{t_n}) \in t \star \kappa. \end{aligned}$$

For part 2 use part 1 with Lemma 3. For part 3 use part 1 with Lemma 4. For part 4 use part 1 with Lemma 7.□

Lemma 9.

1. *If* $t_1 \Rightarrow_1 t_2$ *and* $u_1 \in t_1 \star \kappa$ *then* $u_1 \Rightarrow_n^* u_2$ *for some* $u_2 \in t_2 \star \kappa$.
2. *If* $t_1 \Rightarrow_1 t_2$ *and* $u_1 \in \widetilde{t_1}$ *then* $u_1 \Rightarrow_n^* u_2$ *for some* $u_2 \in \widetilde{t_2}$

Proof. Simultaneous induction on the definition of $\Rightarrow_1$.

For part 1 we prove the case when $t_1 = e[x\,?\,\{p_i \to t_i\}_{i=1}^n] \Rightarrow_1 x\,?\,\{p_i \to e[t_i]\}_{i=1}^n = t_2$ (with $e \neq [\,]$). Then by Lemma 6, $u_1 = t_1 : \kappa = x(\lambda v.v\,?\,\{p_i \to u_i'\}_{i=1}^n)$ where $u_i' \in e[t_i] \cdot \kappa$. If $e[t_i] \in \mathsf{DVal_1}$ for all i then let $k = n$; otherwise choose k minimal such that $e[t_k] \notin \mathsf{DVal_1}$. By Lemma 8 (4), $u_i' \Rightarrow_n^* u_i'' \in e[t_i] \star \kappa$, and by Lemma 5 $u_i'' \in \mathsf{DVal_n}$ for $i < k$. This means that

$$\begin{aligned} u_1\ &=\ x(\lambda v.v\,?\,\{p_i \to u_i'\}_{i=1}^n) \\ &\Rightarrow_n^* x(\lambda v.v\,?\,\{p_1 \to u_1''; \ldots; p_k \to u_k''; p_{k+1} \to u_{k+1}'; \ldots; p_n \to u_n'\}) \in t_2 \star \kappa. \end{aligned}$$

For part 2, use part 1.□

Lemma 8 says that $\underline{t} \Rightarrow_n^* u \in \widetilde{t}$; the result of reducing administrative redexes in $\underline{t}$ is in $\widetilde{t}$. Lemma 9 then says that unfolding is preserved by CBN CPS translation: if $t_1 \Rightarrow_1 t_2$ then $\widetilde{t_1} \ni u_1 \Rightarrow_n^* u_2 \in \widetilde{t_2}$. Thus we have the simulation result:

Theorem 10 (Simulation). *If* $t_1 \Rightarrow_1^* t_2$ *then* $\underline{t_1} \Rightarrow_n^* u$ *for some* $u \in \widetilde{t_2}$.

6 CBV Unfolding

We now introduce CBV unfolding. The set $\mathsf{Val_v}$, ranged over by v, consists of the usual CBV values $\lambda x.t$ and $c\,\mathbf{v}$ as well as dynamic terms Dyn (stuck applications and case expressions). Relative to this set of values, $\mathsf{Ctxt_v}$ and $\mathsf{Redex_v}$ are the usual sets of CBV contexts and redexes; $t \equiv e[r]$ means that r is the next redex in t.

Values and dynamic terms may contain redexes; for instance $\lambda x.((\lambda y.y)\ z)$ is a value with a redex. $\mathsf{DVal_v}$ and DDyn are the sets of values and dynamic terms, respectively, without any redexes. If a term is in $\mathsf{Val_v}\backslash\mathsf{DVal_v}$ or $\mathsf{Dyn}\backslash\mathsf{DDyn}$ it will have form $\mathcal{E}[e[r]]$ where $\mathcal{E} \in \mathsf{DCtxt_v}$ is a dead context; these allow us to get hold of subterms $e[r]$ under a lambda or in the branch of a residual case term. This is all formalized in Fig. 12.

$r \in \mathsf{Redex_v}; v \in \mathsf{Val_v}; e \in \mathsf{Ctxt_v}; q \in \mathsf{Dyn}; \mathcal{V} \in \mathsf{DVal_v}; \mathcal{E} \in \mathsf{DCtxt_v}; \mathcal{Q} \in \mathsf{DDyn};$

$$
\begin{array}{lll}
r & ::= & f\,\mathbf{v} \mid (\lambda x.t)\,v \mid (c\,\mathbf{v})\,?\{\mathbf{p} \to \mathbf{t}\ \} \\
e & ::= & [] \mid e\,t \mid v\,e \mid f\,\mathbf{v}\,e\,\mathbf{t} \mid c\,\mathbf{v}\,e\,\mathbf{t} \mid e\,?\{\mathbf{p} \to \mathbf{t}\ \} \\
v & ::= & \lambda x.t \mid c\,\mathbf{v} \mid q \\
q & ::= & x \mid q\,v \mid q\,?\{\mathbf{p} \to \mathbf{t}\ \} \\
 & & \\
\mathcal{E} & ::= & [] \mid \mathcal{E}' \\
\mathcal{E}' & ::= & \lambda x.\mathcal{E} \mid c\,\boldsymbol{\mathcal{V}}\,\mathcal{E}'\,\mathbf{v} \mid \mathcal{E}'' \\
\mathcal{E}'' & ::= & \mathcal{E}''\,v \mid \mathcal{Q}\,\mathcal{E}' \mid \mathcal{E}''\,?\{\mathbf{p} \to \mathbf{t}\ \} \mid \mathcal{Q}\,?\{\mathbf{p} \to \boldsymbol{\mathcal{V}}; p' \to \mathcal{E}; \mathbf{p''} \to \mathbf{t}\ \} \\
\mathcal{V} & ::= & \lambda x.\mathcal{V} \mid c\,\boldsymbol{\mathcal{V}} \mid \mathcal{Q} \\
\mathcal{Q} & ::= & x \mid \mathcal{Q}\,\mathcal{V} \mid \mathcal{Q}\,?\{\mathbf{p} \to \boldsymbol{\mathcal{V}}\ \}
\end{array}
$$

Fig. 12. CBV contexts, redexes, values

It is not too hard to see that a unique decomposition property holds. Similarly to CBN unfolding we can use this property to define CBV unfolding in Fig. 13. $\Rightarrow_v$ is similar to the unfolding mechanism of Similix [Bon90]. Note, *e.g.* that arguments of function calls and constructors are unfolded before the call is unfolded, and that there are no rules for distributing contexts.

$$
\begin{array}{lll}
f\,\mathbf{v} & \mapsto_v t\{\mathbf{x} := \mathbf{v}\} & \text{if } f\,\mathbf{x} \triangleq t \text{ is a definition} \\
(\lambda x.t)\,v & \mapsto_v t\{x := v\} & \\
(c\,\mathbf{v})\,?\{\mathbf{p} \to \mathbf{t}\} & \mapsto_v t_j\{\mathbf{x} := \mathbf{v}\} & \text{if } p_j \equiv c\,\mathbf{x} \\
 & & \\
\mathcal{E}[e[r]] & \Rightarrow_v \mathcal{E}[e[t]] & \text{if } r \mapsto_v t
\end{array}
$$

Fig. 13. CBV unfolding

7 Indifference

We now show that the CBN and CBV unfolding mechanisms have the same behaviour on CPS-translated terms, *i.e.* for all $t \in \mathsf{Term}$:

$$\underline{t} \Rightarrow_n^* t' \text{ iff } \underline{t} \Rightarrow_v^* t'$$

First, we give a syntax of CPS-translated terms (Lemma 11). Second, for any term t in this syntax, either unfolding stops in both CBN and CBV, or the unfolding steps are the same in CBN and CBV. (Lemma 14). Third, the syntactic class is closed under unfolding by both transformers (Lemma 16), so after one unfolding step the new term is again in the class, and so on.

Step 1. The class of CPS-terms appears in Fig. 14. We assume a set $\mathsf{ContVar}$ of continuation variable names.

Lemma 11. *For all $t \in \mathsf{Term}$: $\underline{t} \in \mathsf{Term_c}$.*

Proof. Induction on t.□

$$\begin{array}{l} x \in \mathsf{TermVar}; t \in \mathsf{Term_c}; u, k \in \mathsf{ContVar}; l \in \mathsf{Cont}; \\ \\ t ::= \lambda k.(f\ \mathbf{t})\ l \mid \lambda k.((\lambda x.t)\ t)\ l \mid \lambda k.(c\ \mathbf{t})\,?\,\{p_i \to t_i\ l_i\}_{i=1}^n \\ \qquad \mid \lambda k.t\ l \mid \lambda k.l\ (c\ \mathbf{t}) \mid \lambda k.l\ \lambda x.t \mid x \\ l ::= \lambda u.u\ t\ l \mid \lambda u.u\,?\,\{p_i \to t_i\ l_i\}_{i=1}^n \mid k \end{array}$$

Fig. 14. Syntax of CPS terms

Step 2. First we introduce CPS-contexts, CPS-redexes, and CPS-values in Fig. 15, and show that any CPS-term is a CPS-value or decomposes uniquely into CPS-context and redex (Lemma 12). Then we show that a CPS-value is a dead value according to both CBN and CBV, and any decomposition $\mathcal{E}[r]$ in CPS has form $\mathcal{E}'[e'[r']]$ where $\mathcal{E}', e', r'$ are a dead context, a context, and a redex in both CBN and CBV (Lemma 13). This gives the desired (Lemma 14).

Lemma 12. *For all $t \in \mathsf{Term}$:*

1. *if $t \in \mathsf{Term_c}$ then either $t \in \mathsf{Val_c}$; or there is a unique pair $(\mathcal{E}, r) \in \mathsf{Ctxt_c} \times \mathsf{Redex_c}$ with $t \equiv \mathcal{E}[r]$.*
2. *if $t \in \mathsf{Cont}$ then either $t \in \mathsf{ContVal}$; or there is a unique pair $(\mathcal{F}, r) \in \mathsf{ContCtxt} \times \mathsf{Redex_c}$ with $t \equiv \mathcal{F}[r]$.*

Proof. Prove 1–2 simultaneously by induction on t.□.

Lemma 13.

1. $\mathsf{Val_c} \subseteq \mathsf{DVal_n} \cap \mathsf{DVal_v}$.

$x \in \mathsf{TermVar}; s \in \mathsf{Term}; k, h \in \mathsf{ContVar}; t \in \mathsf{Term_c}; l \in \mathsf{Cont}; r \in \mathsf{Redex_c};$
$\mathcal{V} \in \mathsf{Val_c}; \mathcal{U} \in \mathsf{ContVal}; \mathcal{E} \in \mathsf{Ctxt_c}; \mathcal{F} \in \mathsf{ContCtxt}$

$$r ::= f\ \mathbf{v} \mid (\lambda x.t)\ t' \mid (c\ \mathbf{t})\ ?\{\mathbf{p} \to \mathbf{s}\ \} \mid (\lambda k.s)\ l \mid (\lambda k.s)\ (c\ \mathbf{t}) \mid (\lambda k.s)\ \lambda x.t$$

$$\mathcal{E} ::= \lambda k.[]\ l \mid \lambda k.[] \mid \lambda k.x\ \mathcal{F} \mid \lambda k.h\ (c\ \boldsymbol{\mathcal{V}}\ \mathcal{E}\ \mathbf{t}) \mid \lambda k.h\ \lambda x.\mathcal{E}$$
$$\mathcal{F} ::= \lambda k.h\ \mathcal{E}\ l \mid \lambda k.h\ \mathcal{V}\ \mathcal{F} \mid \lambda k.h\ ?\{\mathbf{p} \to \boldsymbol{\mathcal{V}}; p' \to x\ \mathcal{F}; \mathbf{p''} \to \mathbf{t}\ \}$$
$$\mid \lambda k.h\ ?\{\mathbf{p} \to \boldsymbol{\mathcal{V}}; p' \to []; \mathbf{p''} \to \mathbf{t}\ \}$$

$$\mathcal{V} ::= \lambda k.x\ \mathcal{U} \mid \lambda k.h\ \lambda x.\mathcal{V} \mid \lambda k.h\ (c\ \boldsymbol{\mathcal{V}}) \mid x$$
$$\mathcal{U} ::= \lambda k.h\ \mathcal{V}\ \mathcal{U} \mid \lambda k.h\ ?\ \{p_i \to x_i\ \mathcal{V}_i\}_{i=1}^{n}$$

Fig. 15. CPS contexts, redexes, values

2. *for all* $\mathcal{E} \in \mathsf{Ctxt_c}$, *either* $\mathcal{E} \in \mathsf{DCtxt_n} \cap \mathsf{DCtxt_v}$; *or* $\mathcal{E} \equiv \mathcal{E}'[\lambda k.[]\ l]$, *where* $\mathcal{E}'[\lambda k.[]] \in \mathsf{DCtxt_n} \cap \mathsf{DCtxt_v}$ *and* $[]\ l \in \mathsf{Ctxt_n} \cap \mathsf{Ctxt_v}$.

Proof. First note that $\mathsf{Cont} \subseteq \mathsf{Val_v}$, $\mathsf{Term_c} \subseteq \mathsf{Val_v}$, and $\mathsf{Redex_c} \subseteq \mathsf{Redex_n} \cap \mathsf{Redex_v}$. Then proceed as follows.

1. Prove by induction on t simultaneously that if t is in $\mathsf{Val_c}$ or $\mathsf{ContVal}$ then $t \in \mathsf{DVal_v}$.
2. First suppose $\mathcal{E}$ is not of form $\mathcal{E}'[\lambda k.[]\ l]$. Then by induction on $\mathcal{E}$ prove that $\mathcal{E} \in \mathsf{DCtxt_n} \cap \mathsf{DCtxt_v}$. If $\mathcal{E} \equiv \mathcal{E}'[\lambda k.[]\ l]$, then clearly, using the first case, $\mathcal{E}'[\lambda k.[]] \in \mathsf{DCtxt_v} \cap \mathsf{DCtxt_n}$, and $[]\ l \in \mathsf{Ctxt_n} \cap \mathsf{Ctxt_v}$ by 1.□

Lemma 14. *For all* $t \in \mathsf{Term_c}$: $t \in \mathsf{DVal_n} \cap \mathsf{DVal_v}$; *or there is a unique triple* $(\mathcal{E}, e, r) \in \mathsf{DCtxt_n} \cap \mathsf{DCtxt_v} \times \mathsf{Ctxt_n} \cap \mathsf{Ctxt_v} \times \mathsf{Redex_n} \cap \mathsf{Redex_v}$ *such that* $t \equiv \mathcal{E}[e[r]]$.

Proof. By the preceding two lemmas.□

Step 3. The class of CPS-terms is closed under CBN and CBV unfoldings.

Lemma 15.

1. *For all* $s \in \mathsf{Term}$, *all* $t \in \mathsf{Term_c}$, *and all* $x \in \mathsf{TermVar}$:
   (a) *if* $s \in \mathsf{Term_c}$ *then* $s\{x := t\} \in \mathsf{Term_c}$.
   (b) *if* $s \in \mathsf{Cont}$ *then* $s\{x := t\} \in \mathsf{Cont}$.
2. *For all* $s \in \mathsf{Term}$, *all* $l \in \mathsf{Cont}$, *and all* $k \in \mathsf{ContVar}$:
   (a) *if* $s \in \mathsf{Term_c}$ *then* $s\{k := l\} \in \mathsf{Term_c}$.
   (b) *if* $s \in \mathsf{Cont}$ *then* $s\{k := l\} \in \mathsf{Cont}$.

Proof. In both 1 and 2 prove a and b simultaneously by induction on s.□

Lemma 16. *For all* $t \in \mathsf{Term_c}$: *if* $t \Rightarrow_n t'$ *then* $t' \in \mathsf{Term_c}$.

Proof. By induction on $t \in \mathsf{Term_c}$ using the preceding lemma.□

Theorem 17. *For all* $t \in \mathsf{Term}$: $\underline{t} \Rightarrow_n^* t'$ *iff* $\underline{t} \Rightarrow_v^* t'$.

Proof. By Lemmas 11, 14, and 16.□

8 Folding

In this section we extend our simulation and indifference results to folding. We first add a folding strategy to our first-order CBN unfolding, arriving at first-order CBN transformation, essentially the well-known first-order deforestation algorithm. Then we argue that CBV folding behaves on CBN CPS translated programs similar to CBN folding on DS programs.

8.1 Folding in CBN Transformation

For most terms t with calls to recursively defined functions, the sequence $t \equiv t_0 \Rightarrow_1 t_1 \Rightarrow_1 t_2 \ldots$ will be infinite. In order to get from a term t and a program p defining all the functions called in t to a new equivalent term t' and program p' we introduce folding.

The idea is to introduce new function definitions when applying the relation $\Rightarrow_1$. Instead of letting $\mathcal{E}[e[r]] \Rightarrow_1 \mathcal{E}[e[t]]$ we rewrite $\mathcal{E}[e[r]]$ to $\mathcal{E}[f^* \mathbf{x}]$ ($\mathbf{x}$ is the free variables of $e[r]$) and introduce a new function $f\,\mathbf{x} \stackrel{\triangle}{=} e[t]$. We can then transform further on $e[t]$ using the same idea.

The question arises what to do with the term $\mathcal{E}[f^* \mathbf{x}]$. This term could have more redexes, but the newly introduced call should not be unfolded. The solution is to view $f^* \mathbf{x}$ as a new syntactic construction and add it to the set $\mathsf{DVal_n}$:

$$t ::= \ldots \mid f^* \mathbf{x} \mid \ldots$$

$$\mathcal{V} ::= \ldots \mid f^* \mathbf{x} \mid \ldots$$

It is easy to see that the unique decomposition property still holds.

Whenever transforming a term $\mathcal{E}[e[r]]$ we check whether the term $e[r]$ has previously been encountered. If so we can transform $\mathcal{E}[e[r]]$ to $\mathcal{E}[f^* \mathbf{x}]$ where $f^* \mathbf{x}$ was the call we introduced the first time we saw $e[r]$. Similarly to [Wad90], we consider folding only for "proper redexes", that is terms with $r \not\equiv x\,?\,\{\mathbf{p} \rightarrow \mathbf{t}\}$.

Fig. 16 defines an algorithm for this.

For an example, recall the double append term from the introduction, see Fig. 17. Applying the CBN transformer to this term and program gives the term and program in Fig. 18, in which we have unfolded all calls to functions that are called exactly once (*post-unfolding*). This program is improved since no intermediate list is constructed.

8.2 Simulation and Indifference for Folding

We now extend the simulation/indifference results to the transformers with folding. We prove that whenever the first-order CBN transformer has the possibility to fold, there is a corresponding folding possibility during the transformation of the CBN CPS translated program.

In general, fold steps can occur when, during transformation of the direct style program, a term of the form $\mathcal{E}[e[r]]$ with $r \not\equiv x\,?\,\{\mathbf{p} \rightarrow \mathbf{t}\}$ is encountered. The following two lemmas characterize the corresponding terms encountered during transformation of the CBN CPS translated program.

$A := \{f_0^*\ \mathbf{x}_0 \stackrel{\triangle}{=} t\}$ where $\{\mathbf{x}_0\} = \mathtt{FV}(t)$; $\phi := \emptyset$;
while $\exists f^*\ \mathbf{x} \stackrel{\triangle}{=} t \in A$ with $t \equiv \mathcal{E}[e[r]]$ do
    $A := A \backslash \{f^*\ \mathbf{x} \stackrel{\triangle}{=} t\}$;
    let $\mathcal{E}[e[r]] \Rightarrow_1 \mathcal{E}[e[t']]$ in
        if $\langle e[r]\{\mathbf{z} := \mathbf{y}\}, g^*\ \mathbf{y}\rangle \in \phi$ and $r \not\equiv x\,?\,\{\mathbf{p} \to \mathbf{t}\}$ then
            $A := A \cup \{f^*\ \mathbf{x} \stackrel{\triangle}{=} \mathcal{E}[g^*\ \mathbf{z}]\}$
        otherwise
            $A := A \cup \{f^*\ \mathbf{x} \stackrel{\triangle}{=} \mathcal{E}[g^*\ \mathbf{y}]\} \cup \{g^*\ \mathbf{y} \stackrel{\triangle}{=} e[t]\}$
            $\phi := \phi\ \cup \langle e[r], g^*\ \mathbf{y}\rangle$
            where $\{\mathbf{y}\} = \mathtt{FV}(e[r])$ and g^* is a fresh name
return term $f_0\ \mathbf{x}_0$ and program A

Fig. 16. CBN transformation with folding

$$a\ (a\ us\ vs)\ ws$$
$$a\ xs\ ys \stackrel{\triangle}{=} xs\ ?\{[] \to ys;\ (z : zs) \to z : a\ zs\ ys\}$$

Fig. 17. Double append

Lemma 18.

1. *If* $t \equiv \mathcal{E}[e[r]]$ *and* $u \in t \star \kappa$ *then* $u \equiv \mathcal{E}'[e[r] : \kappa']$ *for some* $\mathcal{E}'$ *and* κ'
2. *If* $t \equiv \mathcal{E}[e[r]]$ *and* $u \in \widetilde{t}$ *then* $u \equiv \mathcal{E}'[e[r] : \kappa']$ *for some* $\mathcal{E}'$ *and* κ'

($\mathcal{E}, e, r$ are in $\mathsf{DCtxt_1}, \mathsf{Ctxt_1}, \mathsf{Redex_1}$ *and* $\mathcal{E}'$ *is in* $\mathsf{DCtxt_n}$*).*

Proof. Induction on $\mathcal{E}$ using part 1 to prove part 2.□

Lemma 19. *for some* $l \in \mathsf{Cont}$*:*

1. $e[c\ \mathbf{t}\,?\,\{p_i \to t_i'\}_{i=1}^n] : \kappa \equiv c\ \underline{\mathbf{t}}\,?\,\{p_i \to \underline{t_i}\ l\}_{i=1}^n$
2. $e[f\ \mathbf{t}] : \kappa \equiv (f\ \mathbf{t})\ l$

Proof. Induction on the structure of e.□

Consider now the general situation in which a fold step is possible:

$$t_1 \equiv \mathcal{E}_1[e_1[r_1]] \Rightarrow_1^* \mathcal{E}_2[e_2[r_2]] \equiv t_2$$

$$f_1^*\ xs\ ys\ zs$$
$$f_1^*\ xs\ ys\ zs \stackrel{\triangle}{=} xs?\ [] \to ys\ ?\{[] \to zs;\ (w : ws) \to w : (f_4^*\ ws\ zs)\}$$
$$(v : vs) \to v : (f_1^*\ vs\ ys\ zs)$$
$$f_4^*\ ws\ zs \stackrel{\triangle}{=} ws\ ?\{[] \to zs;\ (u : us) \to u : (f_4^*\ uszs)\}$$

Fig. 18. Improved double append term

where $e_1[r_1]$ is a renaming of $e_2[r_2]$, and $r_2 \not\equiv x\,?\,\{\mathbf{p} \to \mathbf{t}\}$. By simulation the corresponding situation in CBN CPS is that

$$u_1 \Rightarrow_n^* u_2$$

with $u_1 \in \widetilde{t_1}$ and $u_2 \in \widetilde{t_2}$. By Lemma 18, $u_1 \equiv \mathcal{E}_1'[e_1[r_1] : \kappa_1]$ and $u_2 \equiv \mathcal{E}_2'[e_2[r_2] : \kappa_2]$.

Now by indifference, $\mathcal{E}_1'$ and $\mathcal{E}_2'$ are passive contexts in both CBN and CBV. Furthermore, it is easy to see from Fig. 10 that since $e_1[r_2]$ is a renaming of $e_2[r_2]$, it is also the case that $e_1[r_1] : \kappa_1$ is a renaming of $e_2[r_2] : \kappa_2$. Finally we can use Lemma 19 to see that $e_1[r_1] : \kappa_1$ (and thus $e_2[r_2] : \kappa_2$) is of one of the following two forms:

$$c\ \underline{\mathbf{t}}\,?\,\{p_i \to t_i'\ l\}_{i=1}^n \tag{1}$$

$$(f\ \underline{\mathbf{t}})\ l \tag{2}$$

To summarise, we have seen that whenever the CBN transformer has the possibility to do a fold step, the CBV transformer will meet the terms $\mathcal{E}_1[u_1]$ and $\mathcal{E}_2[u_2]$, where u_1 and u_2 are renamings of each other and both of one of the simple forms (1) and (2) above. Thus, as long as the folding strategy used by the CBV transformer is able to fold in this simple situation, the CBV transformer will be able to fold (at least) as often as the CBN transformer.

A particularly popular folding strategy in the partial evaluation community is to fold only on dynamic conditionals (terms of the form $x\,?\,\{\mathbf{p} \to \mathbf{t}\}$) and dynamic lambda-expressions. The rationale behind this strategy is that any transformation reduction sequence must eventually meet such a term unless it is transforming an infinite loop completely under static control [Bon90a, section 4.5.2]. This strategy will not be able to fold on u_1 and u_2 as above. However, since u_1 is a renaming of u_2, any dynamic conditional or lambda-expression encountered during transformation of u_1 will also be a renaming of a dynamic conditional or lambda-expression encountered during transformation of u_2. So with this strategy, folding just occurs a bit later; except for the exact placement of residual calls, the result is the same.

8.3 Example

As an example of folding in CBN CPS consider the CPS-translated version of the double append term and the append function in Fig. 19.

$$\lambda k.a\ (\lambda h.a\ xs\ ys\ h)\ zs\ k$$
$$a\ xs\ ys \triangleq \lambda k.xs\ \lambda v.v\ ?\{[] \to ys\ k;\ (z : zs) \to (\lambda h.h\ z : \lambda l.a\ zs\ ys\ l)\ k\}$$

Fig. 19. CBN CPS version of double append

The result of CBN transforming this term and program is seen in Fig. 20 (after post-unfolding). The figure also shows the corresponding DS program

(the CPS program arises from the DS program by CPS translation followed by reduction of administrative redexes). The DS program is essentially the same as the improved double append term and program in Fig. 18.

$$
\begin{array}{lll}
 & & \lambda k.f^*\ xs\ ys\ zs \\
f^*\ xs\ ys\ zs\ k & \triangleq & xs\ \lambda ww?\ [] \qquad\quad \to ys\ \lambda vv?\ [] \qquad\quad \to zs\ k \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (u : us) \to g^*\ u\ us\ zs\ k \\
 & & \qquad\qquad\quad (b : bs) \to k\ (b : \lambda l.f^*\ bs\ ys\ zs\ l) \\
g^*\ u\ us\ zs\ k & \triangleq & k\ (u : \lambda l.us\ \lambda v.v\ ?\{[] \to zs\ l;\ (b : bs) \to g^*\ b\ bs\ zs\ l\}) \\
 & & \\
 & & f^*\ us\ vs\ ws \\
f^*\ us\ vs\ ws & \triangleq & us?\ [] \qquad\quad \to vs\ ?\{[] \to ws;\ (z : zs) \to g^*\ z\ zs\ ws\} \\
 & & \qquad\quad (y : ys) \to y : f^*\ ys\ vs\ ws \\
g^*\ z\ zs\ ws & \triangleq & z : (zs\ ?\{[] \to ws;\ (y : ys) \to y : g^*\ ys\ ws\})
\end{array}
$$

Fig. 20. Improved double append in CBN CPS and DS

9 Related Work

Our paper is a step in the current efforts at DIKU to better understand the relationship between different transformation techniques [Sor94, Glu94, Glu94a, Jon94]. Such efforts facilitate exchange of ideas between different techniques.

Our results are related to previous work on partial evaluation and the CBV CPS translation. In [Con91, Dan91] it was explained how CBV CPS can improve the specialization of programs by allowing static information to propagate across dynamic contexts. Later work [Bon92, Bon94, Law94] describes an extension of "standard" partial evaluation called *CPS-specialization*; CPS-specialization of a program achieves the same effect as plain partial evaluation of the CBV CPS translated program. This paper explains the analogous situation for the CBN CPS translation; CBN CPS is to deforestation as CBV CPS is to CPS-specialization.

Another approach is due to Glück and Jørgensen [Glu94]. They start out with a CBN interpreter with context distribution rules and show that specializing the interpreter to some programs yield the same results as applying deforestation directly to the programs.

One of the points of this paper is that the CBN CPS translation can replace the use of certain special reductions for dynamic conditionals as performed in deforestation. Sabry and Felleisen explain generally for the CBV CPS how CPS translation corresponds to certain extra reductions in direct style [Sab93].

10 Conclusion and Future Work

We have presented the CBN CPS translation as a tool for improving partial evaluation beyond what is possible using previous techniques for binding time improvement, including CBV CPS. Furthermore, we have argued that in general, the effect of CBN CPS with partial evaluation is that of Wadler's deforestation algorithm. Thus, our results should be of practical interest for the application of partial evaluation to "real" problems, as well as of theoretical interest for relating partial evaluation and deforestation.

An important application of these results is the development of stronger program transformers that combine the effects of plain partial evaluation, CBV CPS translation (ie. CPS-specialization), and CBN CPS translation (ie. deforestation). The main difficulty is to automate the decision about when it is safe to transform using CBN CPS, when to use CBV CPS, and when it is necessary to use plain partial evaluation to avoid infinite specialization and code explosion.

A related problem occurs in [Dan92] where the result of a strictness analysis is used to determine when one can CBV CPS translate programs with CBN semantics.

Acknowledgements. For comments and discussions we are indebted to O. Danvy, D. Dussart, R. Glück, J. Hatcliff, N.D. Jones, J. Jørgensen, T. Mogensen, D. Sands, V.F. Turchin and W.-N. Chin.

References

[Bon90] A. Bondorf, O. Danvy. *Automatic Autoprojection of Higher-Order Recursive Equations.* ESOP '90. LNCS vol. 432 pp70-87, 1990

[Bon90a] A. Bondorf. *Self-Applicable Partial Evaluation.* Ph.D. thesis, DIKU-Rapport 90/17, Department of Computer Science, University of Copenhagen, 1990.

[Bon92] A. Bondorf. Improving Binding Times without Explicit CPS-Conversion. In *ACM Lisp and Functional Programming Conference.* San Francisco, California, June 1992.

[Bon94] A. Bondorf, D. Dussart. Improving CPS-Based Partial Evaluation: Writing Cogen by Hand. In *PEPM '94.* Orlando, Florida, 1994.

[Con91] C. Consel, O. Danvy. For a Better Support of Static Data Flow. In *FPCA '91.* (Ed.) John Hughes, LNCS Vol.523, pp.495-519, 1991.

[Dan91] O. Danvy. Semantics-Directed Compilation of Non-Linear Patterns. In *Information Processing Letters.* Vol.37, pp.315-322, March 1991.

[Dan92] O. Danvy, J. Hatcliff. CPS-Transformation After Strictness Analysis. In *ACM Letters on Programming Languages and Systems.* Vol.1, No.3, September 1992.

[Dus95] D. Dussart. Proving Program Transformers Correct by Program Transformation. In *Workshop on Logic, Domains, and Programming Languages, Darmstadt Germany (May 24-27).* 1995.

[Glu94] R. Glück, J. Jørgensen. Generating Transformers for Deforestation and Driving. In *Static Analysis Symposium, Namur, Belgium.* LNCS vol 864, 1994.

[Glu94a] R. Glück, M. H. Sørensen. Partial Deduction and Driving are Equivalent. In *PLILP '94, Madrid, Spain.* LNCS vol 844, 1994.

[Ham94] G. W. Hamilton. *Higher Order Deforestation.* Unpublished manuscript. 1994.

[Hat95] J. Hatcliff, O. Danvy. *Thunks and the λ-calculus.* DIKU-Rapport 95/3, Department of Computer Science, University of Copenhagen, 1995.

[Jon93] N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jon94] N. D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. In *N. D. Jones and M. Hagiya and M. Sato (eds.), Logic, Language and Computation.* LNCS Vol. 792, 206-224, Springer-Verlag 1994.

[Law94] J. L. Lawall, O. Danvy. Continuation-Based Partial Evaluation. In *ACM SIGPLAN Symposium on LISP and Functional Programming, June 27–29.* Orlando, Florida, 1994.

[Mar92] S. Marlow, P. L. Wadler. Deforestation for Higher-Order functions. In *Functional Programming, Glasgow 1992.* Ed. J. Launchbury, Workshops in Computing, 1992.

[Plo75] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. In *Theoretical Computer Science.* 1, 1975.

[Sab93] A. Sabry, M. Felleisen. Reasoning about Programs in Continuation-Passing Style. In *Lisp and Symbolic Computation.* Kluwer Academic Publishers, 1993.

[San95] D. Sands. Total Correctness and Improvement in the Transformation of Recursive Functions. In *TAPSOFT '95.* To appear in LNCS, 1995.

[Sor94] M. H. Sørensen, R. Glück, N. D. Jones. Towards Unifying Deforestation, Supercompilation, Partial Evaluation, and Generalized Partial Computation. In *ESOP '94.* LNCS vol. 788, 1994.

[Tur86] V. F. Turchin. The Concept of a Supercompiler. In *ACM Transactions on Programming Languages and Systems.* Vol. 8, No. 3, pp. 292- 325, 1986.

[Wad90] P. L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *Theoretical Computer Science.* 73, pp231-248, 1990.

[Wei91] D. Weise, R. Conybeare, E. Ruf, S. Seligman. Automatic Online Partial Evaluation. In *FPCA '91.* (Ed.) John Hughes, LNCS Vol.523, pp.495-519, 1991.

This article was processed using the LaTeX macro package with LLNCS style