LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993

Polymorphic Type Assignment and CPS Conversion*

ROBERT HARPER† (*rwh@cs.cmu.edu*)

MARK LILLIBRIDGE‡ (*mdl@cs.cmu.edu*)

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Keywords: Polymorphism, continuations

Abstract. Meyer and Wand established that the type of a term in the simply typed λ-calculus may be related in a straightforward manner to the type of its call-by-value CPS transform. This typing property may be extended to Scheme-like continuation-passing primitives, from which the soundness of these extensions follows. We study the extension of these results to the Damas-Milner polymorphic type assignment system under both the call-by-value and call-by-name interpretations. We obtain CPS transforms for the call-by-value interpretation, provided that the polymorphic let is restricted to values, and for the call-by-name interpretation with no restrictions. We prove that there is no call-by-value CPS transform for the full Damas-Milner language that validates the Meyer-Wand typing property and is equivalent to the standard call-by-value transform up to operational equivalence.

1. Introduction

In their study of the relationship between direct and continuation semantics for the simply typed λ-calculus ($\lambda^{\rightarrow}$), Meyer and Wand note that the type of a term in $\lambda^{\rightarrow}$ may be related in a simple and natural way to the type of its call-by-value continuation passing style (CPS) transform [13]. This result may be extended to the calculus that results from extending $\lambda^{\rightarrow}$ with Scheme-like continuation-passing primitives callcc and throw ($\lambda^{\rightarrow} + \mathsf{cont}$) [1, 4]. Since $\lambda^{\rightarrow}$ under a call-by-value operational semantics is "type safe" in the sense of Milner [14, 2], and since the call-by-value CPS transform faithfully mimics the call-by-value semantics [17], it follows that

*This is a revised version of a paper presented at the ACM SIGPLAN Workshop on Continuations, San Francisco, June 1992.

†This work was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168.

‡Supported by a National Science Foundation Graduate Fellowship.

$\lambda^{\rightarrow} + \mathsf{cont}$ under a call-by-value operational semantics is also type safe.

In a subsequent study Duba, Harper, and MacQueen studied the addition of callcc and throw to Standard ML [15]. The extension of the Meyer-Wand transform to $\lambda^{\rightarrow} + \mathsf{cont}$ establishes the soundness of the monomorphic fragment of the language, but the soundness of the polymorphic language with continuation-passing primitives was left open. It was subsequently proved by the authors [9] that the full polymorphic language is unsound when extended with callcc and throw. The source of this discrepancy may be traced to the interaction between the polymorphic let construct and the typing rules for callcc. Several *ad-hoc* methods for restricting the language to recover soundness have been proposed [8, 23].

In this paper we undertake a systematic study of the interaction between continuations and polymorphism by considering the typing properties of the CPS transform for both the call-by-value and call-by-name variants of the Damas-Milner language [2] and its extension with continuation-passing primitives. We obtain suitable extensions of the Meyer-Wand theorem for the call-by-value CPS transform, provided that the polymorphic let is restricted to values, and for the call-by-name transform, under no restrictions. Finally, we prove that there is no call-by-value CPS transform for the full Damas-Milner language that both satisfies the Meyer-Wand typing property and is equivalent to the usual transform up to operational equivalence. In particular, the standard call-by-value CPS transform fails to preserve typability.

2. Untyped Terms

The language of untyped terms is given by the following grammar:

$$e \quad ::= \quad x \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{callcc} \mid \mathsf{throw}$$

Here x ranges over a countably infinite set of variables. We include the let construct as a primitive because it is needed in the discussion of polymorphic type assignment. The constants callcc and throw stand for continuation-passing primitives whose definitions are derived from analogous constructs in Scheme [1] and Standard ML of New Jersey [4].

We consider two CPS transforms for untyped terms, corresponding to the call-by-value and call-by-name operational semantics [17]. Each CPS transform consists of a transformation $|-|$ for untyped terms and a transformation $||-||$ for untyped values. Exactly what is considered a value depends on which operational semantics is being used. Under a call-by-value interpretation, the set of values is given by the following grammar:

$$v \quad ::= \quad x \mid \lambda x.e \mid \mathsf{callcc} \mid \mathsf{throw}$$

Since throw is a two-argument function, it is possible to regard $\mathsf{throw}\, v$ as a value. We choose to not regard it as a value because to do so complicates somewhat both the transform and the associated theorems due to the resulting overlap between the application and value cases. Under a call-by-name interpretation, the set of values is given by the following grammar:

$$n \quad ::= \quad \lambda x.e \mid \mathsf{callcc} \mid \mathsf{throw}$$

Definition 1 (Call-by-Value CPS Transform) [1]

$$\begin{aligned} |v|_{cbv} &= \lambda k.k\,||v||_{cbv} \\ |e_1\,e_2|_{cbv} &= \lambda k.|e_1|_{cbv}\,(\lambda x_1.|e_2|_{cbv}\,(\lambda x_2.x_1\,x_2\,k)) \\ |\mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2|_{cbv} &= \lambda k.|e_1|_{cbv}\,(\lambda x.|e_2|_{cbv}\,k) \\[1em] ||x||_{cbv} &= x \\ ||\lambda x.e||_{cbv} &= \lambda x.|e|_{cbv} \\ ||\mathsf{callcc}||_{cbv} &= \lambda f.\lambda k.f\,k\,k \\ ||\mathsf{throw}||_{cbv} &= \lambda c.\lambda k.k\,(\lambda x.\lambda l.c\,x) \end{aligned}$$

The transformation of let expressions is chosen to reflect the usual "sequential" evaluation strategy for let's in a call-by-value language in which the let-bound expression, e_1, is fully evaluated prior to evaluation of the body, e_2. (Compare this with the call-by-name transform in Definition 2 below.) The transformation of callcc differs from that in Scheme since continuations are not here represented as functions which are applied to their arguments, but rather are represented directly as continuations which are invoked using the throw primitive. (See Harper, Duba, and MacQueen [8] for further discussion of this point.)

The call-by-value CPS transform is compositional in the sense that it commutes with substitution.

Lemma 1

1. $||\,[v/x]v'\,||_{cbv} = [\,||v||_{cbv}/x\,]\,||v'||_{cbv}$.

2. $|\,[v/x]e\,|_{cbv} = [\,||v||_{cbv}/x\,]\,|e|_{cbv}$.

> **Proof:** The proof proceeds by simultaneous induction on the structure of v' and e. We give here a few illustrative cases; the remainder follow by a similar argument. We drop the "cbv" subscript for the sake of readability.

---

[1] In each equation any bound variable occurring on the right that does not also occur on the left is assumed to be chosen so as to avoid capture.

$v' = x$:
We have by definition $||x|| = x$, and hence $[||v'||/x]||x|| = ||v'|| = ||[v'/x]x||$.

$v' = x' \neq x$:
We have by definition $||x'|| = x'$, and hence $[||v'||/x]||x'|| = x' = ||[v'/x]x'||$.

$v' = \lambda y.e$:
We may assume without loss of generality that $y \neq x$ and that y does not occur freely in $||v||$.

$$\begin{array}{rcll} ||[v/x]\lambda y.e|| & = & ||\lambda y.[v/x]e|| & \\ & = & \lambda y.|[v/x]e| & \\ & = & \lambda y.[||v||/x]|e| & \textit{(by induction)} \\ & = & [||v||/x]\lambda y.|e| & \\ & = & [||v||/x]\;||\lambda y.e|| & \end{array}$$

$e = v'$:
We may assume without loss of generality that k does not occur freely in $||v'||$. Note that values are stable under substitution of a value for a variable.

$$\begin{array}{rcll} |[v/x]v'| & = & \lambda k.k\,||[v/x]v'|| & ([v/x]v' \textit{ is a value}) \\ & = & \lambda k.k\,[||v||/x]||v'|| & \textit{(by induction)} \\ & = & [||v||/x]\lambda k.k\,||v'|| & \\ & = & [||v||/x]\;||v'|| & \end{array}$$

$e = e_1\,e_2$:
We may assume without loss of generality that the variables x_1, x_2, and k do not occur freely in $||v||$.

$$\begin{array}{rcl} |[v/x](e_1\,e_2)| & = & |[v/x]e_1\,[v/x]e_2| \\ & = & \lambda k.|[v/x]e_1|\,(\lambda x_1.|[v/x]e_2|\,(\lambda x_2.x_1 x_2 k)) \\ & = & \lambda k.[||v||/x]|e_1|\,(\lambda x_1.[||v||/x]|e_2|\,(\lambda x_2.x_1 x_2 k)) \\ & = & [||v||/x]\lambda k.|e_1|\,(\lambda x_1.|e_2|\,(\lambda x_2.x_1 x_2 k)) \\ & = & [||v||/x]|e_1\,e_2| \end{array}$$

∎

We shall also have need of a variant call-by-value CPS transform (cbv') defined on untyped terms satisfying the restriction that all **let** expressions are of the form **let** x **be** v **in** e — *i.e.*, the **let**-bound expression is required to be a (call-by-value) value. Because of this restriction, a simpler rule can be given for the **let** case:

$$|\mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e|_{cbv'} \quad = \quad \lambda k.\mathsf{let}\ x\ \mathsf{be}\ ||v||_{cbv'}\ \mathsf{in}\ (|e|_{cbv'}\,k)$$

This simpler rule for **let** expressions is the only difference between the two transforms.

Lemma 2 *Let v and v' be values obeying the restriction on* let *expressions and e be a term obeying the restriction on* let *expressions. Then*

1. $||\,[v/x]v'\,||_{cbv'} = [\,||v||_{cbv'}/x\,]\,||v'||_{cbv'}$.
2. $|\,[v/x]e\,|_{cbv'} = [\,||v||_{cbv'}/x\,]\,|e|_{cbv'}$.

Definition 2 (Call-by-Name CPS Transform) [2]

$$\begin{array}{rcl} |n|_{cbn} & = & \lambda k.k\,||n||_{cbn} \\ |x|_{cbn} & = & x \\ |e_1\,e_2|_{cbn} & = & \lambda k.|e_1|_{cbn}\,(\lambda x_1.x_1\,|e_2|_{cbn}\,k) \\ |\mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2|_{cbn} & = & \lambda k.\mathsf{let}\ x\ \mathsf{be}\ |e_1|_{cbn}\ \mathsf{in}\ (|e_2|_{cbn}\,k) \\ \\ ||\lambda x.e||_{cbn} & = & \lambda x.|e|_{cbn} \\ ||\mathsf{callcc}||_{cbn} & = & \lambda f.\lambda k.f\,(\lambda f'.f'\,(\lambda l.l\,k)\,k) \\ ||\mathsf{throw}||_{cbn} & = & \lambda c.\lambda k.k\,(\lambda x.\lambda l.c\,(\lambda c'.x\,(\lambda x'.c'\,x'))) \end{array}$$

The reader may find it helpful to bear in mind that in call-by-name variables are bound to suspensions, rather than to values. The rather complicated definitions of $||\mathsf{callcc}||_{cbn}$ and $||\mathsf{throw}||_{cbn}$ may be attributed to this fact.

Lemma 3

1. $||\,[e/x]n\,||_{cbn} = [\,|e|_{cbn}/x\,]\,||n||_{cbn}$.
2. $|\,[e/x]e'\,|_{cbn} = [\,|e|_{cbn}/x\,]\,|e'|_{cbn}$.

Proof: By simultaneous induction on the structure of n and e'. ∎

The relationship between the call-by-value and call-by-name CPS transforms and the corresponding call-by-value and call-by-name operational semantics was established for pure λ-terms by Plotkin [17] and was extended to continuation-passing primitives (for the call-by-value case) by Griffin [6]. (See also [5, 19, 20, 21, 3].)

Theorem 1 (Plotkin, Griffin) *The closed expression e evaluates to v under call-by-value iff $|e|_{cbv}\,(\lambda x.x)$ evaluates to $||v||_{cbv}$ under either call-by-value or call-by-name.*

In particular, if e evaluates to a constant c, then $|e|_{cbv}\,(\lambda x.x)$ evaluates to c, and vice-versa, assuming that $||c||_{cbv} = c$. A similar result for the call-by-name interpretation may also be proved [17].

[2] In each equation any bound variable occurring on the right that does not also occur on the left is assumed to be chosen so as to avoid capture.

3. Simple Type Assignment

In this section we review Meyer and Wand's typing theorem for the call-by-value CPS transform for the simply-typed λ-calculus ($\lambda^{\rightarrow}$), and present an analogous result for the call-by-name CPS transform.

Definition 3 ($\lambda^{\rightarrow}$ Types and Contexts)

$$\begin{array}{lccl} \textit{types} & \tau & ::= & b \mid \tau_1 \rightarrow \tau_2 \\ \textit{contexts} & \Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \end{array}$$

Here b ranges over a countable set of base types. We assume that among the base types there is a distinguished type α, which will be used in what follows to represent the "answer" type of a CPS transform.

Definition 4 ($\lambda^{\rightarrow}$ Typing Rules)

$$\Gamma \triangleright x : \Gamma(x) \qquad \text{(VAR)}$$

$$\frac{\Gamma, x{:}\tau_1 \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \rightarrow \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(ABS)}$$

$$\frac{\Gamma \triangleright e_1 : \tau_2 \rightarrow \tau \quad \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright e_1\, e_2 : \tau} \qquad \text{(APP)}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \triangleright e_2 : \tau}{\Gamma \triangleright \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(MONO-LET)}$$

The type system $\lambda^{\rightarrow} + \mathsf{cont}$ is defined by adding the type expression $\tau\,\mathsf{cont}$ and the following typing rules for the continuation-passing primitives:

$$\Gamma \triangleright \mathsf{callcc} : (\tau\,\mathsf{cont} \rightarrow \tau) \rightarrow \tau \qquad \text{(CALLCC)}$$

$$\Gamma \triangleright \mathsf{throw} : \tau\,\mathsf{cont} \rightarrow \tau \rightarrow \tau' \qquad \text{(THROW)}$$

Definition 5 (Call-by-Value Type Transform for $\lambda^{\rightarrow}$)

$$|\tau|_{cbv} = (||\tau||_{cbv} \rightarrow \alpha) \rightarrow \alpha$$

$$\begin{array}{rcl} ||b||_{cbv} & = & b \\ ||\tau_1 \rightarrow \tau_2||_{cbv} & = & ||\tau_1||_{cbv} \rightarrow |\tau_2|_{cbv} \end{array}$$

The type transform is extended to contexts by defining $||\Gamma||_{cbv}(x) = ||\Gamma(x)||_{cbv}$ for each $x \in \mathrm{dom}(\Gamma)$.

Theorem 2 (Meyer-Wand)

1. *If* $\lambda^{\rightarrow} \vdash \Gamma \rhd v : \tau$, *then* $\lambda^{\rightarrow} \vdash ||\Gamma||_{cbv} \rhd ||v||_{cbv} : ||\tau||_{cbv}$.
2. *If* $\lambda^{\rightarrow} \vdash \Gamma \rhd e : \tau$, *then* $\lambda^{\rightarrow} \vdash ||\Gamma||_{cbv} \rhd |e|_{cbv} : |\tau|_{cbv}$.

The call-by-value type transform is extended to $\lambda^{\rightarrow} + \mathsf{cont}$ by defining $||\tau\,\mathsf{cont}||_{cbv} = ||\tau||_{cbv} \rightarrow \alpha$. It is straightforward to verify that Theorem 2 extends to $\lambda^{\rightarrow} + \mathsf{cont}$ in this way [4].

Definition 6 (Call-by-Name Type Transform for $\lambda^{\rightarrow}$) [3]

$$|\tau|_{cbn} = (||\tau||_{cbn} \rightarrow \alpha) \rightarrow \alpha$$

$$\begin{aligned} ||b||_{cbn} &= b \\ ||\tau_1 \rightarrow \tau_2||_{cbn} &= |\tau_1|_{cbn} \rightarrow |\tau_2|_{cbn} \end{aligned}$$

The type transform is extended to contexts by defining $|\Gamma|_{cbn}(x) = |\Gamma(x)|_{cbn}$ for each $x \in \mathrm{dom}(\Gamma)$.

Theorem 3

1. *If* $\lambda^{\rightarrow} \vdash \Gamma \rhd n : \tau$, *then* $\lambda^{\rightarrow} \vdash |\Gamma|_{cbn} \rhd ||n||_{cbn} : ||\tau||_{cbn}$.
2. *If* $\lambda^{\rightarrow} \vdash \Gamma \rhd e : \tau$, *then* $\lambda^{\rightarrow} \vdash |\Gamma|_{cbn} \rhd |e|_{cbn} : |\tau|_{cbn}$.

The call-by-name CPS transform is extended to $\lambda^{\rightarrow} + \mathsf{cont}$ by defining $||\tau\,\mathsf{cont}||_{cbn} = ||\tau||_{cbn} \rightarrow \alpha$, just as for call-by-value. It is straightforward to verify that Theorem 3 extends to $\lambda^{\rightarrow} + \mathsf{cont}$ in this way.

4. Polymorphic Type Assignment

In this section we study the extension of the Meyer-Wand typing property to Damas and Milner's polymorphic type assignment system (DM).

The syntax of types and contexts in (DM) is defined by the following grammar:

[3] The term "call-by-name type transform" is something of a misnomer since there exists a by-value CPS transform that validates the by-name typing property [20, 7]. Nevertheless we stick with the suggestive, if somewhat misleading, terminology.

Definition 7 (DM Types and Contexts)

$$\begin{array}{llll} \textit{monotypes} & \tau & ::= & t \mid b \mid \tau_1 \to \tau_2 \\ \textit{polytypes} & \sigma & ::= & \tau \mid \forall t.\sigma \\ \\ \textit{contexts} & \Gamma & ::= & \bullet \mid \Gamma, x{:}\sigma \end{array}$$

Here t ranges over a countably infinite set of type variables. The typing rules of the Damas-Milner system extend those of $\lambda^{\to}$ as follows:

Definition 8 (Additional DM Typing Rules)

$$\frac{\Gamma \triangleright e : \sigma}{\Gamma \triangleright e : \forall t.\sigma} \quad (t \notin FTV(\Gamma)) \qquad (\text{GEN})$$

$$\frac{\Gamma \triangleright e : \forall t.\sigma}{\Gamma \triangleright e : [\tau/t]\sigma} \qquad (\text{INST})$$

$$\frac{\Gamma \triangleright e_1 : \sigma_1 \quad \Gamma, x{:}\sigma_1 \triangleright e_2 : \tau_2}{\Gamma \triangleright \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\text{POLY-LET})$$

The system $\mathsf{DM} + \mathsf{cont}$ is defined by adding the type expression $\tau\,\mathsf{cont}$, as before, and adding the following typing rules:

$$\Gamma \triangleright \mathsf{callcc} : \sigma_{\mathsf{callcc}} \qquad (\text{CALLCC'})$$

$$\Gamma \triangleright \mathsf{throw} : \sigma_{\mathsf{throw}} \qquad (\text{THROW'})$$

where $\sigma_{\mathsf{callcc}} = \forall t.(t\,\mathsf{cont} \to t) \to t$ and $\sigma_{\mathsf{throw}} = \forall s.\forall t.s\,\mathsf{cont} \to s \to t$.

4.1. Restricted Call-by-Value

Let DM^- denote the sub-system of DM obtained by restricting let expressions so that the bound expression is a call-by-value value. The Meyer-Wand typing theorem may be extended to terms of DM^-, provided that we use the variant call-by-value CPS transform (cbv') given in Section 2.

Definition 9 (Call-by-Value Type Transform for DM^-)

$$\begin{array}{rcl} |\tau|_{cbv} & = & (||\tau||_{cbv} \to \alpha) \to \alpha \\ |\forall t.\sigma|_{cbv} & = & \forall t.|\sigma|_{cbv} \\ \\ ||t||_{cbv} & = & t \\ ||b||_{cbv} & = & b \\ ||\tau_1 \to \tau_2||_{cbv} & = & ||\tau_1||_{cbv} \to |\tau_2|_{cbv} \\ ||\forall t.\sigma||_{cbv} & = & \forall t.||\sigma||_{cbv} \end{array}$$

This definition extends the Meyer-Wand type transform to polymorphic types. In the terminology of Reynolds [19], polymorphic instantiation is given a "trivial" interpretation in that no interesting computation can occur as a result of polymorphic instantiation. The definition of $|\forall t.\sigma|_{cbv}$ reflects the fact that in DM^- there is no need of continuations whose domain is a polymorphic type.

Lemma 4

1. $||[\tau/t]\sigma||_{cbv} = [\,||\tau||_{cbv}\,/\,t\,]\,||\sigma||_{cbv}$.
2. $|[\tau/t]\sigma|_{cbv} = [\,||\tau||_{cbv}\,/\,t\,]\,|\sigma|_{cbv}$.

Proof: By induction on the structure of σ. ∎

With this in mind we may establish type preservation for the variant call-by-value CPS transform on the sublanguage DM^-.

Theorem 4

1. *If* $\mathsf{DM}^- \vdash \Gamma \triangleright v : \sigma$, *then* $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright ||v||_{cbv'} : ||\sigma||_{cbv}$.
2. *If* $\mathsf{DM}^- \vdash \Gamma \triangleright e : \sigma$, *then* $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : |\sigma|_{cbv}$.

Proof: Both parts are proved simultaneously by induction on the structure of typing derivations. We give a few illustrative cases; the rest follow a similar pattern.

VAR Suppose that $\mathsf{DM}^- \vdash \Gamma \triangleright x : \sigma$ by an application of rule VAR. But then $\Gamma(x) = \sigma$, and thus $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright ||x||_{cbv'} : ||\sigma||_{cbv}$ since $||x||_{cbv'} = x$.

ABS Suppose that $\mathsf{DM}^- \vdash \Gamma \triangleright \lambda x.e : \tau_1 \rightarrow \tau_2$ by an application of rule ABS. Then $\mathsf{DM}^- \vdash \Gamma, x{:}\tau_1 \triangleright e : \tau_2$, where $x \notin \mathrm{dom}(\Gamma)$, by a smaller typing derivation. Therefore we have, by induction, $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : |\tau_2|_{cbv}$, and consequently that $||\Gamma||_{cbv} \triangleright \lambda x.|e|_{cbv'} : ||\tau_1||_{cbv} \rightarrow |\tau_2|_{cbv}$. The result follows from the definitions of the term and type transforms.

LET Suppose that $\mathsf{DM}^- \vdash \Gamma \triangleright \mathsf{let}\, x\, \mathsf{be}\, v_1\, \mathsf{in}\, e_2 : \tau_2$ by an application of rule POLY-LET. Then $\mathsf{DM}^- \vdash \Gamma \triangleright v_1 : \sigma_1$ and $\mathsf{DM}^- \vdash \Gamma, x{:}\sigma_1 \triangleright e_2 : \tau_2$, where $x \notin \mathrm{dom}(\Gamma)$, by smaller typing derivations. (Recall that the restriction on DM^- ensures that the let-bound expression must be a value.) By induction it follows that $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright ||v_1||_{cbv'} : ||\sigma_1||_{cbv}$ and that $\mathsf{DM}^- \vdash ||\Gamma||_{cbv}, x{:}||\sigma_1||_{cbv} \triangleright |e_2|_{cbv'} : |\tau_2|_{cbv}$. It is easy to check that $||v_1||_{cbv'}$ is always a value, and hence it follows that $\mathsf{DM}^- \vdash \lambda k.\mathsf{let}\, x\, \mathsf{be}\, ||v_1||_{cbv'}\, \mathsf{in}\, |e_1|_{cbv'}\, k : |\tau_2|_{cbv}$, from which the result follows by the definition of the alternate term transform.

GEN Suppose that $\mathsf{DM}^- \vdash \Gamma \triangleright e : \forall t.\sigma$ by an application of rule GEN. Then $\mathsf{DM}^- \vdash \Gamma \triangleright e : \sigma$ by a smaller derivation, where $t \notin FTV(\Gamma)$. By induction $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : |\sigma|_{cbv}$, from which it follows that $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : \forall t.|\sigma|_{cbv}$ by an application of rule GEN, from which the result follows by observing that $|\forall t.\sigma|_{cbv} = \forall t.|\sigma|_{cbv}$.

INST Suppose that $\mathsf{DM}^- \vdash \Gamma \triangleright e : [\tau/t]\sigma$ by an application of rule INST. Then $\mathsf{DM}^- \vdash \Gamma \triangleright e : \forall t.\sigma$ by a smaller derivation, and thus, by induction, $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : |\forall t.\sigma|_{cbv}$. Consequently, $\mathsf{DM}^- \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv'} : [||\tau||/t]|\sigma|$, which is sufficient for the result, by an application of Lemma 4.

∎

Theorem 4 extends to $\mathsf{DM}^- + \mathsf{cont}$ by defining $||\tau\ \mathsf{cont}||_{cbv} = ||\tau||_{cbv} \rightarrow \alpha$. We need only verify that $||\mathsf{callcc}||_{cbv'}$ and $||\mathsf{throw}||_{cbv'}$ given in Section 2, have the types $||\sigma_{\mathsf{callcc}}||_{cbv}$ and $||\sigma_{\mathsf{throw}}||_{cbv}$, respectively. The soundness of $\mathsf{DM}^- + \mathsf{cont}$ under call-by-value follows by the same reasoning as for $\lambda^{\rightarrow} + \mathsf{cont}$.

4.2. Call-by-Name

Theorem 3 (the Meyer-Wand-like typing theorem for call-by-name) can be extended to the unrestricted DM language.

Definition 10 (Call-by-Name Type Transform for DM)

$$\begin{array}{rcl} |\tau|_{cbn} & = & (||\tau||_{cbn} \rightarrow \alpha) \rightarrow \alpha \\ |\forall t.\sigma|_{cbn} & = & \forall t.|\sigma|_{cbn} \\ \\ ||t||_{cbn} & = & t \\ ||b||_{cbn} & = & b \\ ||\tau_1 \rightarrow \tau_2||_{cbn} & = & |\tau_1|_{cbn} \rightarrow |\tau_2|_{cbn} \\ ||\forall t.\sigma||_{cbn} & = & \forall t.||\sigma||_{cbn} \end{array}$$

Lemma 5

1. $||\,[\tau/t]\sigma\,||_{cbn} = [\,||\tau||_{cbn}\,/\,t\,]\,||\sigma||_{cbn}$.

2. $|\,[\tau/t]\sigma\,|_{cbn} = [\,||\tau||_{cbn}\,/\,t\,]\,|\sigma|_{cbn}$.

Proof: By induction on the structure of σ. ∎

Theorem 5

1. *If* $\mathsf{DM} \vdash \Gamma \rhd n : \sigma$*, then* $\mathsf{DM} \vdash |\Gamma|_{cbn} \rhd ||n||_{cbn} : ||\sigma||_{cbn}$.
2. *If* $\mathsf{DM} \vdash \Gamma \rhd e : \sigma$*, then* $\mathsf{DM} \vdash |\Gamma|_{cbn} \rhd |e|_{cbn} : |\sigma|_{cbn}$.

Proof: Similar to that of Theorem 4. ∎

Theorem 5 extends to $\mathsf{DM} + \mathsf{cont}$ by defining $||\tau\,\mathsf{cont}||_{cbn} = ||\tau||_{cbn} \to \alpha$. We need only verify that $||\mathsf{callcc}||_{cbn}$ and $||\mathsf{throw}||_{cbn}$, given in Section 2, have types $||\sigma_{\mathsf{callcc}}||_{cbn}$ and $||\sigma_{\mathsf{throw}}||_{cbn}$, respectively. The soundness of $\mathsf{DM} + \mathsf{cont}$ under call-by-name operational semantics follows from the extended theorem in a manner similar to that of the call-by-value case for $\mathsf{DM}^- + \mathsf{cont}$.

4.3. Unrestricted Call-by-Value

Having established suitable typing properties for the variant call-by-value transform for DM^- and the call-by-name transform for full DM, it is natural to consider whether there is a call-by-value CPS transform for full DM that satisfies a Meyer-Wand-like typing property. The problem is to extend the CPS transform to general let expressions, *i.e.* those not necessarily satisfying the restriction that let-bound expressions be values.

Let us consider a naïve attempt to extend Theorem 4 to full DM by considering the *cbv* term transform instead of the specialized *cbv′* transform. Consider the polymorphic let case. By induction we have $\mathsf{DM} \vdash ||\Gamma||_{cbv} \rhd |e_1|_{cbv} : |\sigma_1|_{cbv}$ and $\mathsf{DM} \vdash ||\Gamma||_{cbv}, x{:}||\sigma_1||_{cbv} \rhd |e_2|_{cbv} : |\tau_2|_{cbv}$, and we are to show that $\mathsf{DM} \vdash ||\Gamma||_{cbv} \rhd \lambda k.|e_1|_{cbv}\,(\lambda x.|e_2|_{cbv}\,k) : |\tau_2|_{cbv}$. Since e_1 is a general computation, a continuation to receive its value (should it have a value) is required. But since e_1 may be polymorphic, the definition of $|\sigma|_{cbv}$ given earlier is no longer appropriate — we require that $|\sigma|_{cbv} = (||\sigma||_{cbv} \to \alpha) \to \alpha$, irrespective of whether or not σ is a quantified type. The definition of $||\sigma||_{cbv}$ remains the same, in keeping with the intuition that a *value* of polymorphic type is a value of all instances of that type.

These modifications to the type transform take us beyond the Damas-Milner type system since now quantified types can occur as the domains of functions. Accordingly we generalize the target language of the CPS transform to the language DM^+ which is defined by eliminating the distinction between monotypes and polytypes in DM. The resulting system is equivalent to full polymorphic type assignment [16]. Although the decidability of type inference in DM^+ remains an open problem, this is not important for our purposes. The main property of DM^+ that we require is closure under β-reduction [16].

With these changes to the type transformation and the associated enrichment of the target type system, the putative proof of type preservation goes through in the case of let expressions, but now polymorphic generalization is problematic. Specifically, if $\mathsf{DM} \vdash \Gamma \rhd e : \sigma$ with $t \notin FTV(\Gamma)$, then by induction $\mathsf{DM}^{+} \vdash ||\Gamma||_{cbv} \rhd |e|_{cbv} : |\sigma|_{cbv}$, and $t \notin FTV(||\Gamma||_{cbv})$. It follows that $\mathsf{DM}^{+} \vdash ||\Gamma||_{cbv} \rhd |e|_{cbv} : \forall t.|\sigma|_{cbv}$, but this is not enough — we must show that $\mathsf{DM}^{+} \vdash ||\Gamma||_{cbv} \rhd |e| : |\forall t.\sigma|_{cbv}$, and there is no evident way to proceed.

In fact typing is not preserved. The argument proceeds by way of the extension of DM with continuation passing primitives. Under the call-by-value evaluation strategy, $\mathsf{DM} + \mathsf{cont}$ is unsound in that there is a term e such that e has a type τ, but whose value when evaluated under call-by-value fails to have type τ. Assuming that we have base types int and bool, and constants $0 : \mathsf{int}$ and $\mathsf{true} : \mathsf{bool}$, the following term is well-typed with type bool in $\mathsf{DM} + \mathsf{cont}$ but evaluates under call-by-value to 0, a constant of type int:[4]

$$\begin{array}{rcl} e_0 & = & \mathsf{let}\ f\ \mathsf{be}\ \mathsf{callcc}\,(\lambda k.\lambda x.\mathsf{throw}\,k\,\lambda y.x) \\ & & \mathsf{in}\ (\lambda x.\lambda y.y)\,(f\,0)\,(f\,\mathsf{true}) \end{array}$$

Using the typing rules of $\mathsf{DM} + \mathsf{cont}$, the let-bound identifier f is assigned the type $\forall t.t{\rightarrow}t$, and hence may be used at types $\mathsf{int}{\rightarrow}\mathsf{int}$ and $\mathsf{bool}{\rightarrow}\mathsf{bool}$ in the body. But the binding for f grabs the continuation associated with the body of the let expression and saves it. Upon evaluation of $f\,0$, the continuation is invoked and f is effectively re-bound to a constant function returning 0. The body is re-entered, $f\,0$ is evaluated once again (without difficulty), but then $f\,\mathsf{true}$ is evaluated, resulting in 0.

It follows that the call-by-value CPS transform for $\mathsf{DM} + \mathsf{cont}$ does not preserve typability in the sense of Meyer & Wand, for by the correctness of the call-by-value CPS transform for $\mathsf{DM}+\mathsf{cont}$, the expression $|e_0|\,\lambda x.x$ also evaluates to 0 (under call-by-name or call-by-value), yet has type bool, a violation of the subject reduction property of DM^{+} [16]. Consequently the call-by-value CPS transform does not verify the Meyer-Wand typing property even in the absence of continuation-passing primitives. For otherwise we could extend such a transform for the pure DM language to $\mathsf{DM} + \mathsf{cont}$ by first regarding callcc and throw as variables, and then replacing them by the expressions $||\mathsf{callcc}||_{cbv}$ and $||\mathsf{throw}||_{cbv}$. The substitution preserves typing, and results in a correct call-by-value CPS transform for $\mathsf{DM}+\mathsf{cont}$, which is impossible.

[4] This argument can be made without constants but at the cost of increased complexity. Constants of base type can easily be added to any of the transforms presented in this paper by defining $||c|| = c$, where c is a constant. Constants of non-base type must be handled on a case-by-case basis.

It is natural to inquire whether there is some other CPS transform for the pure DM language (without continuation-passing primitives) that is correct for the call-by-value interpretation and satisfies the Meyer-Wand typing property. Although we are unaware of such a transform, it is difficult to prove that none exists, in part because it is unclear what, in general, constitutes a correct CPS transform of an expression. We are assured, however, that any such transform must be substantially different from the standard call-by-value transform and its close relatives.

Theorem 6 *There is no call-by-value CPS transform $|e|$ for* DM *satisfying the following two conditions:*

1. *Equivalence: $|e|$ is operationally equivalent to $|e|_{cbv}$ under either the call-by-value or call-by-name operational semantics.*[5]
2. *Typing: If* $\mathsf{DM} \vdash \Gamma \triangleright e : \sigma$, *then* $\mathsf{DM}^+ \vdash ||\Gamma||_{cbv} \triangleright |e| : |\sigma|_{cbv}$.

Proof: Let $e_0' = [c/\mathsf{callcc}, t/\mathsf{throw}]e_0$, where e_0 is as in the counterexample above, so that $e_0 = [\mathsf{callcc}/c, \mathsf{throw}/t]e_0'$. Under the assumptions of the theorem, $|e_0'|$ is operationally equivalent to $|e_0'|_{cbv}$, and

$$\mathsf{DM}^+ \vdash c{:}||\sigma_{\mathsf{callcc}}||_{cbv}, t{:}||\sigma_{\mathsf{throw}}||_{cbv} \triangleright |e_0'| : |\mathsf{bool}|_{cbv}.$$

Now since

$$\mathsf{DM}^+ \vdash \bullet \triangleright ||\mathsf{callcc}||_{cbv} : ||\sigma_{\mathsf{callcc}}||_{cbv}$$

and

$$\mathsf{DM}^+ \vdash \bullet \triangleright ||\mathsf{throw}||_{cbv} : ||\sigma_{\mathsf{throw}}||_{cbv},$$

it follows that

$$\mathsf{DM}^+ \vdash [||\mathsf{callcc}||_{cbv}/c, ||\mathsf{throw}||_{cbv}/t||]\; |e_0'| : |\mathsf{bool}|_{cbv}.$$

Now

$$[||\mathsf{callcc}||_{cbv}/c, ||\mathsf{throw}||_{cbv}/t||]\; |e_0'|_{cbv}$$

is operationally equivalent to

$$[||\mathsf{callcc}||_{cbv}/c, ||\mathsf{throw}||_{cbv}/t||]\; |e_0'|.$$

But the former term is just $|e_0|_{cbv}$ as defined in Section 2. By the correctness of this transform with respect to the call-by-value operational semantics, we have that $|e_0|_{cbv}\,(\lambda x.x)$ evaluates to 0, and hence

$$[||\mathsf{callcc}||_{cbv}/c, ||\mathsf{throw}||_{cbv}/t||]|e_0'|\,(\lambda x.x)$$

must also evaluate to 0. But the latter term has type bool, in contradiction to the subject reduction property of DM^+. ∎

[5] Two terms e_1 and e_2 are *operationally equivalent* iff they are indistinguishable in all program contexts [18].

The conditions of the theorem leave open the possibility of there being a call-by-value transform for DM that is operationally inequivalent to the standard one, or of there being a type transform for which a Meyer-Wand-like type preservation theorem can be proved but for which either $||\mathsf{callcc}||_{cbv}$ or $||\mathsf{throw}||_{cbv}$ fails to have the required type. The typing condition seems natural and well-motivated; we know of no other uniform assignment of types for which the call-by-value CPS transform validates a Meyer-Wand-like type preservation theorem. On the other hand, the requirement of operational equivalence is rather strong. In particular, it imposes stringent conditions on the transformation of terms with free variables which we exploited in the proof of the theorem. It seems plausible that a variant transform that fails to satisfy the operational equivalence criterion may exist, but we have no evidence to support this conjecture.

4.4. Related Transforms

It seems worthwhile, however, to point out that there is a variant type transform that "almost" works. This transform is defined by taking $||\forall t.\sigma|| = \forall t.|\sigma|$, and $|\sigma| = (||\sigma|| \to \alpha) \to \alpha$. The intuition behind this choice is to regard polymorphic instantiation as a "serious" computation (in roughly the sense of Reynolds [19]). This interpretation is at variance with the usual semantics of ML polymorphism since it admits primitives that have non-trivial computational effects when polymorphically instantiated. Nevertheless, we can use this type transform to extend the Meyer-Wand theorem to a variant call-by-value CPS transform for DM^- and to a variant call-by-name CPS transform for DM, provided that we restrict attention to programs of monomorphic type. It does not provide a variant call-by-value CPS transform for full DM because of the way in which polymorphic generalization is handled.

To make these observations precise, we sketch the definitions of variant CPS transforms based on this type interpretation. The main idea is to define the CPS transform by induction on typing derivations so that the effect of polymorphic generalization and instantiation can be properly handled. We give here only the two most important clauses, those governing the rules GEN and INST:

$$\begin{array}{rcl} |\Gamma \rhd e : \forall t.\sigma| & = & \lambda k.k\,|e|, \ \mathsf{where} \\ |\Gamma \rhd e : \sigma| & = & |e| \\ \\ |\Gamma \rhd e : [\tau/t]\sigma| & = & \lambda k.|e|\,(\lambda x.x\;k), \ \mathsf{where} \\ |\Gamma \rhd e : \forall t.\sigma| & = & |e| \end{array}$$

This definition may be extended to the other inference rules in such a way as to implement either a call-by-name or call-by-value interpretation of application. However, the transform fails (in general) to agree with the usual (call-by-value or call-by-name) ML semantics on terms of polymorphic type. Specifically, the transformation of a GEN rule applies the current continuation to the suspended computation of e. If this continuation is not strict, then an expression that would abort in ML terminates normally after transformation into CPS. For example, suppose that hd and nil are constants with polymorphic types $\forall t.t\,\mathsf{list}{\rightarrow}t$ and $\forall t.t\,\mathsf{list}$, respectively. We may derive the judgement $\bullet \triangleright \mathsf{hd\,nil} : \forall t.t\,\mathsf{list}$ using the given typings for hd and nil, and the rules for application and polymorphic generalization. Using the variant rules given above, the transform of $\mathsf{hd\,nil}$ determined by this typing derivation has the form $\lambda k.k\,|\mathsf{hd\,nil}|$. But this expression evaluates normally on a non-strict continuation k, whereas under the usal ML semantics the expression should abort regardless of whether it is used in a subsequent computation.

By restricting attention to programs of monomorphic type, we may obtain correct variant CPS transforms for DM^- (under call-by-value) and DM (under call-by-name). This is essentially because in DM^- under call-by-value there are no non-trivial polymorphic computations, and because in DM under call-by-name the semantics is defined by substitution. But the above argument shows that the variant call-by-value transform is incorrect for DM under call-by-value. Specifically, it fails to correctly implement the usual ML semantics for expressions such as $\mathsf{let}\,x\,\mathsf{be\,hd\,nil\,in}\,0$ (which, under the above transformation, yields result 0). This suggests that it would be advantageous to separate the two roles of let in ML. Specifically, our results imply the soundness of the variant of ML obtained by evaluating let's sequentially but admitting no polymorphism, and by adding a new definition form, $\mathsf{def}\,x\,\mathsf{is}\,e_1\,\mathsf{in}\,e_2$, which admits polymorphism but is evaluated “by name”.

5. Conclusion

The Meyer-Wand typing theorem for the call-by-value CPS transform for the simply-typed λ-calculus establishes a simple and natural relationship between the type of a term and the type of its call-by-value CPS transform. Meyer and Wand exploited this relationship in their proof of the equivalence of the direct and continuation semantics of $\lambda^{\rightarrow}$ [13]. A minor extension of this result may be used to establish the soundness of typing for $\lambda^{\rightarrow} + \mathsf{cont}$, the extension of $\lambda^{\rightarrow}$ with continuation-passing primitives [4], under call-by-value.

In this paper we have presented a systematic study of the extension of the Meyer-Wand theorem to the Damas-Milner system of polymorphic type assignment. Our main positive results are the extension of the Meyer-Wand theorem to the call-by-value interpretation of a restricted form of polymorphism, and to the call-by-name interpretation of the unrestricted language. These results have as a consequence the soundness (in the sense of Damas and Milner [2]) of these programming languages. We have also argued that there is no "natural" call-by-value CPS transform for the unrestricted language, but this leaves open the possibility of finding a transformation that is radically different in character from the usual one.

Our investigation makes clear that there is a fundamental tension between implicit polymorphism and the call-by-value interpretation of let. This is consistent with earlier results establishing the inconsistency between unrestricted polymorphism and polymorphic reference types [22, 12] and first-class continuations [9, 10]. The source of the inconsistency may be traced to conflicting motivations for the static and dynamic semantics of the language. The polymorphic typing rule for let is motivated by a substitution principle — a let-bound variable is deemed to have exactly those types that may be ascribed to the expression to which it is bound. In particular, if a let-bound variable may be used at multiple types, provided that each of these may be ascribed to the bound expression separately.

But this principle, appealing though it may be in isolation, is inconsistent with the "evaluate-once" semantics of let expressions in the call-by-value dynamic semantics. In a pure functional language it is possible to combine both principles without complication, achieving polymorphism without sacrificing code re-use. Once control or store effects are admitted, however, it becomes untenable to maintain both principles without restriction. We have outlined a CPS transform for the full Damas-Milner language that extends to continuation-passing primitives, but which is not quite equivalent to the usual call-by-value semantics. This suggests that a language in which polymorphic generalization and instantiation are semantically significant would be a well-behaved alternative to ML-style implicit polymorphism. This perspective has been explored in some detail in subsequent work of the authors [11].

6. Acknowledgments

We are grateful to Olivier Danvy, Bruce Duba, Tim Griffin, Mark Leone, and the anonymous referees for their helpful comments and suggestions.

References

1. William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for higher-level semantic algebra. In Maurice Nivat and John C. Reynolds,

editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, Cambridge, 1985.

2. Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

3. Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France. Extended version available as Technical Report CIS-92-28, Kansas State University.

4. Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

5. Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

6. Timothy Griffin. A formulae-as-types notion of control. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990. ACM, ACM.

7. Timothy Griffin. Private communication, January 1992.

8. Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. Revised and expanded version of [4]. To appear, *Journal of Functional Programming.*

9. Robert Harper and Mark Lillibridge. Announcement on the `types` electronic forum, July 1991.

10. Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN–CS–92–1426.

11. Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

12. Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 291–302, Orlando, FL, January 1991. ACM SIGACT/SIGPLAN.

13. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.

14. Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

15. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

16. John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *1986 ACM Conference on LISP and Functional Programming*, pages 308–319, August 1986.

17. Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

18. Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.

19. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972. ACM.

20. John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *Second Colloquium on Automata, Languages, and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 141–156, Saarbrücken, West Germany, 1974. Springer-Verlag.

21. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *1992 ACM Conference on LISP and Functional Programming*, pages 288–298, San Francisco, CA, June 1992. ACM.

22. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

23. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91–160, Department of Computer Science, Rice University, July 1991. To appear, *Information and Computation.*