A Substructural Type System for Delimited Continuations*

Oleg Kiselyov[1] and Chung-chieh Shan[2]

[1] FNMOC oleg@pobox.com
[2] Rutgers University ccshan@rutgers.edu

Abstract. We propose type systems that *abstractly* interpret small-step rather than big-step operational semantics. We treat an expression or evaluation context as a structure in a linear logic with hypothetical reasoning. Evaluation order is not only regulated by familiar focusing rules in the operational semantics, but also expressed by structural rules in the type system, so the types track control flow more closely. Binding and evaluation contexts are related, but the latter are linear.

We use these ideas to build a type system for delimited continuations. It lets control operators change the answer type or act beyond the nearest dynamically-enclosing delimiter, yet needs no extra fields in judgments and arrow types to record answer types. The typing derivation of a direct-style program desugars it into continuation-passing style.

1 Introduction

Cousot and Cousot [14] originally presented abstract interpretation by starting with a small-step operational semantics. Nevertheless, the typical type system abstractly interprets [13] a denotational or big-step operational semantics, in that each typing rule is the abstract interpretation of a denotational equation or a big-step evaluation judgment. Besides simplicity, one reason to start with such a semantics coarser than a transition system is to make the type system *syntax-directed*: the type of each expression, like its denotation or its big-step evaluation result, is determined by structural induction over the expression. However, when the language involves effects (especially control effects), it can be easier to specify and reason with a small-step semantics (especially evaluation contexts) [68].

A canonical effect that makes semantics and types harder to determine inductively is *delimited control* [25, 26]. With this effect, an expression may access its *delimited continuation* [17–19] or *delimited evaluation context* as a first-class value. This ability is useful in backtracking search [12, 18, 44, 59], direct-style representations of monads [30–32], the continuation-passing-style (CPS) transformation [17–19], partial evaluation [6, 7, 10, 16, 23, 33, 37, 47, 64], Web interactions [35, 53], mobile code [50, 56, 60], and linguistics [9, 58].

* Thanks to Olivier Danvy, Andrzej Filinski, Michael Stone, Philip Wadler, and the anonymous referees. The appendices to this paper are online at http://okmij.org/ftp/papers/delim-control-logic.pdf

This paper presents a new type system for delimited control as an example of typing by small-step abstract interpretation. Sect. 2 introduces delimited control, explains why answer types are crucial, and points out shortcomings in how the existing type systems track answer types. We then address the shortcomings in the rest of the paper. As a stepping stone, Sect. 3 introduces small-step typing using the familiar simply-typed λ-calculus. Sect. 4 then presents the $\lambda\xi_0$-calculus, a language with delimited control and small-step typing, and a type-checking algorithm for it. Our Twelf code online at `http://pobox.com/~oleg/ftp/packages/small-step-typechecking.tar.gz` implements type checking and contains numerous tests and sample derivations.

2 Answer types

The intuition behind delimited control may be conveyed by the two programs below. They are written in the language with delimited control formally defined in Fig. 3, enriched with string "literals" and concatenation $\frown$. The first program shows that a *control delimiter* alone does not affect the evaluation result.

$$\#\ \$\ \text{“Goldilocks said: ”} \frown (\#\ \$\ \text{“This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \quad (1)$$

This program contains two control delimiters, notated $\#\ \$\ \ldots$ where the subexpression $\ldots$ extends as far to the right as possible. (We pronounce # "reset" and \$ "plug".) The delimiter to the left surrounds the whole program, whereas the delimiter to the right surrounds the subexpression that computes what Goldilocks said. The program computes the string "Goldilocks said: This porridge is too hot. ". The delimiters affect the result only in the presence of a control operator that captures a delimited continuation, as in the following program.

$$\#\ \$\ \text{“Goldilocks said: ”} \frown \big(\#\ \$\ \text{“This porridge is ”} \frown \big(\xi_0 k.(k\ \$\ \text{“too hot”}) \frown (k\ \$\ \text{“too cold”}) \frown (k\ \$\ \text{“just right”})\big) \frown \text{“. ”}\big) \quad (2)$$

The control operator $\xi_0 k$ (pronounced "shift-zero k") removes, and binds k to, the current continuation up to the nearest dynamically-enclosing delimiter. Once this continuation is captured, an expression such as "too hot" can be plugged into it, notated $k\ \$$ "too hot" in the scope of k. In (2), k prepends "This porridge is " and appends ". " to any string plugged in, so the program computes "Goldilocks said: This porridge is too hot. This porridge is too cold. This porridge is just right. ". The prefix "Goldilocks said: " is not tripled because it is not captured in k.

These examples illustrate the distinction between delimited and undelimited control: a delimited continuation represents only a prefix of the default future of the computation. This prefix maps a subexpression's value (such as "too hot") to an intermediate result at the delimiter (such as "This porridge is too hot. "). A type system for delimited control must thus track these intermediate results' types as part of the effects of expressions. These types are called *answer types*.

The only answer type in (1) and (2) is that of strings, but real programs need different answer types at multiple delimiters. On one hand, it is useful for an

expression to access its delimited continuation beyond the nearest dynamically-enclosing delimiter: to combine multiple monadic effects [18, 31, 32], to normalize λ-terms with sums [7], and to simulate exceptions and mutable references [39] and dynamic binding [45]. These uses motivate type systems [38, 39, 51] that maintain a stack or heap of answer types. On the other hand, it is also useful for an expression to change the answer type, that is, to capture one delimited continuation then install another with a different answer type: to create functions [18], to find list prefixes [10], to represent parameterized monads [5], and to analyze questions and polarity in natural language [58]. These uses motivate a type system [17] that is sensitive to evaluation order.

Unfortunately, no existing type system for delimited control subsumes all others, so no clear choice emerges for practical use. Moreover, the existing type systems attach answer types to judgments and arrows as effect annotations [32, 34, 48, 61–63, 65]. These annotations obscure any logical interpretation of the types via the Curry-Howard correspondence [4, 36, 42].

For example, Danvy and Filinski [17] uses typing judgments of the form $\rho, \alpha \vdash E : \tau, \beta$, where ρ is a typing environment, α and β are answer types, E is a term, and τ is its type. If E changes the answer type, then the answer types α and β may differ. If α, τ, β are atomic, then this judgment indicates that the CPS transformation of E has the type $(\tau \to \alpha) \to \beta$ in the simply-typed λ-calculus. The typing rule for λ-expressions reads

$$\frac{[x \mapsto \sigma]\rho, \alpha \vdash E : \tau, \beta}{\rho, \delta \vdash \lambda x.\, E : (\sigma/\alpha \to \tau/\beta), \delta.} \tag{3}$$

If $\sigma, \alpha, \tau, \beta$ are atomic, then the type $\sigma/\alpha \to \tau/\beta$ above indicates that the CPS transformation of $\lambda x.\, E$ has the simple type $\sigma \to (\tau \to \alpha) \to \beta$.

The extra fields for answer types in these judgments and arrow types still leave no room for delimiters beyond the nearest dynamically-enclosing one. Also, the comma and slash are not logical connectives in their own right, so the logical interpretation of the extra fields is unclear, unlike with undelimited control [36] or the simply-typed λ-calculus, which do not have varying answer types. Kameyama [42] logically interprets a static variant of Danvy and Filinski's system that does not allow changing the answer type. Ariola et al. [4] embed Danvy and Filinski's type system in subtractive logic, but the embedding is not full: the target includes undelimited control but the source does not.

In sum, we want a type system for delimited control that accommodates an arbitrary number of changing answer types. We achieve this goal by assigning types not just to expressions but also to evaluation contexts, as guided by CPS. For example, the delimited continuation k in (2) yields a string answer when a string is plugged into it; we write this type as $\mathrm{string} \uparrow \mathrm{string}$. The ξ_0-term in (2) is an expression that yields a string answer when it is plugged into such a delimited continuation; we write this type as $(\mathrm{string} \uparrow \mathrm{string}) \downarrow \mathrm{string}$. The return types to the right of the function-like *connectives* $\uparrow$ and $\downarrow$ are answer types.

An evaluation context is not usually part of an expression. Thus, to assign types to evaluation contexts, we need to revise our notion of a syntax-directed type system. We do so first for the λ-calculus, then return to delimited control.

Statements $M ::= C \$ E$

Terms $E, F ::= V \mid FE$

Values $V ::= x \mid \lambda x.\, E$

Coterms $C ::= \# \mid E, C \mid C; V$

Types $T, U ::= U \to T \mid \text{string} \mid \text{int} \mid \cdots$

Cotypes $S ::= U \uparrow T$

Statement contexts $M[\,] ::= C[\,] \$ E \mid C \$ E[\,]$

Term contexts $E[\,] ::= [\,] \mid E[\,]E \mid EE[\,]$

Coterm contexts $C[\,] ::= E[\,], C \mid E, C[\,] \mid C[\,]; V$

Statement equality

$$C \$ FE = E, C \$ F \qquad C \$ VE = C; V \$ E$$

Transitions

$$C \$ (\lambda x.\, E)V \rightsquigarrow C \$ E\{x \mapsto V\}$$

Typing

$$\frac{\begin{array}{c}[x:U]\\ \vdots\\ \#\$E:T\end{array}}{\lambda x.\, E : U \to T}\,\lambda \qquad \frac{[x:U] \quad \begin{array}{c}\vdots\\ M[x]:T\end{array}}{M[F]:T}\,M[U] \qquad \frac{\begin{array}{c}[x:U]\\ \vdots\\ Vx:T\end{array}}{V : U \to T}\,{\to}\text{I} \qquad \frac{\begin{array}{c}[x:U]\\ \vdots\\ C\$x:T\end{array}}{C : U \uparrow T}\,{\uparrow}\text{I}$$

$$\frac{V:U}{\#\$V:U}\,\# \qquad \frac{F : U \to T \quad E : U}{FE : T}\,{\to}\text{E} \qquad \frac{C : U \uparrow T \quad E : U}{C \$ E : T}\,{\uparrow}\text{E}$$

Fig. 1. Warm-up: the simply typed λ-calculus with small-step typing. For uniformity with Fig. 3 below, the notation is somewhat unconventional: we use the metavariable E for terms and also $E[\,]$ for a term with a term-hole. The only variable binder is λx. Following the Barendregt variable convention, the variable x in the $M[U]$, $\to$I, and $\uparrow$I rules is to be chosen fresh, not to occur free in the conclusion. The assumption $x : U$ discharged in these rules always occurs exactly once, because the hole $[\,]$ appears linearly in a context and no rule duplicates subterms.

3 Warm-up: small-step typing

Fig. 1 shows a type system for the pure λ-calculus that includes small-step as well as big-step abstract interpretation. The purpose of this system is to prepare for the main development in Sect. 4. Many aspects of this system seem contrived and redundant when taken alone, but they are necessary for delimited control. The accompanying Twelf code in `lfix-calc.elf` implements small-step abstract interpretation for this language and contains numerous sample derivations.

Besides *terms* E, this language defines two other syntactic categories: *coterms* C and *statements* M. Whereas a term can contain subterms, a statement is a complete program like a top-level term (a common notion in small-step semantics). A statement is formed by, and decomposes into, plugging a term into a coterm. This distinction between terms and statements is refined in Sect. 4.

A *statement context* $M[\ ]$ (respectively *term context* $E[\ ]$, *coterm context* $C[\ ]$) is a statement (respectively term, coterm) with a hole that can be filled by any term, such that the hole is not under a binder λx. We write $M[E]$ for the context $M[\ ]$ filled with the term E. Judgments of the form $M : T$, $E : T$, and $C : S$ assign types T and cotypes S to statements M, terms E, and coterms C.

A coterm is an evaluation context, that is, a defunctionalized continuation of a substitution-based evaluation function [1–3, 21]: the coterm $\#$ is the identity continuation; the coterm E, C means to apply to the argument term E then continue with the coterm C; the coterm $C; V$ means to apply the function value V then continue with the coterm C. Formally, a simple bijection maps coterms C to term contexts $E[\ ]$ in which only values appear to the left of the hole.

Definition 1. *Associate with each coterm C a term context $C^\dagger[\ ]$ by induction:*

$$\#^\dagger[\ ] = [\ ], \qquad (E, C)^\dagger[\ ] = C^\dagger\big[[\ ]E\big], \qquad (C; V)^\dagger[\ ] = C^\dagger\big[V[\ ]\big]. \tag{4}$$

Every coterm "comes with its own control delimiter", in that it always ends in the identity continuation $\#$. Hence a coterm represents a complete (delimited) continuation, not a list of stack frames. It makes no sense to "concatenate" coterms, for example to try to combine the coterms $E_1, \#$ and $E_2, \#$ into $E_1, (E_2, \#)$.

A statement, of the form $C \,\$\, E$ (pronounced "plug"), represents the term E plugged into the coterm C. It is a state of the CK machine [25, 27, 28]. A statement can also be understood as a zipper [41] over a term.

Among the binary constructors, $\$$ has the lowest precedence, and juxtaposition (for function application) the highest. All binary constructors associate to the right, except juxtaposition associates to the left.

3.1 A substructural logic for expressions and evaluation contexts

Two *statement equality* rules enforce left-to-right, call-by-value evaluation, as evaluation contexts [25] and focusing [20] do in other accounts. Formally, our equality rules are equations in the multisorted algebra of statements, terms, and coterms, as well as the following reversible typing rules.

$$\frac{C \,\$\, FE : T}{E, C \,\$\, F : T} = \qquad\qquad \frac{C \,\$\, VE : T}{C; V \,\$\, E : T} = \tag{5}$$

These rules let us navigate around a term using a statement as a zipper.

Proposition 1. *The statement equality rules equate the statements $C_1 \,\$\, E_1$ and $C_2 \,\$\, E_2$ iff the terms $C_1^\dagger[E_1]$ and $C_2^\dagger[E_2]$ are equal.*

Because $C^\dagger[\ ]$ is always an evaluation context for left-to-right, call-by-value evaluation, Prop. 1 ties evaluation order to transitions in the dynamic semantics as well as types in the static semantics. The $\#$ typing rule makes $\# \,\$\, V$ effectively equivalent to V, so as to type $\#$ as the identity continuation.

The separator $:$ in judgments is the turnstile in a substructural logic. This logic has four sorts (namely statements, terms, values, and coterms). It allows no exchange, associativity, weakening, or contraction except by the structural

rules in (5). It builds structures from values using six multiplicative-conjunctive punctuation marks [54], or *modes*: four binary (juxtaposition and \$, ;), one nullary (#), and one unary (the implicit coercion from a value to a term). This logic is thus a restricted *multimodal type-logical grammar* (TLG).

Multimodal TLG is a generalization of the Lambek calculus [46] whose proof theory and Kripke semantics are well-studied and well-behaved: there are sound and complete natural-deduction and sequent calculi with cut elimination [49, 52]. Our statements, terms, and coterms (to the left of the turnstile) are TLG structures, restricted to be sort-correct. Our types and cotypes (to the right of the turnstile) are TLG formulae, restricted to use only two implication connectives $\rightarrow$ and $\uparrow$ out of the four pairs available in TLG (one pair per binary mode).

Viewed as a substructural logic, this type system is mostly familiar. The $\rightarrow$I, $\rightarrow$E, $\uparrow$I, and $\uparrow$E rules establish $\rightarrow$ as the right-implication of juxtaposition and $\uparrow$ as the right-implication of \$. As in the Lambek calculus, x occurs linearly in the premises of $\rightarrow$I and $\uparrow$I; these premises could be just $VU : T$ and $C \,\$\, U : T$ if, in the spirit of abstract interpretation, types were values. Of these rules, only $\rightarrow$I is needed for delimited control in Sect. 4, but we include introduction and elimination rules for all binary connectives to relate them to TLG. Still, as in the original Lambek calculus, no binary mode comes with any product connective, such as any connective $*$ such that $F : T$ and $E : U$ justify $FE : T * U$. This distinction between modes and connectives is standard in substructural logic.

In contrast to the binary modes, the (implicit) unary mode for coercing values into terms does correspond to an (implicit) product connective. The $M[U]$ rule is the standard elimination rule for this connective in natural deduction. This rule lets us use any expression with a pure type—which in the pure λ-calculus is any type—as a value. This rule is more general than the $\uparrow$E rule in that it allows substituting a nonvalue F into an operand position in $M[\ ]$ even if the corresponding (preceding) operator position contains a nonvalue as well. In particular, the equality rules can treat a term F of a pure type U as a value x.

Finally, the familiar λ rule creates a function value. Unlike in the $\rightarrow$I and $\uparrow$I rules, the bound variable x in the λ rule may appear multiple times, or not at all, in the body E of the abstraction $\lambda x.\, E$. In other words, a λ-bound variable is intuitionistic rather than substructural: it admits weakening and contraction.

The transition rule in Fig. 1 is β-reduction, restricted to when the argument V is a value. Transitions operate on statements, not terms: to run a term E as a complete program, we run the statement $\# \,\$\, E$.

3.2 Normalizing small-step derivations to big-step derivations

Despite all these rules, the system is equivalent to the simply-typed λ-calculus.

Proposition 2. *Write $E :: T$ if the term E has the type T in the simply-typed λ-calculus. Then, under any typing assumptions $x_1 : T_1$, $x_1 :: T_1$, …, $x_n : T_n$, $x_n :: T_n$: (a) $E : T$ iff $E :: T$. (b) $C \,\$\, E : T$ iff $C^\dagger[E] :: T$. (c) $C : U \uparrow T$ iff $\lambda x.\, C^\dagger[x] :: U \rightarrow T$.*

Proof. [$\Rightarrow$] By induction on a derivation in our system.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{inc} : \text{int} \to \text{int} \quad [x : \text{int}]^2}{\text{inc}\, x : \text{int}}\,{\to}\text{E} \qquad \dfrac{[y : \text{int}]^1}{\#\ \$\ y : \text{int}}\,\#}{\#\ \$\ \text{inc}\, x : \text{int}}\,M[U]^1}{\#;\, \text{inc}\ \$\ x : \text{int}}\,=}{\#;\, \text{inc} : \text{int} \uparrow \text{int}}\,{\uparrow}\text{I}^2 \qquad \dfrac{\text{inc} : \text{int} \to \text{int} \quad 2 : \text{int}}{\text{inc}\, 2 : \text{int}}\,{\to}\text{E}}{\#;\, \text{inc}\ \$\ \text{inc}\, 2 : \text{int}}\,{\uparrow}\text{E}}{\#\ \$\ \text{inc}\,(\text{inc}\, 2) : \text{int}}\,=
$$

$$
\dfrac{\dfrac{\text{inc} : \text{int} \to \text{int} \quad \dfrac{\text{inc} : \text{int} \to \text{int} \quad 2 : \text{int}}{\text{inc}\, 2 : \text{int}}\,{\to}\text{E}}{\text{inc}\,(\text{inc}\, 2) : \text{int}}\,{\to}\text{E} \qquad \dfrac{[y : \text{int}]^1}{\#\ \$\ y : \text{int}}\,\#}{\#\ \$\ \text{inc}\,(\text{inc}\, 2) : \text{int}}\,M[U]^1
$$

Fig. 2. Two derivations of “$\#\ \$\ \text{inc}\,(\text{inc}\, 2) : \text{int}$” from “$\text{inc} : \text{int} \to \text{int}$” and “$2 : \text{int}$”

[⇐] (a) By induction on a simple-type derivation, using our →E rule and
$$
\dfrac{\dfrac{\begin{matrix}[x : U]^2 \\ \vdots \\ E : T\end{matrix} \quad \dfrac{[y : T]^1}{\#\ \$\ y : T}\,\#}{\#\ \$\ E : T}\,M[U]}{\lambda x.\, E : U \to T}\,\lambda^2.
$$
(b) Feed the conclusion of
$$
\dfrac{\begin{matrix}\vdots\ \text{Use (a)} \\ C^{\dagger}[E] : T\end{matrix} \quad \dfrac{[y : T]^1}{\#\ \$\ y : T}\,\#}{\#\ \$\ C^{\dagger}[E] : T}\,M[U]^1
$$
to Prop. 1. (c) Derive $C\ \$\ x : U$ by (b), then use ↑I. □

Because the simply-typed λ-calculus enjoys preservation, progress, and decidable type reconstruction, our system does as well.

Fig. 2 shows two typing derivations of the same statement $\#\ \$\ \text{inc}\,(\text{inc}\, 2)$, where the value inc is the integer increment function. At the bottom is the result of converting the familiar derivation in the simply-typed λ-calculus to our system using part (b) of Prop. 2. We call this derivation *big-step* because it follows the applicative structure of the expression: it determines the type of $\text{inc}\,(\text{inc}\, 2)$ from the type of its parts inc and $\text{inc}\, 2$. At the top is a *small-step* derivation, which separates the expression $\text{inc}\, 2$ from its evaluation context $\#;\, \text{inc}$. This derivation represents the term $\text{inc}\, 2$ by the variable x in $\#;\, \text{inc}\ \$\ x$. Thus x in the typing context is an abstract value, in the sense of abstract interpretation [13, 14].

Because all expressions in the λ-calculus are pure, they can be derived by both big-step and small-step. (In Sect. 4, impure expressions—which incur delimited-control effects—require small-step derivations.) These derivations are related by a normalization process (not cut elimination, because our type system is based on natural deduction rather than sequents) detailed in Appendix A. There we normalize the small-step derivation in Fig. 2 to the big-step derivation below.

4 Delimited control

Fig. 3 defines the static and dynamic semantics of the $\lambda\xi_0$-calculus, a new language with delimited control. The most prominent difference between this system

Terms $E, F ::= V \mid FE \mid C \,\$\, E \mid \xi_0 k.\, E$

Values $V ::= x \mid \lambda x.\, E$

Coterms $C ::= k \mid \# \mid E, C \mid C; V$

Types $T ::= U \mid S \downarrow T$

Pure types $U ::= U \to T \mid \text{string} \mid \text{int} \mid \cdots$

Cotypes $S ::= U \uparrow T$

Term contexts $E[\,] ::= [\,] \mid E[\,]E \mid EE[\,] \mid C[\,] \,\$\, E \mid C \,\$\, E[\,]$

Coterm contexts $C[\,] ::= E[\,], C \mid E, C[\,] \mid C[\,]; V$

Term equality

$$C \,\$\, FE = E, C \,\$\, F \qquad C \,\$\, VE = C; V \,\$\, E \qquad \# \,\$\, V = V$$

Transitions

$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, (\lambda x.\, E)V \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, E\{x \mapsto V\}$$
$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, C \,\$\, (\xi_0 k.\, E) \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, E\{k \mapsto C\}$$

Typing

$$\frac{\begin{array}{c}[x : U] \\ \vdots \\ E : T\end{array}}{\lambda x.\, E : U \to T}\,\lambda \qquad \frac{\begin{array}{c}[k : S] \\ \vdots \\ E : T\end{array}}{\xi_0 k.\, E : S \downarrow T}\,\xi_0 \qquad \frac{\begin{array}{c}[x : U] \\ \vdots \\ F : U \quad E[x] : T\end{array}}{E[F] : T}\,E[U]$$

$$\frac{\begin{array}{c}[x : U] \\ \vdots \\ Vx : T\end{array}}{V : U \to T} \to\text{I} \qquad \frac{\begin{array}{c}[k : S] \\ \vdots \\ k \,\$\, E : T\end{array}}{E : S \downarrow T} \downarrow\text{I} \qquad \frac{\begin{array}{c}[x : U] \\ \vdots \\ C \,\$\, x : T\end{array}}{C : U \uparrow T} \uparrow\text{I}$$

$$\frac{F : U \to T \quad E : U}{FE : T} \to\text{E} \qquad \frac{C : S \quad E : S \downarrow T}{C \,\$\, E : T} \downarrow\text{E} \qquad \frac{C : U \uparrow T \quad E : U}{C \,\$\, E : T} \uparrow\text{E}$$

Fig. 3. The $\lambda\xi_0$-calculus: syntax and semantics

and Fig. 1 is new non-value terms of the form $\xi_0 k.\, E$. These terms have *impure* types of the form $(U \uparrow T_1) \downarrow T_2$. As we discussed for the example (2) above, such a type means that, when the term is plugged into a coterm of cotype $U \uparrow T_1$ (in other words, a coterm which yields an answer of type T_1 when a value of type U is plugged into it), the combination yields an answer of type T_2. All other types are *pure*. We distinguish pure types by using the metavariable U rather than T.

A term of the form $\xi_0 k.\, E$ in the $\lambda\xi_0$-calculus may capture not just its immediately surrounding delimited continuation in the covariable k but also delimited continuations beyond the nearest dynamically-enclosing delimiter, if the body E invokes another control operator $\xi_0 k.\, E'$. Hence our primitive control operator is *dynamic* [17–19]: the answer types T_1 and T_2 in the impure type $(U \uparrow T_1) \downarrow T_2$ may themselves be impure. We also allow changing the answer type, so T_1 and T_2 may differ. Thus our type system is the first to achieve both desiderata in Sect. 2: to reach beyond the nearest delimiter and to change the answer type.

More precisely, our ξ_0 is not Danvy and Filinski's *shift* but the variation in their Appendix C [17], which we pronounce "shift-zero". In the untyped setting, ξ_0, shift, *control* [25, 26], and their variants [38–40] are all macro-expressible in terms of each other [43, 57]. In the typed setting, ξ_0 easily emulates shift [17] (shift $k.\, E$ translates to $\xi_0 k.\, \# \,\$\, E$), but it remains to relate ξ_0 to other type systems of control. In particular, unlike Gunter et al.'s system [38, 39], we assure that a program of a pure type never gets stuck due to a missing delimiter. We are also able to type more terms, for example $\xi_0 k.\, \lambda x.\, k \,\$\, x$, which changes the answer type.

Existing languages with delimited control generally introduce a primitive expression form, called "reset" or "prompt", to insert a control delimiter. In contrast, our language includes terms of the form $C \,\$\, E$, which means to plug the term E into the coterm C. We call these terms *statements*. Unlike in Fig. 1, a statement is a term. Because every coterm "comes with its own delimiter" in that it always ends in either the identity continuation # or a covariable k, our term $\# \,\$\, E$ serves the purpose of "reset E" or "prompt E" in previous work, even though \$ alone is not a delimiter. Now that $\# \,\$\, E$ is as much a term as E, we replace the typing rule # in Fig. 1 by a new term equality rule $\# \,\$\, V = V$.

To evaluate programs that use delimited control, Fig. 3 defines two transition rules. The first rule substitutes an argument value V into the body E in $\lambda x.\, E$, whereas the second rule substitutes an argument coterm C into the body E in $\xi_0 k.\, E$. Both rules operate inside a term context $C_1 \,\$\, \cdots \,\$\, C_n \,\$\, [\,]$, where $n \geq 0$. This term context is the *metacontinuation* [67] that appears in CPS semantics [17–19] and abstract machines [11, 24] for delimited control.

As in Sect. 3.1, the term equality rules in Fig. 3 are equations in the multi-sorted algebra of terms and coterms as well as the reversible typing rules

$$\frac{E'[C \,\$\, F\, E] : T}{E'[E, C \,\$\, F] : T} = \qquad \frac{E'[C \,\$\, V\, E] : T}{E'[C; V \,\$\, E] : T} = \qquad \frac{E'[V] : T}{E'[\# \,\$\, V] : T} = \tag{6}$$

and the corresponding rules replacing $E'[\,]$ and T by $C'[\,]$ and S.

As a substructural logic, the $\lambda\xi_0$-calculus has the same binary modes as Fig. 1, but allows not just the right-implication $\uparrow$ of the \$ mode but also its dual, the left-implication $\downarrow$. As before, this logic has neither negation nor multiple conclusions, so we interpret delimited control as multimodal *intuitionistic* logic via the Curry-Howard correspondence. The implication connectives each come with their own introduction and elimination rules, but $\downarrow$E and $\uparrow$E both conclude with $C \,\$\, E : T$. This apparent ambiguity is standard in type systems descended from the Lambek calculus [46]. Indeed, if the first of the two transition rules in Fig. 3 did not restrict β-reductions to take place only when the argument is a value, then the term $\# \,\$\, (\lambda x.\, \text{"call by name"})(\xi_0 k.\, \text{"call by value"})$ would transition not only to "call by value" but also to $\# \,\$$ "call by name", which is equal to "call by name". Just as the dynamic semantics restricts argument terms to values, the static semantics restricts argument types to pure types.

With both $\uparrow$ and $\downarrow$ present, each term has an infinite number of types. For example, Fig. 4 shows that a string also has the types (string $\uparrow$ int) $\downarrow$ int and

$$\dfrac{\dfrac{[k : \text{string} \uparrow \text{int}]^1 \quad x : \text{string}}{k \,\$\, x : \text{int}} \uparrow\text{E}}{x : (\text{string} \uparrow \text{int}) \downarrow \text{int}} \downarrow\text{I}^1 \qquad \dfrac{\dfrac{[k' : \text{int} \uparrow T]^2 \quad \dfrac{[k : \text{string} \uparrow \text{int}]^1 \quad x : \text{string}}{k \,\$\, x : \text{int}} \uparrow\text{E}}{k' \,\$\, k \,\$\, x : T} \uparrow\text{E}}{k \,\$\, x : (\text{int} \uparrow T) \downarrow T} \downarrow\text{I}^2}{x : (\text{string} \uparrow \text{int}) \downarrow (\text{int} \uparrow T) \downarrow T} \downarrow\text{I}^1$$

Fig. 4. Two type derivations for a string x

$(\text{string}\uparrow\text{int})\downarrow(\text{int}\uparrow T)\downarrow T$ for any T. In fact, the entailment relation of the logic is a partial order of subtyping, which we notate as $T_1 \leq T_2$. This relation is generated by $U \leq (U \uparrow T) \downarrow T$ along with congruences, covariance, and contravariance.

We show two example terms before stating formal properties. Appendix B gives the derivations. The term $(\xi_0 k.\, 1)(\xi_0 k.\, \text{“x”})$ has the type $S \downarrow \text{int}$ for any cotype S. The derivation is CPS-like, even though the term is in direct style like all our programs. Since the type is impure, the term may (and does) get stuck if run alone, but can appear in safe programs such as $\lambda x.(\xi_0 k.\, 1)(\xi_0 k.\, \text{“x”})$. As with (only) Danvy and Filinski's type system for shift [17], the answer type varies between int (for the subterm $\xi_0 k.\, 1$) and string (for $\xi_0 k.\, \text{“x”}$), but the overall answer type is int rather than string due to (left-to-right) evaluation order.

The impure term $\text{inc}(\xi_0 k.\, \xi_0 k'.\, \text{“x”})$ reaches beyond the nearest enclosing delimiter, which shift does not allow. It has the type $(\text{int} \uparrow T) \downarrow S \downarrow \text{string}$, where int is the type of the result of inc.

Proposition 3 (Preservation). *If $E[E_1] : T$ and $E_1 \rightsquigarrow E_2$ then $E[E_2] : T$.*

We sketch the proof by stating three lemmas. The first lemma is termed *direct compositionality on demand* by Barker [8]: a subterm of a typed term is typed.

Lemma 1. *If $E[E_1] : T$, then there exists some type T_1 such that $E_1 : T_1$ and whenever $E_1' : T_1$ we have $E[E_1'] : T$.*

The two remaining lemmas are less trivial than usual because the typing rules $\to$I, $\downarrow$I, and $\uparrow$I are not syntax-directed.

Lemma 2. *If $(\lambda x.\, E)V : T$ then $E\{x \mapsto V\} : T$.*

Lemma 3. *If $C \,\$\, (\xi_0 k.\, E) : T$ then $E\{k \mapsto C\} : T$.*

Proposition 4 (Progress). *If $C \,\$\, E : U$, then either $C \,\$\, E$ is a value (that is, $C = \#$ and E is a value) or $C \,\$\, E \rightsquigarrow E'$ for some E'.*

The small-step type-checking algorithm in Sect. 4.1 offers an appealing proof of this proposition: on one hand, it is correct with respect to the static semantics (Corollary 1 below); on the other hand, it is sound as an abstract interpretation of the dynamic semantics.

Proposition 5 (Determinism). *If $E \rightsquigarrow E_1'$ and $E \rightsquigarrow E_2'$, then $E_1' = E_2'$.*

Proposition 6 (Termination). *If $E : T$, then there is no infinite transition sequence $E \rightsquigarrow E_1 \rightsquigarrow E_2 \rightsquigarrow \cdots$.*

4.1 Type-checking algorithm

Fig. 5 shows a type-checking algorithm, expressed as moded inference rules, for a variant of our language in which binders are annotated with types and cotypes. The accompanying Twelf code in `lxi0-calc.elf` implements this algorithm. It produces a CPS-like derivation from a direct-style program. For the term $(\xi_0 k.\, 1)$ $(\xi_0 k.\, \text{“x”})$ above, our algorithm returns its type and even the cotypes of k's.

Our type checker performs abstract interpretation by traversing the term, just as the focusing process of an evaluator would traverse the term in search of a *focus*, and replacing nonvariable subterms by typed variables one by one.

Definition 2. *A* focus *is a term of the form* $V_1 V_2$, $C \mathbin{\$} E$, *or* $\xi_0 k{:}S.\, E$.

As explained in Sect. 3.2, each typed variable is an abstract value. In addition, our covariables are cotyped and can be thought of as abstract covalues.

Whereas the focusing process of an evaluator need only traverse a known term plugged into a known coterm, the type checker often needs to traverse a term without knowing what coterm it may be plugged into. For example, to check the function $\lambda x{:}\text{int}.\, \text{inc}(\xi_0 k{:}\text{int}\uparrow\text{string}.\, 3)$, the checker needs to visit the subterm $\xi_0 k{:}\text{int}\uparrow\text{string}.\, 3$ without knowing the context where the function will be applied and hence its body evaluated. In other words, the checker needs to use an equality rule $\dfrac{\langle\rangle;\text{inc} \mathbin{\$} (\xi_0 k{:}\text{int}\uparrow\text{string}.\, 3) : T}{\langle\rangle \mathbin{\$} \text{inc}(\xi_0 k{:}\text{int}\uparrow\text{string}.\, 3) : T}$ where $\langle\rangle$ represents an unknown coterm. To perform such traversals, we introduce the notion of an *incomplete coterm* $C\langle\rangle$, which is like a coterm but ends in $\langle\rangle$ rather than in $\#$ or k. Some of the checker's judgments use the notation $C\langle\rangle \mathbin{\dot{\$}} E$. Intuitively, $C\langle\rangle \mathbin{\dot{\$}} E$ means the term obtained by plugging E into the term context $C\langle\rangle^\dagger[\,]$ defined below.

Definition 3. *Associate with each incomplete coterm* $C\langle\rangle$ *a term context* $C\langle\rangle^\dagger[\,]$*:*

$$\langle\rangle^\dagger[\,] = [\,], \qquad (E, C\langle\rangle)^\dagger[\,] = C\langle\rangle^\dagger\big[[\,]E\big], \qquad (C\langle\rangle; V)^\dagger[\,] = C\langle\rangle^\dagger\big[V[\,]\big]. \tag{7}$$

The pair $C\langle\rangle$ and E is a zipper over the term $C\langle\rangle^\dagger[E]$, “unzipped” to display the subterm E. We also use $C \mathbin{\dot{\$}} E$ to mean the term $C \mathbin{\$} E$, where C is a (complete) coterm. Whereas $C \mathbin{\$} E$ is always a statement, the term $C\langle\rangle^\dagger[E]$ is only a statement when $C\langle\rangle$ is $\langle\rangle$ and E is a statement.

Unlike a ξ_0-bound covariable like k, this unknown coterm is not annotated with its cotype. Rather, the checker infers its (greatest) cotype, using a judgment form $S \Leftarrow C\langle\rangle \mathbin{\dot{\$}} E : T$ whose modes are rather special: the only output parameter is S. Any unknown coterm into which the term $C\langle\rangle^\dagger[E]$ is plugged for evaluation needs to have the cotype S in order to yield an answer of type T.

Our approach to “visit subterms in evaluation position before the context in which they occur” may be an instance of *tridirectional type-checking* [22]. The 4th–6th rules for $C\langle ?\rangle \mathbin{\dot{\$}} E \Rightarrow \hat{T}$ and the 2nd–4th rules for $\hat{S} \Leftarrow C\langle\rangle \mathbin{\dot{\$}} E : T$ are focusing rules: they traverse the applicative structure of E to find the next subterm to abstractly interpret according to the evaluation order.

Proposition 7 (Termination). *Under any typing assumptions* $x_1 : U_1, \ldots,$ $k_1 : S_1, \ldots,$ *any query using the inference rules in Fig. 5 terminates.*

Terms (annotated) $E, F ::= V \mid FE \mid C \,\$\, E \mid \xi_0 k{:}S.\, E$

Values (annotated) $V ::= x \mid \lambda x{:}U.\, E$

Incomplete coterms $C\langle\rangle ::= \langle\rangle \mid E, C\langle\rangle \mid C\langle\rangle; V$

Possibly incomplete coterms $C\langle ?\rangle ::= C \mid C\langle\rangle$

Judgments (hats indicate output as opposed to input parameters)

$$T_1 \le T_2 \qquad E \Rightarrow \hat{T} \qquad C\langle ?\rangle \,\dot{\varsigma}\, E \Rightarrow \hat{T} \qquad \hat{S} \Leftarrow C\langle\rangle \,\dot{\varsigma}\, E : T$$
$$V : \hat{U} \qquad k : \hat{S} \qquad C\langle ?\rangle \,\dot{\varsigma}\, U \Rightarrow \hat{T} \qquad \hat{S} \Leftarrow C\langle\rangle \,\dot{\varsigma}\, U : T$$

Initial query for type inference $\langle\rangle \,\dot{\varsigma}\, E \Rightarrow T$

Inference rules for $T_1 \le T_2$

$$\frac{U \text{ primitive}}{U \le U} \qquad \frac{U_1 \le U_2 \quad T_1 \le T_2}{U_1 \le (U_2 \uparrow T_1) \downarrow T_2} \qquad \frac{U_2 \le U_1 \quad T_1 \le T_2}{U_1 \to T_1 \le U_2 \to T_2} \qquad \frac{U_1 \le U_2 \quad T_2 \le T_1 \quad T_1' \le T_2'}{(U_1 \uparrow T_1) \downarrow T_1' \le (U_2 \uparrow T_2) \downarrow T_2'}$$

Inference rules for $E \Rightarrow \hat{T}$

$$\frac{V_1 : U_1 \to T \quad V_2 : U_2 \quad U_2 \le U_1}{V_1 V_2 \Rightarrow T} \qquad \frac{C \,\dot{\varsigma}\, E \Rightarrow T}{C \,\$\, E \Rightarrow T} \qquad \frac{\begin{array}{c}[k : S]\\ \vdots\\ \langle\rangle \,\dot{\varsigma}\, E \Rightarrow T\end{array}}{\xi_0 k{:}S.\, E \Rightarrow S \downarrow T}$$

Inference rules for $C\langle ?\rangle \,\dot{\varsigma}\, E \Rightarrow \hat{T}$

$$\frac{V : U}{\langle\rangle \,\dot{\varsigma}\, V \Rightarrow U} \qquad \frac{k : U_1 \uparrow T \quad V : U_2 \quad U_2 \le U_1}{k \,\dot{\varsigma}\, V \Rightarrow T} \qquad \frac{V : U}{\# \,\dot{\varsigma}\, V \Rightarrow U} \qquad \frac{C\langle ?\rangle; V \,\dot{\varsigma}\, E \Rightarrow T}{E, C\langle ?\rangle \,\dot{\varsigma}\, V \Rightarrow T}$$

$$\frac{C\langle ?\rangle \,\dot{\varsigma}\, V_1 V_2 \Rightarrow T}{C\langle ?\rangle; V_1 \,\dot{\varsigma}\, V_2 \Rightarrow T} \qquad \frac{F \text{ or } E \text{ is not a value} \quad E, C\langle ?\rangle \,\dot{\varsigma}\, F \Rightarrow T}{C\langle ?\rangle \,\dot{\varsigma}\, FE \Rightarrow T} \qquad \frac{E \Rightarrow U \quad C\langle ?\rangle \,\dot{\varsigma}\, U \Rightarrow T}{C\langle ?\rangle \,\dot{\varsigma}\, E \Rightarrow T}$$

$$\frac{E \Rightarrow (U \uparrow T_1) \downarrow T \quad C \,\dot{\varsigma}\, U \Rightarrow T_2 \quad T_2 \le T_1}{C \,\dot{\varsigma}\, E \Rightarrow T} \qquad \frac{E \Rightarrow (U \uparrow T_1) \downarrow T \quad S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, U : T_1}{C\langle\rangle \,\dot{\varsigma}\, E \Rightarrow S \downarrow T}$$

Inference rules for $\hat{S} \Leftarrow C\langle\rangle \,\dot{\varsigma}\, E : T$

$$\frac{V : U}{U \uparrow T \Leftarrow \langle\rangle \,\dot{\varsigma}\, V : T} \qquad \frac{S \Leftarrow C\langle\rangle; V \,\dot{\varsigma}\, E : T}{S \Leftarrow E, C\langle\rangle \,\dot{\varsigma}\, V : T} \qquad \frac{S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, V_1 V_2 : T}{S \Leftarrow C\langle\rangle; V_1 \,\dot{\varsigma}\, V_2 : T}$$

$$\frac{F \text{ or } E \text{ is not a value} \quad S \Leftarrow E, C\langle ?\rangle \,\dot{\varsigma}\, F : T}{S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, FE : T} \qquad \frac{E \Rightarrow U \quad S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, U : T}{S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, E : T}$$

$$\frac{E \Rightarrow (U \uparrow T_1) \downarrow T_2 \quad T_2 \le T \quad S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, U : T_1}{S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, E : T}$$

Other inference rules

$$\frac{\begin{array}{c}[x : U]\\ \vdots\\ \langle\rangle \,\dot{\varsigma}\, E \Rightarrow T\end{array}}{\lambda x{:}U.\, E : U \to T} \qquad \frac{\begin{array}{c}[x : U]\\ \vdots\\ C\langle ?\rangle \,\dot{\varsigma}\, x \Rightarrow T\end{array}}{C\langle ?\rangle \,\dot{\varsigma}\, U \Rightarrow T} \qquad \frac{\begin{array}{c}[x : U]\\ \vdots\\ S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, x : T\end{array}}{S \Leftarrow C\langle\rangle \,\dot{\varsigma}\, U : T}$$

Fig. 5. Type-checking algorithm for delimited control

Proposition 8. *Under any typing assumptions* $x_1 : U_1, \ldots, k_1 : S_1, \ldots$*:*

(a) $U_1 \leq U_2$ *iff* $E : U_1$ *entails* $E : U_2$*.*
(b) $E \Rightarrow T$ *iff* E *is a focus and* T *is a least type such that* $E : T$*.*
(c) $C \mathbin{¢} E \Rightarrow T$ *iff* T *is a least type such that* $C \mathbin{\$} E : T$*.*
(d) $C\langle\rangle \mathbin{¢} E \Rightarrow T$ *iff* T *is a least type such that* $C\langle\rangle^\dagger[E] : T$*.*
(e) $S \Leftarrow C\langle\rangle \mathbin{¢} E : T$ *iff* S *is a greatest cotype such that* $C\langle\rangle^\dagger[E] : S \downarrow T$*.*

For T *to be a least type such that* $E : T$ *means that* $E : T$ *and, for all* T'*, if* $E : T'$ *then* $T \leq T'$*. For* $U \uparrow T_1$ *to be a greatest cotype such that* $E : (U \uparrow T_1) \downarrow T_2$ *means that* $E : (U \uparrow T_1) \downarrow T_2$ *and, for all* U' *and* T_1'*, if* $E : (U' \uparrow T_1') \downarrow T_2$ *then* $U \leq U'$ *and* $T_1' \leq T_1$*.*

Corollary 1 (Correctness). $\langle\rangle \mathbin{¢} E \Rightarrow T$ *iff* T *is a least type such that* $E : T$*.*

Given that we annotate binders with types, this corollary shows the least type is unique because the type-checking algorithm is deterministic.

5 Conclusion

We model syntax using a substructural logic, such that terms in the language are structures in the substructural logic. This approach is standard in logical analyses of natural language but less common for programming languages. Types as abstract interpretations fall out, because structures in logic naturally contain formulas, and formulas are types—or abstract values—in the language. Hypothetical reasoning and structural rules in the logic model small-step transitions. Thus our type systems embody small-step abstract interpretation.

Beyond reconstructing the simply-typed λ-calculus, the fruit of our approach is the $\lambda\xi_0$-calculus. It is the first type system for delimited continuations in which control effects may change the answer type as well as act beyond the nearest dynamically-enclosing delimiter. The types are built up from binary connectives, which can clearly be interpreted as implication in intuitionistic logic. This feature is enabled by small-step typing, which lets us assign cotypes to delimited evaluation contexts. We also presented and implemented a type-checking algorithm, which again operates by small-step abstract interpretation.

Modeling syntax using a substructural logic lets us take advantage of established results, such as proof rules and reductions. It further draws attention to the similarity between hypothetical reasoning in the binding context and in the evaluation context—for example, between the ξ_0 and $\downarrow$I rules in Fig. 3. The two kinds of contexts differ simply in that the binding context is intuitionistic whereas the evaluation context is substructural: while the former is usually written to the left of $\vdash$ and admits weakening and contraction, the latter is usually written between $\vdash$ and : and only allows structural rules that model focusing. Typing derivations thus follow evaluation order and control flow: the typing derivation of a direct-style term is essentially its CPS transformation.

Our work exhibits a duality closely related to that for undelimited continuations [15, 29, 55, 66], but investigating the delimited case remains future work.

Given our analogy between small-step type-checking and small-step evaluation, we should relate our proof and term normalizations.

It remains to extend this work to other control operators, such as Felleisen's *control* [25, 26] and *named prompts* for delimiters, and to relate it to other substructural logics, such as those with additives and exponentials. We also look forward to mechanizing our proofs.

References

[1] Ager, Mads Sig, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proc. 5th intl. conf. principles & practice of declarative prog.*, 8–19.

[2] Ager, Mads Sig, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Info. Proc. Lett.* 90(5):223–232.

[3] ———. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comp. Sci.* 342(1):149–172.

[4] Ariola, Zena M., Hugo Herbelin, and Amr Sabry. 2004. A type-theoretic foundation of continuations and prompts. In *ICFP*, 40–53.

[5] Atkey, Robert. 2006. Parameterised notions of computation. In *MSFP 2006*, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society.

[6] Balat, Vincent, and Olivier Danvy. 2002. Memoization in type-directed partial evaluation. In *GPCE*, ed. Don S. Batory, Charles Consel, and Walid Taha, 78–92. LNCS 2487.

[7] Balat, Vincent, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 64–76.

[8] Barker, Chris. 2007. Direct compositionality on demand. In *Direct compositionality*, ed. Chris Barker and Pauline Jacobson, 102–131. New York: Oxford University Press.

[9] Barker, Chris, and Chung-chieh Shan. 2006. Types as graphs: Continuations in type logical grammar. *J. Logic, Lang. & Info.* 15(4):331–370.

[10] Biernacka, Małgorzata, Dariusz Biernacki, and Olivier Danvy. 2005. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Comp. Sci.* 1(2:5).

[11] Biernacka, Małgorzata, and Olivier Danvy. 2005. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comp. Sci.* To appear. BRICS Report RS-05-38.

[12] Biernacki, Dariusz, and Olivier Danvy. 2004. From interpreter to logic engine by defunctionalization. In *LOPSTR 2003*, ed. Maurice Bruynooghe, 143–159. LNCS 3018.

[13] Cousot, Patrick. 1997. Types as abstract interpretations. In *POPL*, 316–331.

[14] Cousot, Patrick, and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 238–252.

[15] Curien, Pierre-Louis, and Hugo Herbelin. 2000. The duality of computation. In *ICFP*, 233–243.

[16] Danvy, Olivier. 1996. Type-directed partial evaluation. In *POPL*, 242–257.

[17] Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

[18] ———. 1990. Abstracting control. In *Proc. conf. Lisp & funct. prog.*, 151–160.

[19] ———. 1992. Representing control: A study of the CPS transformation. *Math. Structures Comp. Sci.* 2(4):361–391.

[20] Danvy, Olivier, and Lasse R. Nielsen. 2001. Syntactic theories in practice. In *RULE 2001: 2nd intl. workshop on rule-based prog.*, ed. Mark van den Brand and Rakesh Verma, 358–374. Electron. Notes in Theor. Comp. Sci. 59(4), Elsevier.

[21] ———. 2004. Refocusing in reduction semantics. Report RS-04-26, BRICS.

[22] Dunfield, Joshua, and Frank Pfenning. 2004. Tridirectional typechecking. In *POPL*, 281–292.

[23] Dybjer, Peter, and Andrzej Filinski. 2002. Normalization and partial evaluation. In *APPSEM 2000*, ed. Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, 137–192. LNCS 2395.

[24] Dybvig, R. Kent, Simon L. Peyton Jones, and Amr Sabry. 2005. A monadic framework for delimited continuations. Tech. Rep. 615, Indiana U.

[25] Felleisen, Matthias. 1987. The calculi of λ_v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana U.

[26] ———. 1988. The theory and practice of first-class prompts. In *POPL*, 180–190.

[27] Felleisen, Matthias, and Matthew Flatt. 2006. Programming languages and lambda calculi. `http://www.cs.utah.edu/plt/publications/pllc.pdf`.

[28] Felleisen, Matthias, and Daniel P. Friedman. 1987. Control operators, the SECD machine, and the λ-calculus. In *Formal description of prog. concepts III*, ed. Martin Wirsing, 193–217. Elsevier.

[29] Filinski, Andrzej. 1989. Declarative continuations: An investigation of duality in programming language semantics. In *Proc. category theory & comp. sci.*, ed. David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, 224–249. LNCS 389.

[30] ———. 1994. Representing monads. In *POPL*, 446–457.
[31] ———. 1996. Controlling effects. Ph.D. thesis, CMU.
[32] ———. 1999. Representing layered monads. In *POPL*, 175–188.
[33] ———. 2001. Normalization by evaluation for the computational lambda-calculus. In *TLCA*, ed. Samson Abramsky, 151–165. LNCS 2044.
[34] Gifford, David K., and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proc. conf. Lisp & funct. prog.*, 28–38.
[35] Graunke, Paul Thorsen. 2003. Web interactions. Ph.D. thesis, Northeastern U.
[36] Griffin, Timothy G. 1990. A formulae-as-types notion of control. In *POPL*, 47–58.
[37] Grobauer, Bernd, and Zhe Yang. 2001. The second Futamura projection for type-directed partial evaluation. *Higher-Order & Symbolic Comp.* 14(2–3):173–219.
[38] Gunter, Carl A., Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Funct. prog. lang. & comp. architecture: 7th conf.*, ed. Simon L. Peyton Jones, 12–23.
[39] ———. 1998. Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/`.
[40] Hieb, Robert, and R. Kent Dybvig. 1990. Continuations and concurrency. In *Proc. 2nd symposium on principles & practice of parallel prog.*, 128–136.
[41] Huet, Gérard. 1997. The zipper. *J. Funct. Prog.* 7(5):549–554.
[42] Kameyama, Yukiyoshi. 2001. Towards logical understanding of delimited continuations. In *Proc. 3rd workshop on continuations*, ed. Amr Sabry, 27–33. Tech. Rep. 545, Indiana U.
[43] Kiselyov, Oleg. 2005. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Tech. Rep. 611, Indiana U.
[44] Kiselyov, Oleg, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP*, 192–203.
[45] Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP*, 26–37.
[46] Lambek, Joachim. 1958. The mathematics of sentence structure. *Amer. Math. Monthly* 65(3): 154–170.
[47] Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Proc. conf. Lisp & funct. prog.*, 227–238.
[48] Lucassen, John M. 1987. Types and effects: Towards the integration of functional and imperative programming. Ph.D. thesis, MIT.
[49] Moortgat, Michael. 1997. Categorial type logics. In *Handbook of logic and language*, ed. Johan F. A. K. van Benthem and Alice G. B. ter Meulen, chap. 2. Elsevier.
[50] Murphy, Tom, VII, Karl Crary, and Robert Harper. 2005. Distributed control flow with classical modal logic. In *CSL*, ed. C.-H. Luke Ong, 51–69. LNCS 3634.
[51] Murthy, Chetan R. 1992. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *Proc. workshop on continuations*, ed. Olivier Danvy and Carolyn Talcott, 49–71. Tech. Rep. STAN-CS-92-1426, Stanford U.
[52] Polakow, Jeff. 2001. Ordered linear logic and applications. Ph.D. thesis, CMU.
[53] Queinnec, Christian. 2004. Continuations and web servers. *Higher-Order & Symbolic Comp.* 17(4):277–295.
[54] Restall, Greg. 2000. *An introduction to substructural logics*. London: Routledge.
[55] Selinger, Peter. 2001. Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Math. Structures Comp. Sci.* 11:207–260.
[56] Sewell, Peter, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-level programming language design for distributed computation. In *ICFP*, 15–26.
[57] Shan, Chung-chieh. 2004. Shift to control. In *Proc. Scheme workshop*, ed. Olin Shivers and Oscar Waddell, 99–107. Tech. Rep. 600, Indiana U.
[58] ———. 2005. Linguistic side effects. Ph.D. thesis, Harvard U.
[59] Sitaram, Dorai. 1993. Handling control. In *PLDI*, 147–155.
[60] Sumii, Eijiro. 2000. An implementation of transparent migration on standard Scheme. In *Proc. Scheme workshop*, ed. Matthias Felleisen, 61–63. Tech. Rep. 00-368, Rice U.
[61] Talpin, Jean-Pierre, and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *J. Funct. Prog.* 2(3):245–271.
[62] ———. 1994. The type and effect discipline. *Info. & Comp.* 111(2):245–296.
[63] Thielecke, Hayo. 2003. From control effects to typed continuation passing. In *POPL*, 139–149.
[64] Thiemann, Peter. 1999. Combinators for program generation. *J. Funct. Prog.* 9(5):483–525.
[65] Wadler, Philip L. 1998. The marriage of effects and monads. In *ICFP*, 63–74.
[66] ———. 2003. Call-by-value is dual to call-by-name. In *ICFP*.
[67] Wand, Mitchell, and Daniel P. Friedman. 1988. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp & Symbolic Comp.* 1(1):11–37.
[68] Wright, Andrew K., and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Info. & Comp.* 115(1):38–94.

A Derivation normalization

Definition 4. *The* size *of a derivation is the number of inferences (horizontal rules) in it. In particular, the undischarged identity inference $x : U$ has size* 0*. The* measure *of a derivation is the number of inferences in it excluding the left premise of any use of the $M[U]$ rule (or of the two admissible rules in* (8) *below).*

Lemma 4. *The rules below are admissible. In terms of both size and measure, the $E[U]$ rule costs no inference, whereas the $C[U]$ rule costs one inference.*

$$\frac{\begin{array}{c}[x : U]\\ \vdots\\ F : U \quad E[x] : T\end{array}}{E[F] : T}\, E[U] \qquad\qquad \frac{\begin{array}{c}[x : U]\\ \vdots\\ F : U \quad C[x] : S\end{array}}{C[F] : S}\, C[U] \tag{8}$$

Proof. By induction on the second premise. □

Proposition 9 (Normalization). *No infinite derivation sequence $D_1, D_2, \ldots$ is such that each D_{i+1} results from applying a reduction in Fig. 6 to a part of D_i.*

Proof. Every reduction either reduces the size of the derivation or keeps the same size but reduces the total measure of all subderivations ending with $M[U]$. □

Fig. 7 shows how the small-step derivation at the top of Fig. 2 normalizes to the big-step derivation at the bottom there, using the reductions 3, 8, 1, and finally 5 in Fig. 6. Also possible are η-reductions, but we do not need them here.

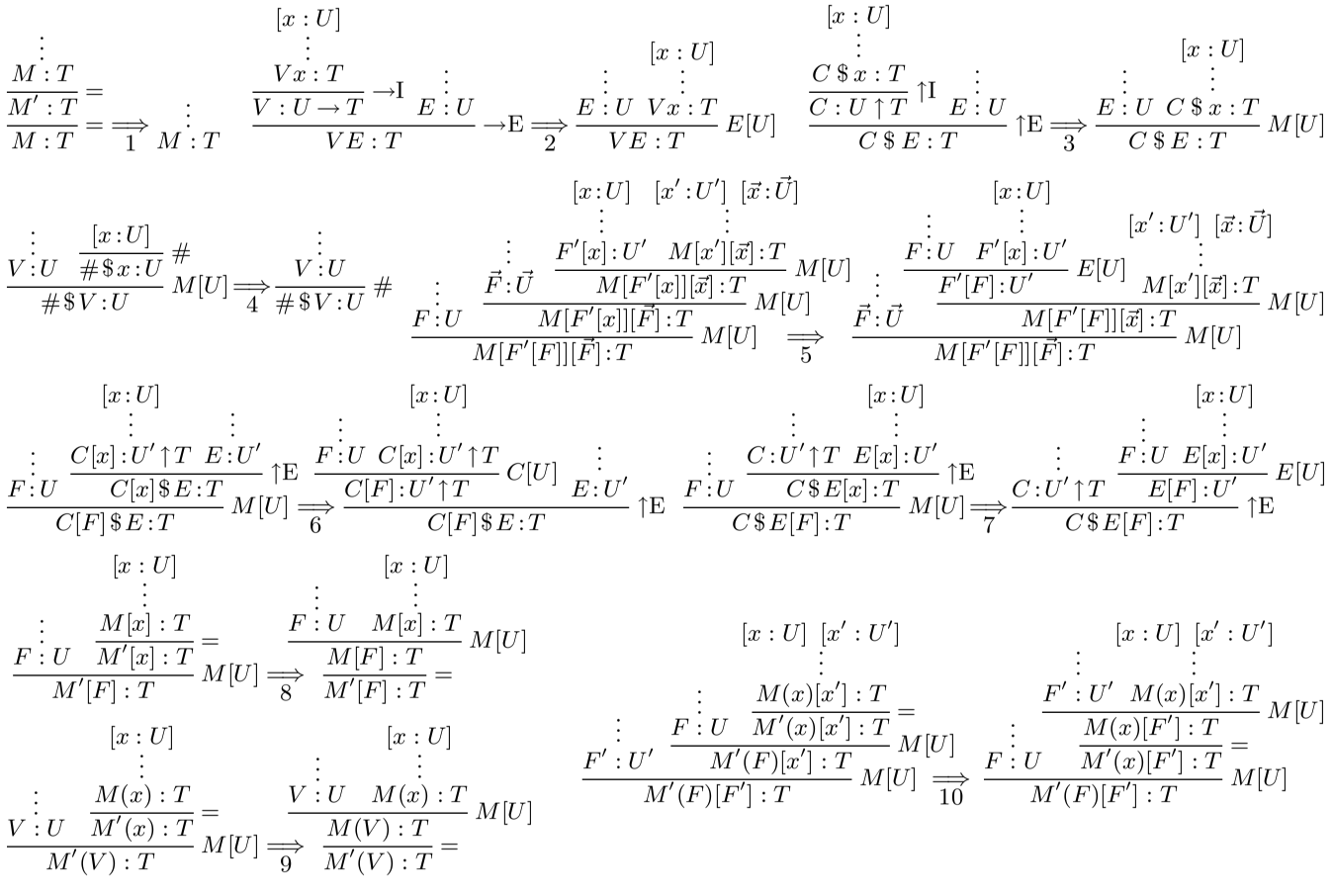

Fig. 6. Reductions on typing derivations. Reductions 2, 5, 6, and 7 use the admissible $E[U]$ and $C[U]$ rules in Lemma 4. In reductions 9 and 10, the notation $M(\,)$ means either $C\,\$\,(\,)E$ or $C;(\,)\,\$\,E$, where $(\,)$ is a special hole to be plugged with a value.

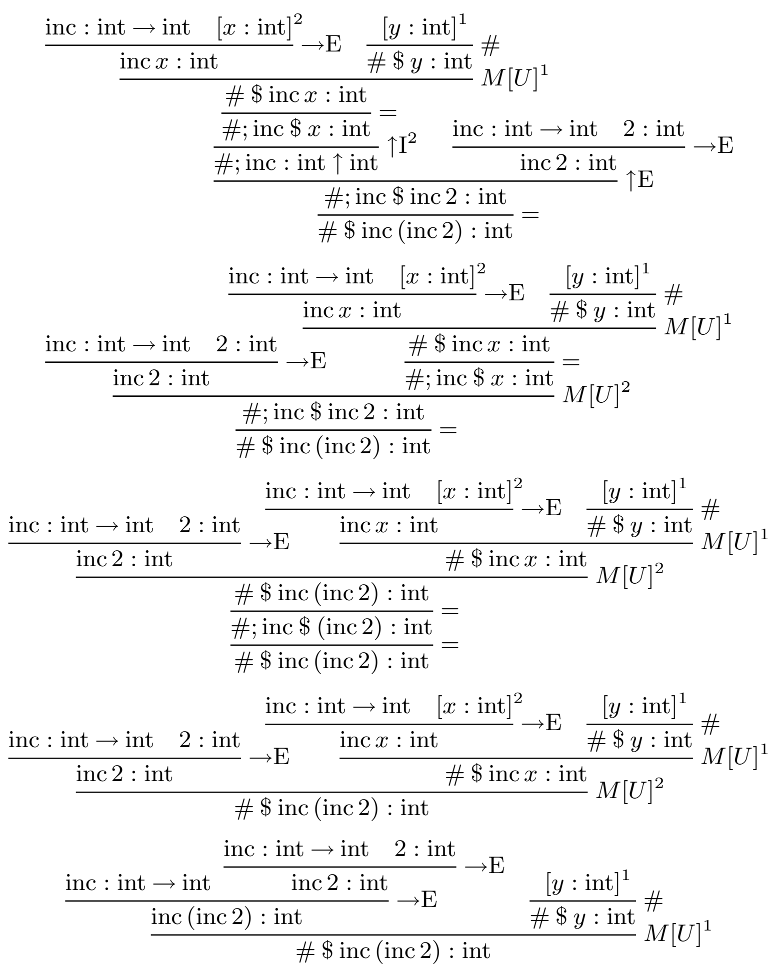

Fig. 7. Reducing a small-step derivation to a big-step derivation

B Example derivations

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[k : U \uparrow T]^1 \quad \dfrac{[f : U' \to U]^2 \quad [x : U']^3}{fx : U}\,{\to}\mathrm{E}}{k \,\$\, fx : T}\,{\uparrow}\mathrm{E}}{k; f \,\$\, x : T} =}{k; f : U' \uparrow T}\,{\uparrow}\mathrm{I}^3 \quad \dfrac{\overline{\text{“x”} : \text{string}}}{(\xi_0 k.\ \text{“x”}) : (U' \uparrow T) \downarrow \text{string}}\,\xi_0}{k; f \,\$\, (\xi_0 k.\ \text{“x”}) : \text{string}}\,{\downarrow}\mathrm{E}}{(\xi_0 k.\ \text{“x”}), k \,\$\, f : \text{string}} =}{(\xi_0 k.\ \text{“x”}), k : (U' \to U) \uparrow \text{string}}\,{\uparrow}\mathrm{I}^2 \quad \dfrac{\overline{1 : \text{int}}}{(\xi_0 k.\ 1) : ((U' \to U) \uparrow \text{string}) \downarrow \text{int}}\,\xi_0}{(\xi_0 k.\ \text{“x”}), k \,\$\, (\xi_0 k.\ 1) : \text{int}}\,{\downarrow}\mathrm{E}}{k \,\$\, (\xi_0 k.\ 1)(\xi_0 k.\ \text{“x”}) : \text{int}} =}{(\xi_0 k.\ 1)(\xi_0 k.\ \text{“x”}) : (U \uparrow T) \downarrow \text{int}}\,{\downarrow}\mathrm{I}^1$$

Fig. 8. Deriving a type for $(\xi_0 k.\ 1)(\xi_0 k.\ \text{“x”})$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[k : \text{int} \uparrow T]^1 \quad \dfrac{[x : \text{int}]^2}{\text{inc}\, x : \text{int}}\,{\uparrow}\mathrm{E}}{k \,\$\, \text{inc}\, x : T}}{k; \text{inc} \,\$\, x : T} =}{k; \text{inc} : \text{int} \uparrow T}\,{\uparrow}\mathrm{I}^2 \quad \dfrac{\dfrac{\overline{\text{“x”} : \text{string}}}{\xi_0 k'.\ \text{“x”} : S \downarrow \text{string}}\,\xi_0}{\xi_0 k.\ \xi_0 k'.\ \text{“x”} : (\text{int} \uparrow T) \downarrow S \downarrow \text{string}}\,\xi_0}{k; \text{inc} \,\$\, (\xi_0 k.\ \xi_0 k'.\ \text{“x”}) : S \downarrow \text{string}}\,{\downarrow}\mathrm{E}}{k \,\$\, \text{inc}(\xi_0 k.\ \xi_0 k'.\ \text{“x”}) : S \downarrow \text{string}} =}{\text{inc}(\xi_0 k.\ \xi_0 k'.\ \text{“x”}) : (\text{int} \uparrow T) \downarrow S \downarrow \text{string}}\,{\downarrow}\mathrm{I}^1$$

Fig. 9. Deriving a type for $\text{inc}(\xi_0 k.\ \xi_0 k'.\ \text{“x”})$