Higher-Order Symb Comput (2009) 22: 233–273
DOI 10.1007/s10990-007-9006-0

A type-theoretic foundation of delimited continuations

Zena M. Ariola · Hugo Herbelin · Amr Sabry

Published online: 24 October 2007

Abstract There is a correspondence between classical logic and programming language calculi with first-class continuations. With the addition of control delimiters, the continuations become composable and the calculi become more expressive. We present a fine-grained analysis of control delimiters and formalise that their addition corresponds to the addition of a single dynamically-scoped variable modelling the special top-level continuation. From a type perspective, the dynamically-scoped variable requires effect annotations. In the presence of control, the dynamically-scoped variable can be interpreted in a purely functional way by applying a store-passing style. At the type level, the effect annotations are mapped within standard classical logic extended with the dual of implication, namely subtraction. A continuation-passing-style transformation of lambda-calculus with control and subtraction is defined. Combining the translations provides a decomposition of standard CPS transformations for delimited continuations. Incidentally, we also give a direct normalisation proof of the simply-typed lambda-calculus with control and subtraction.

Keywords Callcc · Monad · Prompt · Reset · Shift · Subcontinuation · Subtraction

This paper is an extended version of the conference article "A Type-Theoretic Foundation of Continuations and Prompts." (Ariola et al., 2004).
Z.M. Ariola supported by National Science Foundation grant number CCR-0204389.
A. Sabry supported by National Science Foundation grant number CCR-0204389, by a Visiting Researcher position at Microsoft Research, Cambridge, U.K., and by a Visiting Professor position at the University of Genova, Italy.

Z.M. Ariola
University of Oregon, Eugene, OR 97403, USA

H. Herbelin (✉)
INRIA-Futurs, Orsay, France
e-mail: hugo.herbelin@inria.fr

A. Sabry
Indiana University, Bloomington, USA

1 Introduction

Programming practice suggests that control operators add expressive power to purely functional languages. For example, control operators permit the implementation of backtracking [26], coroutines [27], and lightweight processes [52], which go beyond pure functional programming. Of course, any complete program which uses these abstractions can be globally transformed using the continuation-passing style (CPS) transformation to a purely functional program, but this misses the point. As Felleisen [14] formalises and proves, the additional expressiveness of control operators comes from the fact that no *local* transformation of program fragments using control operators is possible.

There is another, simpler, way to formalise the additional expressive power of control operators that is based on the Curry-Howard isomorphism [32]. The pure λ-calculus corresponds to (minimal) intuitionistic logic; extending it with the control operator $\mathcal{C}$ [13] makes it expressive enough to computationally interpret classical logic proofs [23]. Felleisen [14] also showed that the control operator *callcc* is less expressive than $\mathcal{C}$. The logical counterpart is that *callcc* is enough to computationally express proofs of *minimal classical logic* where it is possible to prove Peirce's law but not double negation elimination, whereas $\mathcal{C}$ corresponds to (non-minimal) classical logic which proves double negation elimination (see Ariola and Herbelin, [1]).

So far, only a small family of control operators have been shown to be connected to proof constructions in logic, and the situation for the other control operators is much less understood. In particular, the addition of control delimiters [13] gives rise to delimited continuations which are more expressive than their undelimited counterparts [47]. Even for the well-studied operators for delimited continuations *shift* and *reset* [10], it is not clear how to formalise this additional expressiveness for at least three reasons:

1. Several type systems and type-and-effect systems have been proposed for these operators [9, 33, 34, 40], and it is not apparent whether one of these systems is the "right" one even when moving to the worlds of CPS or monads [51]. Furthermore, for the type-and-effect systems, there are several possible interpretations of the effect annotations, which should be completely eliminated anyway to get a correspondence with a standard logic.
2. Although the semantics of many control operators (including *shift* and *reset* [35]) can be given using local axioms, a few properties of these systems, when seen as rewriting systems, have been investigated. Especially, many of the systems are not confluent, and under none of the type-and-effect systems is a property such as strong normalisation known to be ensured.
3. Delimited continuations can simulate a large number of other computational effects like state and exceptions [19]. This seems to indicate that the understanding of the expressive power of such control operators must also include an understanding of the expressive power of other effects. Riecke and Thielecke [44] and Thielecke [48, 49] have formalised the expressive power of various combinations of continuations, exceptions and state but their results are not expressed using standard type systems and logics.

We show that the additional expressiveness of control delimiters can be characterised in terms of a *dynamically-scoped continuation variable*, and that in the presence of dynamic binding, continuations can model other effects like state and exceptions.

Environment-passing transformation is the standard way to interpret dynamic binding in a functional way. In the presence of control, it turns out that a store-passing transformation is needed. This is because the dynamically-scoped continuation variable can be passed back to the contexts surrounding the binding locations.

At the level of types, if a continuation in the store is assigned the type $\neg T$, then a store-passing transformation translates functional types of the form $A \to B$ into types of the form $A \wedge \neg T \to B \wedge \neg T$. Interestingly, we have the following equivalences that hold *classically* but not intuitionistically:

$$A \wedge \neg T \to B \quad \text{iff} \quad A \to B \vee T \quad \text{iff} \quad A \wedge \neg T \to B \wedge \neg T.$$

This informally complies with the fact that in the presence of control delimiters, all of the following particular cases are definable: functions which refer to a dynamic variable of type $\neg T$ without relying on control, functions which throw an exception of type T, functions which manipulate a state variable of type $\neg T$.

Felleisen's approach of the theory of control [13] relies on a reification of evaluation contexts into continuations. Contrastingly, our approach, inspired by Ariola and Herbelin [1], itself inspired by Parigot [42], directly deals with contexts (or more precisely with elementary pieces of context).

Passing around a store which is a context requires to be able to manipulate the pair of a term and of a context. From this perspective, the connective of logical subtraction $A - T$ [6, 7, 43], which is classically equivalent to $A \wedge \neg T$, naturally arises. This connective will be used to interpret the effect annotations.

The next section reviews some background material on abortive continuations. We present Felleisen and Hieb [15] revised theory of control and the $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus, an extended syntactic variant of call-by-value $\lambda\mu$-calculus [41, 42] from previous work by the first two authors [1]. Section 3 explains the semantics of delimited continuations by generalising the $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus to include just one dynamically-scoped variable. Section 4 investigates various ways to extend the type system of $\lambda_{\mathcal{C}\mathsf{tp}}$ to accommodate the dynamically-scoped variable. We present three systems (closely related to existing type-and-effect systems) and reason about their properties. Sections 5 and 6 explain how to interpret the effect annotations in a standard logic. The first focuses on motivating the dual connective of implication, namely subtraction, and formally defining $\lambda^{-}_{\mathcal{C}}$, a calculus with control operators and subtraction, but no dynamic binding. The second focuses on using the subtractive type for explaining the action on the effect annotations of a store-passing style transformation when what is stored is an evaluation context. Section 7 develops a CPS transformation for $\lambda^{-}_{\mathcal{C}}$ that ultimately shows that known CPS transformations for delimited continuations are decomposable into a store-passing style transformation into $\lambda^{-}_{\mathcal{C}}$ followed by a standard CPS transformation into the pure λ-calculus. The first transformation takes care of the control delimiter; the second transformation takes care of the control operator. Section 8 concludes with a discussion of some of the other control operators. Also discussed is the Curry-Howard problem for control delimiters. Finally, Appendix gives the details of the proof of strong normalisation for simply-typed $\lambda^{-}_{\mathcal{C}}$.

2 Abortive continuations and classical logic

We review the semantics of abortive continuations and their connection to classical logic. We start with Felleisen and Hieb [15] revised theory of control.

2.1 The $\lambda_{\mathcal{C}}$-calculus: Felleisen and Hieb revised theory of control

Figure 1 introduces the syntax of a call-by-value calculus extended with the unary operator $\mathcal{C}$. Variables and lambda-abstractions are called values. There is no distinction between

Fig. 1 Syntax of $\lambda_{\mathcal{C}}$

$$
\begin{array}{lll}
x, a, v, f, c, k \in \mathit{Vars} & & \\
M, N \in \mathit{Terms} & ::= x \mid \lambda x.\ M \mid M\ N \mid \mathcal{C}\ M & \\
V \in \mathit{Values} & ::= x \mid \lambda x.\ M &
\end{array}
$$

Fig. 2 Reduction rules of call-by-value $\lambda_{\mathcal{C}}$ (Felleisen and Hieb)

$$
\begin{array}{lll}
\beta_v: & (\lambda x.\ M)\ V & \to M\ [V/x] \\
\mathcal{C}_L: & (\mathcal{C}M)\ N & \to \mathcal{C}\ (\lambda c.\ M\ (\lambda f.\ \mathcal{A}\ (c\ (f\ N)))) \\
\mathcal{C}_R: & V\ (\mathcal{C}M) & \to \mathcal{C}\ (\lambda c.\ M\ (\lambda x.\ \mathcal{A}\ (c\ (V\ x)))) \\
\mathcal{C}_{idem}: & \mathcal{C}\ (\lambda c.\ \mathcal{C}M) & \to \mathcal{C}\ (\lambda c.\ M\ (\lambda x.\ \mathcal{A}\ x)) \\
\mathcal{C}_{top}: & \mathcal{C}M & \to \mathcal{C}\ (\lambda c.\ M\ (\lambda x.\ \mathcal{A}\ (c\ x)))
\end{array}
$$

Fig. 3 Syntax of $\lambda_{\mathcal{C}\mathsf{tp}}$

$$
\begin{array}{rcl}
x, a, v, f \in \mathit{Vars} & & \\
k \in \mathit{KVars} & & \\
\mathit{KConsts} & = & \{\ \mathsf{tp}\ \} \\
M, N \in \mathit{Terms} & ::= & x \mid \lambda x.\ M \mid M\ N \mid \mathcal{C}(\lambda k.\ J) \\
V \in \mathit{Values} & ::= & x \mid \lambda x.\ M \\
J \in \mathit{Jumps} & ::= & k\ M \mid \mathsf{tp}\ M
\end{array}
$$

continuations and variables, and a continuation does not need to be applied. The reduction semantics is given in Fig. 2. Our previous work [1] introduced an alternative theory of control called $\lambda_{\mathcal{C}\mathsf{tp}}$ (For a thorough comparison of the two systems see [2].)

2.2 The $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus: syntax and dynamic semantics

The syntax of the call-by-value $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus is in Fig. 3. It is a call-by-value variant of Parigot's $\lambda\mu$-calculus [42] where μ is written $\mathcal{C}$. Ariola and Herbelin [1] showed that a top-level continuation constant has to be added to the $\lambda\mu$-calculus to recover the expressiveness of the control calculus of Felleisen and Hieb [15]. The resulting calculus has the following properties:

- Every use of the control operator $\mathcal{C}$ must be syntactically applied to a procedure which receives the captured continuation. For example, whereas $\lambda y.\ \mathcal{C}\ y$ is a legal $\lambda_{\mathcal{C}}$ term, it is not a legal $\lambda_{\mathcal{C}\mathsf{tp}}$ term;
- Variables bound to continuations are distinct from other variables and can only occur in application position. For example, whereas $\mathcal{C}(\lambda k.\ k)$ or $\mathcal{C}(\lambda k.\ \lambda x.\ k)$ are legal $\lambda_{\mathcal{C}}$ terms, they are not legal $\lambda_{\mathcal{C}\mathsf{tp}}$ terms;
- Applications of continuations are called *jumps* and must occur exactly when a continuation is also captured. For example, whereas $\mathcal{C}(\lambda k.\ (k\ x)\ y)$ is a legal $\lambda_{\mathcal{C}}$ term, it is not a legal $\lambda_{\mathcal{C}\mathsf{tp}}$ term;
- There is a continuation constant tp which denotes the top-level continuation.

We sometimes use the following convenient abbreviations:

$$\mathcal{A}\ M \triangleq \mathcal{C}(\lambda\_.\mathsf{tp}\ M) \qquad \text{(Abbrev. 1)}$$

$$\mathcal{T}h\ k\ M \triangleq \mathcal{C}(\lambda\_.k\ M) \qquad \text{(Abbrev. 2)}$$

In both abbreviations, the variable _ is an anonymous continuation variable that does not occur in the body of the abstraction. Thus, the first abbreviation corresponds to jumping to

Fig. 4 Translation of $\lambda_{\mathcal{C}}$ in $\lambda_{\mathcal{C}_{\mathsf{tp}}}$

$$
\begin{aligned}
(x)^\circ &= x \\
(\lambda x.\ M)^\circ &= \lambda x.\ M^\circ \\
(M\ N)^\circ &= M^\circ\ N^\circ \\
(\mathcal{C} M)^\circ &= \mathcal{C}(\lambda k.\mathsf{tp}\ (M^\circ\ (\lambda x.\ \mathcal{T}h\ k\ x)))
\end{aligned}
$$

Fig. 5 Reductions of call-by-value $\lambda_{\mathcal{C}_{\mathsf{tp}}}$

$$
\begin{array}{lll}
\beta_v: & (\lambda x.M)\ V & \to M\ [V/x] \\
\mathcal{C}_L: & \mathcal{C}(\lambda k.\ J)\ N & \to \mathcal{C}(\lambda k.\ J\ [k\ (\Box\ N)/k]) \\
\mathcal{C}_R: & V\ \mathcal{C}(\lambda k.\ J) & \to \mathcal{C}(\lambda k.\ J\ [k\ (V\ \Box)/k]) \\
\mathcal{C}_{elim}: & \mathcal{C}(\lambda k.k\ M) & \to M \qquad \text{where } k \notin FV(M) \\
 & & \\
\mathcal{C}_{idem}: & \mathsf{tp}\ \mathcal{C}(\lambda k.\ J) & \to J\ [\mathsf{tp}\ \Box/k] \\
\mathcal{C}_{idem'}: & k'\ \mathcal{C}(\lambda k.\ J) & \to J\ [k'\ \Box/k]
\end{array}
$$

the top-level continuation. The second abbreviation corresponds to jumping to a previously-captured continuation k.

The translation of $\lambda_{\mathcal{C}}$ terms into the $\lambda_{\mathcal{C}_{\mathsf{tp}}}$-calculus is given in Fig. 4. For example, $\lambda y.\mathcal{C} y$ becomes $\lambda y.\mathcal{C}(\lambda k.\mathsf{tp}\ (y\ (\lambda x.\mathcal{T}h\ k\ x)))$. Notice how the captured continuation is given a name k; the implicit jump to the top-level associated with the continuation capture is made explicit; and the implicit aborting of the context when k is applied is also made explicit.

The semantics is given by the local reduction rules in Fig. 5. The first four reduction rules are similar to the ones given by Felleisen and Hieb [15] for $\lambda_{\mathcal{C}}$ but with important differences discussed below:

- Like the original calculus, the rules $\mathcal{C}_L$ and $\mathcal{C}_R$ allow one to *lift* the control operation step-by-step until it reaches a point where it can no longer be lifted. When the control operator reaches a jump to the top-level (rule $\mathcal{C}_{idem}$), the continuation captured is the trivial continuation modelled by tp.
- Unlike $\lambda_{\mathcal{C}}$, the reduction rules of Fig. 5 use *structural substitutions* for continuations. The general form of structural substitution relies on the definition of contexts:

$$E ::= \Box \mid E\ M \mid V\ E\,,$$

  and on the notation $E[N]$ which stands for the replacement of the hole $\Box$ of E with N. The structural substitution $M[q\ E/k]$ (resp. $J[q\ E/k]$) reads as: "replace every jump of the form $k\ N$ in M (resp. J) by the jump $(q\ E[N])$ (and recursively in N)." The substitutions $M[\mathsf{tp}\ E/k]$ and $J[\mathsf{tp}\ E/k]$ are defined similarly. For example,

$$\mathsf{tp}\ \mathcal{C}(\lambda k.k\ (M\,\mathcal{C}(\lambda k'.k\ N))) \to \mathsf{tp}\ (M\,\mathcal{C}(\lambda k'.\mathsf{tp}\ N))$$

  assuming that k does not occur free in M and N. What is distinctive about this structural substitution is that it is performed *irrespective of whether the jump argument is a value or not.* In particular, in the above example, the application of M is clearly not a value and N need not be a value. In contrast, $\lambda_{\mathcal{C}}$ requires the argument to every jump to be reduced to a value *before* performing the jump.
- Unlike the original presentation, the abortive behaviour of $\mathcal{C}$ is not hard-wired in the reduction rules but in the syntax. Each call $k\ M$ to a continuation k bound by some $\mathcal{C}$ is necessarily surrounded by some other $\mathcal{C}(\lambda k'.\Box)$ making it abortive. Contrastingly, the original reduction rules dynamically insert the context $\mathcal{C}(\lambda\_.\Box)$ around each reference to

k (an abortive operator which is useless, as shown by the original $\mathcal{C}_{idem}$, when k is already surrounded by some $\mathcal{C}$).

Rule $\mathcal{C}_{idem'}$ is a generalisation of $\mathcal{C}_{idem}$ which, apart from the notion of structural substitution, is similar to the rule $M\ (\mathcal{C}\ N) \to N\ (\lambda x.(M\ x))$ where M has type $\neg A$, proposed by Barbanera and Berardi [5]. In terms of the $\lambda\mu$-calculus, both $\mathcal{C}_{idem'}$ and $\mathcal{C}_{idem}$ correspond to Parigot's "renaming" rule [42]. The $\mathcal{C}_{elim}$ rule, which is also used by Hofmann [31] for completeness reasons, was also proposed by Felleisen and Hieb [15] since it leads to a better simulation of evaluation. However, unlike our system, Felleisen and Hieb extended reduction theory is non-confluent.

The difference between $\lambda_{\mathcal{C}}$ and $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ is not observable with respect to reduction to values [2]:

Proposition 1 *Let M be a closed $\lambda_{\mathcal{C}}$ term*:

$$\mathcal{A}\ M \twoheadrightarrow_{\lambda_{\mathcal{C}}} \mathcal{A}\ V \quad \textit{iff} \quad \mathsf{tp}\ M^{\circ} \twoheadrightarrow_{\lambda_{\mathcal{C}_{\mathsf{tp}}}} \mathsf{tp}\ V'.$$

However, it is significant as the use of structural substitution avoids introducing useless and artificial β-expansions which are not even β_v-expansions. In contrast, the β-expansions in the original calculus force both the evaluation of the argument to the jump in some continuation and then the erasure of this continuation, instead of the equivalent but more natural (and efficient) choice of first erasing the continuation and then evaluating the argument to the jump.

2.3 The $\lambda_{\mathcal{C}_{\mathsf{tp}}}$-calculus: static semantics

The calculus $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ was originally motivated by the Curry-Howard correspondence between classical logic and control operators. The calculus is both a clarification of the connection between the control operator $\mathcal{C}$ and classical logic [23], and a counterpart to Parigot's $\lambda\mu$-calculus for a formulation of classical logic based on the classical rule *Reductio Ad Absurdum* (see below).

According to the Curry-Howard correspondence, the natural interpretation of the tp continuation constant is as a constructor for the elimination rule of the "false" connective $\bot$, i.e., as a primitive continuation expecting an argument of type $\bot$. In the current approach, we generalise the use of tp. We assign it the role of the top-level continuation for a top-level that can have any fixed but arbitrary type and not only the type $\bot$ as in the characterisation of (non-minimal) classical logic.

The resulting system of simple types for $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ is given in Fig. 6. The set of base types in $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ could include $\bot$ to represent an empty type or the proposition "false" but that type plays no special role in the judgements. The judgements do however refer to a special type $\perp\!\!\!\perp$ which is used as the type of jumps, i.e., the type of expressions in the syntactic category J. The type $\perp\!\!\!\perp$ is not a base type; it can never occur as the conclusion of any judgement for terms in the syntactic category M, but it may occur in the context Γ as the return type of a continuation variable. We sometimes abbreviate $T \to \perp\!\!\!\perp$ as $\neg_{\perp\!\!\!\perp} T$.

The judgements are of two forms: $\Gamma \vdash M : A; T$ for typing terms and $\Gamma \vdash J :\perp\!\!\!\perp; T$ for typing jumps. In both cases the type T is the arbitrary but fixed type for the top-level; it acts as a global parameter to the type system [40]. The rule *RAA* (*Reductio Ad Absurdum*) is similar to the double-negation rule in Griffin's system [23] except that it uses the special type $\perp\!\!\!\perp$ instead of $\bot$. According to the $\to_e^{\mathsf{tp}}$ rule, the special top-level continuation can only be invoked with a term of the distinguished type T. According to the $\to_e^{k}$ rule, continuation

$$b \in \mathit{BaseType}$$
$$A, B, T \in \mathit{TypeExp} ::= b \mid A \to B$$
$$\Gamma \in \mathit{Contexts} ::= \cdot \mid \Gamma, x : A \mid \Gamma, k : A \to \perp\!\!\!\perp$$

$$\frac{}{\Gamma, x : A \vdash x : A; T}\ Ax$$

$$\frac{\Gamma, x : A \vdash M : B; T}{\Gamma \vdash \lambda x.M : A \to B; T} \to_i \qquad \frac{\Gamma \vdash M : A \to B; T \quad \Gamma \vdash M' : A; T}{\Gamma \vdash M M' : B; T} \to_e$$

$$\frac{\Gamma, k : A \to \perp\!\!\!\perp \vdash J : \perp\!\!\!\perp; T}{\Gamma \vdash \mathcal{C}(\lambda k.J) : A; T}\ RAA$$

$$\frac{\Gamma, k : A \to \perp\!\!\!\perp \vdash M : A; T}{\Gamma, k : A \to \perp\!\!\!\perp \vdash k\ M : \perp\!\!\!\perp; T} \to_e^k \qquad \frac{\Gamma \vdash M : T; T}{\Gamma \vdash \mathsf{tp}\ M : \perp\!\!\!\perp; T} \to_e^{\mathsf{tp}}$$

Fig. 6 Type system of $\lambda_{\mathcal{C}_{\mathsf{tp}}}$

variables are invoked with a term of the type expected by the context in which they were captured.

It is possible to inject a continuation of type $T \to \perp\!\!\!\perp$ into the type of *functions* $T \to \perp$:

$$\frac{\dfrac{\dfrac{\dfrac{}{\Gamma, k : T \to \perp\!\!\!\perp, x : T \vdash x : T; T}}{\Gamma, k : T \to \perp\!\!\!\perp, x : T \vdash k\ x : \perp\!\!\!\perp; T}}{\Gamma, k : T \to \perp\!\!\!\perp, x : T \vdash \mathcal{C}(\lambda\_.k\ x) : \perp; T}}{\Gamma, k : T \to \perp\!\!\!\perp \vdash \lambda x.\mathcal{C}(\lambda\_.k\ x) : T \to \perp; T}$$

Thus, it does no harm to informally think of $\perp\!\!\!\perp$ as $\perp$ remembering that an explicit coercion is required to move from one to the other (the way to coerce $T \to \perp$ into $T \to \perp\!\!\!\perp$ is given in Intermezzo 15). But the special nature of the type $\perp\!\!\!\perp$ can perhaps be best understood by examining the situation in the isomorphic $\lambda\mu$-calculus extended with tp. In that type system, there is no need for $\perp\!\!\!\perp$: the types of continuation variables are maintained on the right-hand side of the sequent and jumps have no type. In other words, a more accurate understanding of $\perp\!\!\!\perp$ is as a special symbol denoting "no type."

Simple experience with the type system reveals that the top-level type sometimes provides helpful additional information about the type of a term. For example:

$$\vdash \mathcal{A}\ 5\ ==\ \texttt{"Hello"} : \mathsf{bool};\ \mathsf{int}$$

accurately predicts that the expression returns a bool or jumps to the top-level with an int. However, the top-level type sometimes provides misleading information about the type of a term. For example, in the judgement:

$$\vdash \lambda x.(x\ +\ \mathcal{A}\ \texttt{"Hello"}) : \mathsf{int} \to \mathsf{int};\ \mathsf{string}$$

the presence of the type string is actually associated with the *definition* of the function and not its use. This confusion could lead to problems as discussed in Sects. 4.1 and 4.2 where we talk about generalisations of the system.

Proposition 2 (i) $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ *is confluent*. (ii) *Subject reduction*: *Given* $\lambda_{\mathcal{C}_{\mathsf{tp}}}$ *terms* M *and* N, *if* $\Gamma \vdash M : A; T$ *and* $M \to N$, *then* $\Gamma \vdash N : A; T$.

(ii) *Progress*: *A closed* $\lambda_{\mathcal{C}\mathsf{tp}}$ *jump* J *is either a jump to the top-level with a value* (*i.e. a jump* $\mathsf{tp}\ V$) *or a head-reducible jump* (*i.e. either of the form* $\mathsf{tp}\ E[N]$ *with* N *a redex or* $\mathsf{tp}\ \mathcal{C}(\lambda k.\, J)$ *which is a redex*).

(ii) *Simply typed terms are strongly normalising*.

Proof Confluence is shown in [2] where it relies on the standard parallel reduction method, using extra tricks from [4]. Subject reduction and progress are routine verifications. Strong normalisation is somehow standard but it can also be obtained as a consequence of Proposition 16, interpreting tp as a free continuation variable. □

3 Delimited continuations

We modify the $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus by adding control delimiters. We then show that the delimiter can be eliminated in favour of a single dynamically-scoped variable.

3.1 Adding control delimiters

As shown in Fig. 7, we extend the syntax of the $\lambda_{\mathcal{C}\mathsf{tp}}$ calculus with a control delimiter #. The use of # delimits the continuation captured by $\mathcal{C}$. For example, the occurrence of $\mathcal{C}$ in:

$$1\ +\ (\#\ (\mathsf{tp}\ (2\ +\ (\mathcal{C}(\lambda k \ldots)))))$$

only captures the continuation corresponding to the context $(2\ +\ \Box)$. Similarly, in the following expression, the use of # delimits the jump:

$$1\ +\ (\#\ (\mathsf{tp}\ 2)).$$

The jump to the top-level is blocked by the #, and the entire expression should evaluate to 3. This intuition is consistent with the original motivation for introducing control delimiters: they provide an explicit "top-level" for the evaluation of a subexpression [13].

The semantics is formalised by adding one new reduction to the reductions of $\lambda_{\mathcal{C}\mathsf{tp}}$ as shown in Fig. 8. The reduction $\#_v$ formalises that the role of the # terminates once its subexpression has been simplified to a value, which by definition can no longer perform any control actions.

Fig. 7 Syntax of $\lambda_{\mathcal{C}\#\mathsf{tp}}$

$$\begin{array}{rcl}
x, a, v, f & \in & \mathit{Vars} \\
k & \in & \mathit{KVars} \\
\mathit{KConsts} & = & \{\ \mathsf{tp}\ \} \\
M, N & \in & \mathit{Terms} ::= x \mid \lambda x.\, M \mid M\ N \mid \mathcal{C}(\lambda k.\, J) \mid \#\ J \\
V & \in & \mathit{Values} ::= x \mid \lambda x.\, M \\
J & \in & \mathit{Jumps} ::= k\ M \mid \mathsf{tp}\ M
\end{array}$$

Fig. 8 Reductions of call-by-value $\lambda_{\mathcal{C}\#\mathsf{tp}}$

$$\begin{array}{lll}
\beta_v : & (\lambda x.M)\ V & \to M\ [V/x] \\
\mathcal{C}_L : & \mathcal{C}(\lambda k.\, J)\ N & \to \mathcal{C}(\lambda k.\, J\ [k\ (\Box\ N)/k]) \\
\mathcal{C}_R : & V\ (\mathcal{C}(\lambda k.\, J)) & \to \mathcal{C}(\lambda k.\, J\ [k\ (V\ \Box)/k]) \\
\mathcal{C}_{idem} : & \mathsf{tp}\ \mathcal{C}(\lambda k.\, J) & \to J\ [\mathsf{tp}\ \Box/k] \\
\mathcal{C}_{idem'} : & k'\ \mathcal{C}(\lambda k.\, J) & \to J\ [k'\ \Box/k] \\
\mathcal{C}_{elim} : & \mathcal{C}(\lambda k.\, k\ M) & \to M \qquad \text{where } k \notin FV(M) \\
\#_v : & \#\ (\mathsf{tp}\ V) & \to V
\end{array}$$

Remark 3 The presence of the delimiter # makes it possible to express the operators *shift* and *reset* given in the seminal paper of Danvy and Filinski [9]. Filinski [19] showed that both operators are definable from Felleisen's $\mathcal{C}$ and # operators. Expressed in $\lambda_{\mathcal{C}\#\mathsf{tp}}$, we get:

$$\begin{aligned}\mathcal{S}\ (\lambda q.\ M) &= \mathcal{C}(\lambda k.\mathsf{tp}\ (\lambda q.\ M)\ (\lambda x.\#\ (k\ x))),\\ \langle M\rangle &= \#\ (\mathsf{tp}\ M).\end{aligned}$$

The essence of the above definition is to surround each invocation of a continuation by the delimiter. This however does not mean that each jump in $\lambda_{\mathcal{C}\#\mathsf{tp}}$ must be surrounded by the delimiter. For example, one can write # (tp ($\mathcal{C}(\lambda k.$ tp $((\lambda x.\ (\mathcal{T}h\ k\ x) + 10)\ (\#\ (k\ 3)))) + 5)$), which evaluates to 8. This does not mean either that any term that behaves like $\mathcal{S}\ (\lambda q.\ M)$ must have the form above. For example, $\mathcal{C}(\lambda k.$ tp $((\lambda x.\ (\#\ (k\ x)) + 10)\ (\#\ (k\ 3))))$ behaves the same as $\mathcal{S}\ (\lambda q.\ ((\lambda x.\ q\ x + 10)\ (q\ 3)))$.

3.2 Dynamic binding of the top-level continuation

The discussion and semantics in the previous section explain that the addition of control delimiters amounts to having different bindings to the top-level continuation. To better understand this idea, we propose to modify the original $\lambda_{\mathcal{C}\mathsf{tp}}$ which has a *constant* corresponding to the top-level continuation by making the top-level continuation a *variable*. In this section, we explain why this variable cannot be a statically-scoped variable.

Example 4 (Static vs. Dynamic #) Using the $\lambda_{\mathcal{C}\#\mathsf{tp}}$ reductions, we have (we underline the redex in the first two steps):

$$\begin{aligned}&\#\ (\mathsf{tp}\ ((\underline{\#\ (\mathsf{tp}\ (\lambda\_.\ \mathcal{C}(\lambda\_.\mathsf{tp}\ (\lambda\_.\ 3))))})\ (\lambda\_.\ \mathcal{C}(\lambda\_.\mathsf{tp}\ 4))))\\ \rightarrow\ &\#\ (\mathsf{tp}\ (\underline{(\lambda\_.\ \mathcal{C}(\lambda\_.\mathsf{tp}\ (\lambda\_.\ 3)))\ (\lambda\_.\ \mathcal{C}(\lambda\_.\mathsf{tp}\ 4))}))\\ \rightarrow\ &\#\ (\mathsf{tp}\ \mathcal{C}(\lambda\_.\mathsf{tp}\ (\lambda\_.\ 3)))\\ \rightarrow\ &\#\ (\mathsf{tp}\ (\lambda\_.\ 3))\\ \rightarrow\ &(\lambda\_.\ 3)\end{aligned}$$

In the original term, the third occurrence of tp is statically associated with the second occurrence of #. However, after one reduction step this occurrence of tp is actually associated with the first occurrence of #.

Intuitively, a use of a control delimiter introduces a new binding for the top-level continuation *for the duration of the evaluation of its subexpression*. In other words, the standard semantics of control delimiters requires that the binding for the top-level continuation be *dynamic* (as we will formally prove in the remainder of the paper).

It is possible to use static scope for control delimiters. Not surprisingly, this choice gives a different semantics. Indeed, Thielecke [50] considers several variations of control operators with different scope rules. He presents a system with static scope and a system with dynamic scope and shows that the system with static scope (adequately extended with infinitely many continuation variables so that capture-free substitution and β-reduction are correctly defined) corresponds to classical logic and that the system with dynamic scope corresponds to intuitionistic logic. Kameyama [34] also develops a type system and a semantics for a static variant of control delimiters which he shows has a different semantics than the usual operators.

Fig. 9 Syntax of $\lambda_{\mathcal{C}\widehat{tp}}$

$$
\begin{array}{l}
x, a, v, f, c \in \mathit{Vars} \\
k \in \mathit{StaticKVars} \\
\mathit{DynKVars} = \{\ \widehat{tp}\ \} \\
M, N \in \mathit{Terms} ::= x \mid \lambda x.\, M \mid M\ N \mid \mathcal{C}(\lambda k.\, J) \mid \mathcal{C}(\lambda \widehat{tp}.\, J) \\
\quad V \in \mathit{Values} ::= x \mid \lambda x.\, M \\
\quad J \in \mathit{Jumps} ::= k\ M \mid \widehat{tp}\ M
\end{array}
$$

3.3 The $\lambda_{\mathcal{C}\widehat{tp}}$-calculus

As motivated in the previous section the control delimiter can be modelled in an extension of the $\lambda_{\mathcal{C}_{\mathsf{tp}}}$-calculus where the top-level continuation is a dynamically-scoped variable. This extension is given in Fig. 9. The grammar now has two distinct uses of $\mathcal{C}$: one for regular statically-scoped continuation variables and one for the unique dynamically-scoped continuation variable $\widehat{tp}$. We introduce a new abbreviation for an expression which aborts to the closest occurrence of this dynamically-scoped continuation variable:

$$\widehat{\mathcal{A}}\ M \triangleq \mathcal{C}(\lambda\_.\ \widehat{tp}\ M) \qquad \text{(Abbrev. 3)}$$

The anonymous variable _ ranges only over regular variables and static continuation variables but not over the dynamic variable. Similarly, the notions of free and bound variables only apply to the regular variables and the static continuation variables but not the dynamic variable.

In order to define the semantics of $\lambda_{\mathcal{C}\widehat{tp}}$ we first need an understanding of the semantics of dynamic binding in the absence of control operators [38]. A dynamic abstraction $(\lambda\widehat{x}.M)$ is generally like a regular function in the sense that when it is called with a value V, the formal parameter $\widehat{x}$ is bound to V. But:

DS1 The association between $\widehat{x}$ and V established when a function is called lasts *exactly* as long as the evaluation of the body of the function. In particular, the association is disregarded when the function returns, and this happens *even* if the function returns something like $(\lambda\_.\ldots\widehat{x}\ldots)$ which contains an occurrence of $\widehat{x}$: no closure is built and the occurrence of $\widehat{x}$ in the return value is allowed to escape.

DS2 The association between $\widehat{x}$ and V introduced by one function may capture occurrences of $\widehat{x}$ that escape from other functions. For example, if we have two dynamic functions f and g with f directly or indirectly calling g, then during the evaluation of the body of f, occurrences of $\widehat{x}$ that are returned in the result of g will be captured. Turning this example around, the occurrences of $\widehat{x}$ that escape from g are bound by the closest association found up the dynamic chain of calls.

In the presence of control operators, the situation is more complicated because the evaluation of the body of a function may abort or throw to a continuation instead of returning; and capturing a continuation inside the body of a function allows one to re-enter (and hence re-evaluate) the body of the function more than once.

Despite the additional complications, the reduction rules of $\lambda_{\mathcal{C}\widehat{tp}}$ in Fig. 10 look essentially like the reduction rules of the $\lambda_{\mathcal{C}_{\mathsf{tp}}}$-calculus. The noteworthy addition is the rule $\mathcal{C}_{elim'}$ which allows occurrences of $\widehat{tp}$ in V to escape as suggested by DS1 above. Another change, implied by DS2, is that the meta-operation of substitution must allow for the capture of $\widehat{tp}$. The substitution operations $M\ [V/x]$ and $J\ [k\ E/k]$ do not rename $\widehat{tp}$ but are otherwise standard.

Fig. 10 Reductions of call-by-value $\lambda_{\mathcal{C}\widehat{tp}}$

$$
\begin{array}{lll}
\beta_v: & (\lambda x.\, M)\ V & \to M\ [V/x] \\
\mathcal{C}_L: & \mathcal{C}(\lambda k.\, J)\ N & \to \mathcal{C}(\lambda k.\, J\ [k\ (\Box\ N)/k]) \\
\mathcal{C}_R: & V\ \mathcal{C}(\lambda k.\, J) & \to \mathcal{C}(\lambda k.\, J\ [k\ (V\ \Box)/k]) \\
\mathcal{C}_{idem}: & \widehat{tp}\ \mathcal{C}(\lambda k.\, J) & \to J\ [\widehat{tp}\ \Box/k] \\
\mathcal{C}_{idem'}: & k'\ \mathcal{C}(\lambda k.\, J) & \to J\ [k'\ \Box/k] \\
\mathcal{C}_{elim}: & \mathcal{C}(\lambda k.\, k\ M) & \to M \qquad \text{where } k \notin FV(M) \\
\mathcal{C}_{elim'}: & \mathcal{C}(\lambda \widehat{tp}.\, \widehat{tp}\ V) & \to V \qquad \text{even if } \widehat{tp} \text{ occurs in } V
\end{array}
$$

In particular, we have that:

$$\mathcal{C}(\lambda \widehat{tp}.\ J)[M/x] \equiv \mathcal{C}(\lambda \widehat{tp}.\ J[M/x])$$

even if $\widehat{tp}$ occurs in M.

Example 5 (Dynamic capture) The reduction:

$$
\begin{array}{l}
(\lambda x.\, \lambda y.\, \mathcal{C}(\lambda \widehat{tp}.\, \widehat{tp}\ (x\ y)))\ (\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y)) \\
\quad \to_{\beta_v} (\lambda y.\, \mathcal{C}(\lambda \widehat{tp}.\, \widehat{tp}\ (x\ y)))\ [\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y)/x] \\
\quad \equiv_{\alpha} \lambda y'.\, \mathcal{C}(\lambda \widehat{tp}.\, \widehat{tp}\ ((\lambda\_.\mathcal{C}(\lambda\_.\ \widehat{tp}\ y))\ y'))
\end{array}
$$

captures $\widehat{tp}$ but not y.

The exact correspondence between the syntax and reductions of $\lambda_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda_{\mathcal{C}\widehat{tp}}$ allows to express the following:

Proposition 6 *The calculi $\lambda_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda_{\mathcal{C}\widehat{tp}}$ are isomorphic.*

4 The $\lambda_{\mathcal{C}\widehat{tp}}$-calculus: static semantics

We consider several natural type (and effect) systems for both $\lambda_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda_{\mathcal{C}\widehat{tp}}$ that are inspired by systems in the literature.

4.1 Type system I: fixed answer type

The simplest way to produce a type system for $\lambda_{\mathcal{C}\widehat{tp}}$ is to adapt the type system of $\lambda_{\mathcal{C}\mathsf{tp}}$ in minor ways: change the occurrences of tp to be $\widehat{tp}$, and add a new variant of the rule *RAA* for typing the new use of $\mathcal{C}$. The resulting system is in Fig. 11.

The type system $\Lambda^{\to fixed}_{\mathcal{C}\widehat{tp}}$ induces the following rule on the $\lambda_{\mathcal{C}\#\mathsf{tp}}$ side:

$$\frac{\Gamma \vdash M : T; T}{\Gamma \vdash \#\ (\mathsf{tp}\ M) : T; T}$$

In other words, every occurrence of the control delimiter in the program must be used at the same fixed type T. Thus, control operators can only be used in contexts that agree with the top-level type. This situation is similar to the type system one gets when defining control operations on top of a continuation monad with a fixed answer type [51]. Despite its obvious restrictions, this system may be adequate for some applications [19].

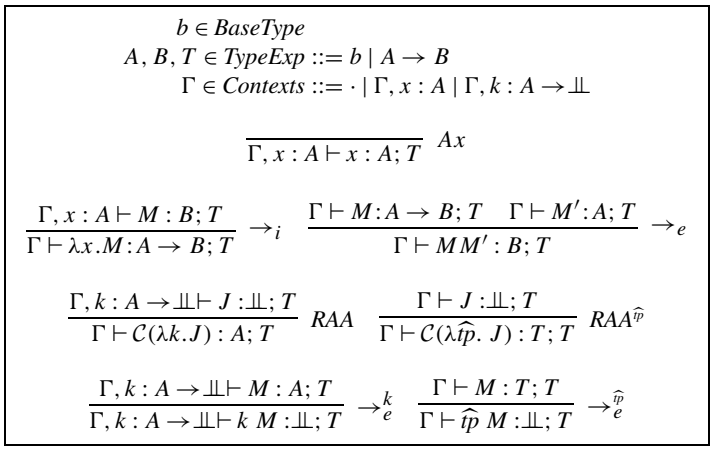

Fig. 11 $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ fixed}$: a type system of $\lambda_{\mathcal{C}\widehat{tp}}$

A standard question about type systems is whether they ensure normalisation of typed λ-terms. It turns out that the type system $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ fixed}$ hides a recursive type in case the top-level type is non-atomic. Strong normalisation is not guaranteed as the following example shows. (See also Proposition 25.)

Example 7 (Loss of SN) We present the example using $\lambda_{\mathcal{C}\#\mathsf{tp}}$. Let $T = (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ be the fixed top-level type, then we can calculate the following types for the given $\lambda_{\mathcal{C}\#\mathsf{tp}}$ expressions:

$$\begin{array}{ll} g :: \mathsf{int} \to T & = \lambda\_.\lambda\_.0 \\ f :: T & = \lambda x.\ (\#\ (\mathsf{tp}\ (g\ (x\ 0))))\ x \\ s :: \mathsf{int} \to \mathsf{int} & = \lambda\_.(\mathcal{A}\ f) \\ e :: \mathsf{int} & = f\ s \end{array}$$

Despite being well-typed, e goes into an infinite loop:

$$f\ s \longrightarrow\!\!\!\!\rightarrow (\#\ (\mathsf{tp}\ (g\ (\mathcal{A}\ f))))\ s \longrightarrow\!\!\!\!\rightarrow (\#\ (\mathsf{tp}\ f))\ s \longrightarrow\!\!\!\!\rightarrow f\ s.$$

4.2 Type system II: dynamic binding as an effect

The requirement that all occurrences of $\widehat{tp}$ (or equivalently, all occurrences of #) are typed with the same fixed top-level type T is overly restrictive. Each introduction of $\widehat{tp}$ can be given a different type. This is captured by the following variant of rule $RAA^{\widehat{tp}}$:

$$\frac{\Gamma \vdash J : \perp\!\!\!\perp; A}{\Gamma \vdash \mathcal{C}(\lambda \widehat{tp}.\ J) : A; T}$$

In this rule, if $\widehat{tp}$ is introduced in a context expecting a value of type A, then it can be called with arguments of type A. In other words, a new top-level type is introduced for the typing of J. Thus, the judgement:

$$\Gamma \vdash M : A; T$$

$$b \in \mathit{BaseType}$$
$$A, B, T \in \mathit{TypeExp} ::= b \mid A \to_T B$$
$$\Gamma \in \mathit{Contexts} ::= \cdot \mid \Gamma, x : A \mid \Gamma, k : A \to_T \bot\!\!\!\bot$$

$$\frac{}{\Gamma, x : A \vdash x : A; T}\ Ax$$

$$\frac{\Gamma, x : A \vdash M : B; T}{\Gamma \vdash \lambda x.M : A \to_T B; T'}\ \to_i \qquad \frac{\Gamma \vdash M : A \to_T B; T \quad \Gamma \vdash M' : A; T}{\Gamma \vdash M M' : B; T}\ \to_e$$

$$\frac{\Gamma, k : A \to_T \bot\!\!\!\bot \vdash J : \bot\!\!\!\bot; T}{\Gamma \vdash \mathcal{C}(\lambda k.J) : A; T}\ RAA \qquad \frac{\Gamma \vdash J : \bot\!\!\!\bot; A}{\Gamma \vdash \mathcal{C}(\lambda \widehat{tp}.\ J) : A; T}\ RAA^{\widehat{tp}}$$

$$\frac{\Gamma, k : A \to_T \bot\!\!\!\bot \vdash M : A; T}{\Gamma, k : A \to_T \bot\!\!\!\bot \vdash k\ M : \bot\!\!\!\bot; T}\ \to_e^k \qquad \frac{\Gamma \vdash M : T; T}{\Gamma \vdash \widehat{tp}\ M : \bot\!\!\!\bot; T}\ \to_e^{\widehat{tp}}$$

Fig. 12 $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ \mathit{eff}_{eq}}$: a type-and-effect system of $\lambda_{\mathcal{C}\widehat{tp}}$

can now be read as "term M returns a value of type A to its immediate context or a value of type T to its enclosing #." On the $\lambda_{\mathcal{C}\#\mathsf{tp}}$ side, this corresponds to the following rule:

$$\frac{\Gamma \vdash M : A; A}{\Gamma \vdash \#\ (\mathsf{tp}\ M) : A; T}$$

This modification is closely related to Murthy's type system [40].

Unfortunately, this modification is unsound by itself as it changes the type of $\widehat{tp}$ without taking into account that it is dynamically bound. Simply modifying the $RAA^{\widehat{tp}}$ rule of the type system of Fig. 11 produces a type system for the static variant of #: the term considered in Example 4 which evaluates to $(\lambda\_.3)$ is given the type int.

Therefore, in addition to having the modified rule $RAA^{\widehat{tp}}$, we also need to modify the system to take into account that $\widehat{tp}$ is a dynamic variable. A possible modification is to repeat what must be done for other dynamically-bound entities like exceptions [25], and to add an effect annotation on *every* arrow type to pass around the type of $\widehat{tp}$. The system with dynamic effect annotations is shown in Fig. 12. Using this system the term in Example 4 has type $A \to_B \mathsf{int}$ which is consistent with the value $\lambda\_.3$. The term $\lambda x.(x\ +\ \mathcal{A}\ \texttt{"Hello"})$ has the type $\mathsf{int} \to_{\mathsf{string}} \mathsf{int}$ with the string constraint correctly associated with the function *call* not the function *definition*.

Proposition 8 *Subject reduction*: *given* $\lambda_{\mathcal{C}\widehat{tp}}$ *terms* M *and* N *if* $\Gamma \vdash M : A; T$ *in* $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ \mathit{eff}_{eq}}$ *and* $M \to N$ *then* $\Gamma \vdash N : A; T$ *in* $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ \mathit{eff}_{eq}}$.

On the $\lambda_{\mathcal{C}\#\mathsf{tp}}$ side, the corresponding type-and-effect system (which we do not present) addresses the loss of strong normalisation discussed in Example 7. The effect annotations impose the following recursive constraint $T = (\mathsf{int} \to_T \mathsf{int}) \to_T \mathsf{int}$. In other words, for the terms to typecheck, we must allow recursive type definitions. This is a situation similar to the one described by Lillibridge [36] where unchecked exceptions can be used to violate strong normalisation. More generally, we can prove that under the type-and-effect system $\Lambda_{\mathcal{C}\widehat{tp}}^{\to\ \mathit{eff}_{eq}}$ of Fig. 12, typed $\lambda_{\mathcal{C}\widehat{tp}}$-terms are strongly normalising. (See Proposition 25.)

Intermezzo 9 In the approach discussed above, different occurrences of the symbol # in a program may have different types. Gunter et al. [24] take the (quite natural) position that occurrences of # with different types should have different names. Each name still has a fixed type but that type is not constrained to be the same as the top-level, nor is it constrained to be related to the types of the other names. As in $\Lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow\, eff_{eq}}$, different occurrences of control delimiters can have different types and the type of the delimiter must be propagated to the control operator. However, it is closer in design to the system with a fixed answer type as all calls to a delimiter of a given name must have the same type. Moreover, since the type system has no effect annotations, Example 7 still typechecks and loops, and well-typed control operations may refer to non-existent control delimiters.

4.3 Understanding the dynamic annotations

The dynamic annotations we used happen to produce a sound type system, but on closer inspection they are not the "right" annotations. In the following discussion, $\neg_{\shortparallel} T$ denotes the type of a continuation variable expecting an argument of type T, as defined in Sect. 2.3.

It is standard to embed type systems with effects into regular type systems using a monadic transformation. Since our effect annotations have to do with $\widehat{tp}$ which is a dynamic variable, a first guess would be to use the environment-passing transformation used to explain dynamic scope [38]. At the level of types, the environment-passing transformation (written $(\cdot)^*$) maps $A \rightarrow_T B$ to $A^* \wedge \neg_{\shortparallel} T^* \rightarrow B^*$, which means that every function is passed the (unique) environment binding as an additional argument (we defer to Sect. 5.1 the question of how to concretely represent the pair of a term and a continuation variable). Judgements $\Gamma \vdash A; T$ are mapped to $\Gamma^*, \neg_{\shortparallel} T^* \vdash A^*$ which means that every expression must be typed in the context of its environment. Writing the translation for the pure fragment is easy, but when it comes to $\mathcal{C}$, the translation of the typing rule *RAA* would produce a continuation whose *input type* is $A^* \wedge \neg_{\shortparallel} T^*$, because continuations also need the environment. But as the rule *RAA* shows, the input type of the continuation corresponds to the *return type* of the expression that captures it. In other words, expressions that might capture continuations must return both their value *and* the environment variable. Thus, our environment-passing embedding becomes a *store-passing transformation*.

A second interpretation of the annotations would be using exceptions: each delimiter installs an exception handler, and calls to $\widehat{tp}$ throw exceptions to the dynamically-closest handler. Indeed, exceptions can be simulated with ordinary dynamic variables [38] and with dynamic continuation variables [24]. According to the standard monadic interpretation of exceptions [37], this leads to a transformation mapping $A \rightarrow_T B$ to $A^* \rightarrow B^* \vee T^*$, which means that every function may return a value or throw an exception to its delimiter. Judgements $\Gamma \vdash A; T$ are mapped to $\Gamma^* \vdash A^* \vee T^*$ and have a similar interpretation. When writing such a translation in a general setting [44, 48, 49], one is faced with a choice: should a use of a control operator capture the current exception handler or not? In our setting, the question is: if a continuation is captured under some #, and later invoked under another #, should calls to $\widehat{tp}$ refer to the first # or the second? It is clear that our semantics requires the second choice, which turns out to be consistent with the SML/NJ control operators `capture` and `escape`. In combination with exceptions, these control operators can simulate state [49, Fig. 12] which means that our embedding also becomes a *store-passing transformation*.

The above discussion suggests the following interpretation: functions $A \rightarrow_T B$ are mapped to $A^* \wedge \neg_{\shortparallel} T^* \rightarrow B^* \wedge \neg_{\shortparallel} T^*$ and judgements $\Gamma \vdash A; T$ are mapped to $\Gamma^*, \neg_{\shortparallel} T^* \vdash A^* \wedge \neg_{\shortparallel} T^*$. This interpretation is a standard store-passing one, which is consistent with Filinski's observation that *shift* and *reset* can be implemented using continuations and state [19,

20]. The entire analysis is also consistent with the fact that in *classical logic*, the following formulae are all logically equivalent (even though they are not computationally equivalent):

$$\begin{array}{ll} A \wedge \neg T \to B & (\widehat{tp} \textit{ as an environment}) \\ A \to B \vee T & (\widehat{tp} \textit{ as an exception}) \\ A \wedge \neg T \to B \wedge \neg T & (\widehat{tp} \textit{ as a state}) \end{array}$$

4.4 Type system III: state as effects

As the analysis in the previous section shows, $\widehat{tp}$ can be understood as a state parameter and this understanding leads to a different, more expressive, type-and-effect system, which maintains the type of $\widehat{tp}$ *before* and *after* each computation. This generalisation gives the type-and-effect system $\Lambda^{\overrightarrow{eff}}_{\mathcal{C}\widehat{tp}}$ of Fig. 13, which is essentially identical to the one developed by Danvy and Filinski as early as [9].

In the cases of jumps and continuations, the judgements and types of the new type system are the same as before. When typing terms, the judgements have the form $\Gamma; U \vdash M : A; T$ with T and U describing the top-level continuation before and after the evaluation of the term M, respectively. For terms without $\mathcal{C}$, we can show by induction that the two formulae T and U are the same. For terms of the form $\mathcal{C}(\lambda k.J)$, the formula U is the type of the top-level continuation when k is invoked. Implication has two effects $A \ {}_U{\to}_T \ B$ with T describing the top-level continuation *before* the call and U describing the top-level continuation *after* the call. These changes make the typing of applications sensitive to the order of evaluation of the function and argument: the rule $\to_e$ assumes the function is evaluated before the argument. The new system is sound.

Proposition 10 *Subject reduction*: *given* $\lambda_{\mathcal{C}\widehat{tp}}$ *terms* M *and* N, *if* $\Gamma; U \vdash M : A; T$ *and* $M \to N$ *then* $\Gamma; U \vdash N : A; T$.

$$\begin{array}{c} b \in \mathit{BaseType} \\ A, B, T, U \in \mathit{TypeExp} ::= b \mid A \ {}_U{\to}_T \ B \\ \Gamma \in \mathit{Contexts} ::= \cdot \mid \Gamma, x : A \mid \Gamma, k : A \to_T \perp\!\!\!\perp \end{array}$$

$$\frac{}{\Gamma, x : A; T \vdash x : A; T} \ Ax$$

$$\frac{\Gamma, x : A; U \vdash M : B; T}{\Gamma; T' \vdash \lambda x.M : A \ {}_U{\to}_T \ B; T'} \ {\to_i}$$

$$\frac{\Gamma; U_1 \vdash M : A \ {}_{U_2}{\to}_{T_1} \ B; T_2 \quad \Gamma; T_1 \vdash N : A; U_1}{\Gamma; U_2 \vdash MN : B; T_2} \ {\to_e}$$

$$\frac{\Gamma, k : A \to_U \perp\!\!\!\perp \vdash J : \perp\!\!\!\perp; T}{\Gamma; U \vdash \mathcal{C}(\lambda k.J) : A; T} \ RAA \qquad \frac{\Gamma \vdash J : \perp\!\!\!\perp; A}{\Gamma; T \vdash \mathcal{C}(\lambda \widehat{tp}.\ J) : A; T} \ RAA^{\widehat{tp}}$$

$$\frac{\Gamma, k : A \to_U \perp\!\!\!\perp; U \vdash M : A; T}{\Gamma, k : A \to_U \perp\!\!\!\perp \vdash k\ M : \perp\!\!\!\perp; T} \ {\to^k_e} \qquad \frac{\Gamma; U \vdash M : U; T}{\Gamma \vdash \widehat{tp}\ M : \perp\!\!\!\perp; T} \ {\to^{\widehat{tp}}_e}$$

Fig. 13 $\Lambda^{\overrightarrow{eff}}_{\mathcal{C}\widehat{tp}}$: another type-and-effect system of $\lambda_{\mathcal{C}\widehat{tp}}$

Let A_{df} be the operation of changing each arrow $B \to_T C$ into the arrow $B\ {}_T{\to}_T\ C$ in the formula A. Let Γ_{df} be the extension of this operation to Γ. Let A_b be the operation of adding twice the same atomic effect b on the occurrences of $\to$ in the effect-free formula A (i.e. $B \to C$ becomes $B\ {}_b{\to}_b\ C$). Let Γ_b be the extension of this operation to an effect-free Γ. The new type-and-effect system generalises the previous systems.

Proposition 11 (i) *If* $\Gamma \vdash M : A; T$ (*resp.* $\Gamma \vdash J :\!\perp\!\!\!\perp; T$) *in* $\Lambda^{\to\ eff_{eq}}_{\mathcal{C}\widehat{tp}}$ *then* $\Gamma_{df}; T_{df} \vdash M : A_{df}; T_{df}$ (*resp.* $\Gamma_{df} \vdash J :\!\perp\!\!\!\perp; T_{df}$) *in* $\Lambda^{\to\ eff}_{\mathcal{C}\widehat{tp}}$.

(ii) *For atomic* b, *if* $\Gamma \vdash M : A; b$ (*resp.* $\Gamma \vdash J :\!\perp\!\!\!\perp; b$) *in* $\Lambda^{\to\ fixed}_{\mathcal{C}\widehat{tp}}$ *then* $\Gamma_b; b \vdash M : A_b; b$ (*resp.* $\Gamma_b \vdash J :\!\perp\!\!\!\perp; b$) *in* $\Lambda^{\to\ eff}_{\mathcal{C}\widehat{tp}}$.

The added expressiveness of the type system is illustrated with the following example.

Example 12 The term $\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (1\ +\ \mathcal{C}(\lambda k.\ \widehat{tp}\ (2\ ==\ \mathcal{C}(\lambda\widehat{tp}.\ k\ 3)))))$ is rejected by $\Lambda^{\to\ eff_{eq}}_{\mathcal{C}\widehat{tp}}$ but accepted in $\Lambda^{\to\ eff}_{\mathcal{C}\widehat{tp}}$ with the following rule for typing the equality:

$$\frac{\Gamma; T_1 \vdash M : A; T \quad \Gamma; U \vdash N : A; T_1}{\Gamma; U \vdash M == N : \mathsf{bool}; T}$$

The first occurrence of k is delimited by the outermost occurrence of $\widehat{tp}$ which is of type int, but when k is invoked, the context is delimited by a type bool. Indeed, the term evaluates to a boolean value as shown below:

$$\begin{aligned}
&\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (1\ +\ \mathcal{C}(\lambda k.\ \widehat{tp}\ (2\ ==\ \mathcal{C}(\lambda\widehat{tp}.\ k\ 3)))))\\
\to\ &\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (\mathcal{C}(\lambda k.\ \widehat{tp}\ (2\ ==\ \mathcal{C}(\lambda\widehat{tp}.\ k\ (1+3))))))\\
\to\ &\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (2\ ==\ \mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (1+3))))\\
\twoheadrightarrow\ &\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (2\ ==\ 4)) \twoheadrightarrow \mathsf{false}
\end{aligned}$$

In general, the new type system is expressive enough to give different types to different occurrences of $\widehat{tp}$, as shown in the next example.

Example 13 In the term $\mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (\widehat{\mathcal{A}}\ 5 == \widehat{\mathcal{A}}\ \texttt{"Hello"}))$ the visible occurrence of $\widehat{tp}$ and the two implicit occurrences in $\widehat{\mathcal{A}}$ have the three different types: bool, int and string respectively, as indicated in the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;\ \mathsf{int} \vdash 5 : \mathsf{int}; \mathsf{int}}\ Ax \quad \dfrac{}{;\ \mathsf{string} \vdash \texttt{"Hello"} : \mathsf{string}; \mathsf{string}}\ Ax}{;\ \mathsf{string} \vdash \widehat{\mathcal{A}}\ 5 : A; \mathsf{int} \qquad ;\ \mathsf{bool} \vdash \widehat{\mathcal{A}}\ \texttt{"Hello"} : A; \mathsf{string}}}{;\ \mathsf{bool} \vdash \widehat{\mathcal{A}}\ 5 == \widehat{\mathcal{A}}\ \texttt{"Hello"} : \mathsf{bool}; \mathsf{int}}}{\vdash \widehat{tp}\ (\widehat{\mathcal{A}}\ 5 == \widehat{\mathcal{A}}\ \texttt{"Hello"}) :\!\perp\!\!\!\perp; \mathsf{int}}\ \to^{\widehat{tp}}_{e}}{;\ T \vdash \mathcal{C}(\lambda\widehat{tp}.\ \widehat{tp}\ (\widehat{\mathcal{A}}\ 5 == \widehat{\mathcal{A}}\ \texttt{"Hello"})) : \mathsf{int}; T}\ RAA^{\widehat{tp}}$$

5 The $\lambda^{-}_{\mathcal{C}}$-calculus

We show that the dual connective of implication, namely subtraction (written $A - B$) arises as the natural type for carrying around the type of the top-level continuation, and formally introduce the $\lambda^{-}_{\mathcal{C}}$-calculus.

Fig. 14 Interpreting the effect annotations

$$
\begin{array}{ll}
b^* & = b \\
(A\ {}_U{\to}_T\ B)^* & = (A^* - T^*) \to (B^* - U^*) \\
 & \\
(\cdot)^* & = \cdot \\
(\Gamma, x : A)^* & = \Gamma^*, x : A^* \\
(\Gamma, k : A \to_U \perp\!\!\!\perp)^* & = \Gamma^*, k : (A^* - U^*) \to \perp\!\!\!\perp
\end{array}
$$

5.1 Subtraction

Our analysis in Sect. 4.3 together with our understanding of the system $\Lambda_{\mathcal{C}\widehat{\mathit{ur}}}^{\to\ \mathit{eff}}$ suggest we interpret the more general effect annotations as follows:

$$
\begin{aligned}
(A\ {}_U{\to}_T\ B)^* &= A^* \wedge \neg_{\perp\!\!\!\perp} T^* \to B^* \wedge \neg_{\perp\!\!\!\perp} U^*, \\
(\Gamma; U \vdash A; T)^* &= \Gamma^*, \neg_{\perp\!\!\!\perp} T^* \vdash A^* \wedge \neg_{\perp\!\!\!\perp} U^*.
\end{aligned}
$$

But the question arises: how to represent $\neg_{\perp\!\!\!\perp} T^*$ as a type? A natural attempt is to embed $\neg_{\perp\!\!\!\perp} T^*$ into the type of functions $T^* \to \perp$ as shown in Sect. 2.3, but it is also possible to investigate a more abstract solution based on the subtraction connective.

The *subtractive type* has been previously studied by Rauszer [43] and Crolard [6], and has been integrated by Curien and Herbelin [8] in their study of the *duality* between the *producers* of values (which are regular terms) and the *consumers* of values (which are contexts or continuations). Subtraction has been also considered in Filinski [18] where it is named a *co-exponential*. In particular, Filinski presents a duality between functions of type $A \to B$ and continuations of type $A - B$ (there written $[B \leftarrow A]$) that are interpreted as continuation transformers. In our work, the subtraction arises as the natural type for pairing a term and an evaluation context and we use it as a direct representation of this information rather than to rely on an embedding of evaluation contexts (of type $\neg_{\perp\!\!\!\perp} T$) into functions (of type $\neg T$).

The actual interpretation of effects we use is in Fig. 14. (The correctness of the interpretation is discussed in Sect. 6 after we explain the target of the translation in detail.) The interpretation shows that a continuation k has type $(A - T) \to \perp\!\!\!\perp$. Understanding $A - T$ as the type $A \wedge \neg T$ means that, as Danvy and Filinski explain, every delimited continuation must be given a value and another continuation (called the metacontinuation in their original article [10] and referring to the top-level continuation as we explain). The type $A - T$ is also equivalent to $\neg(\neg T \to \neg A)$ and with that view, a delimited continuation is more like a continuation transformer, which is an idea closely related to Queinnec and Moreau's formalisation of delimited continuations as the difference between two continuations [39].

The embedding of a function type is of the form $(A - T) \to (B - U)$ which captures the idea that the top-level type T needed to execute the function's body should be available when the function is called, and that the new top-level type U is returned as part of the result.

Intermezzo 14 Figure 15 presents the subtraction rules introduced by Crolard [7] in the style of Parigot's classical natural deduction. In that setting one works with two sets of assumptions: Γ and Δ, where Δ maintains the continuation variables. In a setting with continuation variables written on the left-hand side of the sequent, the natural representation of $\Delta = \{A_1, \ldots, A_n\}$ is as $\neg_{\perp\!\!\!\perp} \Delta$ defined as $\{\neg_{\perp\!\!\!\perp} A_1, \ldots, \neg_{\perp\!\!\!\perp} A_n\}$. With this notation, the

Fig. 15 Subtractive logic in Parigot's style (classical natural deduction)

$$\frac{\Gamma \vdash \Delta, C; A \quad \Gamma, B \vdash \Delta, C; C}{\Gamma \vdash \Delta, C; A - B}\ -i$$

$$\frac{\Gamma \vdash \Delta; A - B \quad \Gamma, A \vdash \Delta; B}{\Gamma \vdash \Delta; C}\ -e$$

Fig. 16 Classical subtractive logic

$$A, B ::= X \mid A \to B \mid A - B$$
$$\Gamma ::= \cdot \mid \Gamma, A \mid \Gamma, A \to \perp\!\!\!\perp$$

$$\frac{}{\Gamma, A \vdash A}\ Ax$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$$

$$\frac{\Gamma, A \to \perp\!\!\!\perp \vdash \perp\!\!\!\perp}{\Gamma \vdash A}\ RAA \qquad \frac{\Gamma, A \to \perp\!\!\!\perp \vdash A}{\Gamma, A \to \perp\!\!\!\perp \vdash \perp\!\!\!\perp}\ \to_e^k$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash \perp\!\!\!\perp}{\Gamma \vdash A - B}\ -i \qquad \frac{\Gamma \vdash A - B \quad \Gamma, A, B \to \perp\!\!\!\perp \vdash \perp\!\!\!\perp}{\Gamma \vdash \perp\!\!\!\perp}\ -e$$

subtraction rules rewrite to:

$$\frac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash A \quad \Gamma, B, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash C}{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash A - B}\ -i$$

$$\frac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta \vdash A - B \quad \Gamma, A, \neg_{\perp\!\!\!\perp} \Delta \vdash B}{\Gamma, \neg_{\perp\!\!\!\perp} \Delta \vdash C}\ -e$$

Figure 16 presents our variant of the classical natural deduction with subtraction. A formula is built from a set of atomic formulae (X), which we leave unspecified, and two logical connectives. We make use of two kinds of judgements: $\Gamma \vdash A$ and $\Gamma \vdash \perp\!\!\!\perp$. The rules for subtraction are slightly different. We can recover Crolard's rules as follows:

$$\frac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash A \quad \dfrac{\Gamma, B, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash C}{\Gamma, B, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash \perp\!\!\!\perp}\ \to_e^k}{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash A - B}\ -i$$

$$\frac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash A - B \quad \dfrac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} B, A, \neg_{\perp\!\!\!\perp} C \vdash B}{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} B, A, \neg_{\perp\!\!\!\perp} C \vdash \perp\!\!\!\perp}\ \to_e^k}{\dfrac{\Gamma, \neg_{\perp\!\!\!\perp} \Delta, \neg_{\perp\!\!\!\perp} C \vdash \perp\!\!\!\perp}{\Gamma, \neg_{\perp\!\!\!\perp} \Delta \vdash C}\ RAA}\ -e$$

5.2 Syntax, semantics, and type system

We formalise the $\lambda_{\bar{C}}$-calculus by extending the original $\lambda_{C_{\mathsf{tp}}}$ with subtraction and removing the special constant tp since we are maintaining the top-level continuation using the subtractive type. The resulting calculus is a call-by-value variant of Crolard's extension of the $\lambda\mu$-calculus with subtraction [7].

Fig. 17 Syntax of $\lambda_{\mathcal{C}}^{-}$

$$
\begin{array}{rcl}
x, a, v, f & \in & \mathit{Vars} \\
k, tp & \in & \mathit{KVars} \\
M, N & \in & \mathit{Terms} ::= x \mid \lambda x.\, M \mid M\, N \mid \mathcal{C}(\lambda k.\, J) \mid \\
 & & \quad (M, \lambda x.\, J_c[E[x]]) \\
V & \in & \mathit{Values} ::= x \mid \lambda x.\, M \mid (V, \lambda x.\, J_c[E[x]]) \\
J & \in & \mathit{Jumps} ::= k\ M \mid \mathbf{let}\ (x,k) = M\ \mathbf{in}\ J \\
 & & \\
J_c & \in & \mathit{ElemJumpCtx} ::= k\ \Box \mid \mathbf{let}\ (x,k) = \Box\ \mathbf{in}\ J \\
F & \in & \mathit{ElemCtxt} ::= \Box\ M \mid V\ \Box \mid (\Box, \lambda x.\, J_c[E[x]]) \\
E & \in & \mathit{EvCtxt} ::= \Box \mid E[F]
\end{array}
$$

$$
\begin{array}{lll}
\beta_v : & (\lambda x.M)V & \to M\,[V/x] \\
\mathcal{C}_{\mathit{lift}} : & F[\mathcal{C}(\lambda k.\, J)] & \to \mathcal{C}(\lambda k.\, J\,[k\ F/k]) \\
\mathcal{C}_{\mathit{elim}} : & \mathcal{C}(\lambda k.k\ M) & \to M \quad \text{where } k \notin FV(M) \\
\mathcal{C}_{\mathit{idem}} : & k'\ \mathcal{C}(\lambda k.\, J) & \to J\,[k'\ \Box/k] \\
\mathit{Sub}_v^{\mathit{lift}} : & \mathbf{let}\ (x,k) = \mathcal{C}(\lambda k'.\, J') & \to J'\ [\mathbf{let}\ (x,k) = \Box\ \mathbf{in}\ J/k'] \\
 & \mathbf{in}\ J & \\
\mathit{Sub}_v^{\mathit{base}} : & \mathbf{let}\ (x,k) = (V, \lambda x.\, J_c[x]) & \to J\,[J_c/k; V/x] \\
 & \mathbf{in}\ J & \\
\mathit{Sub}_v^{\mathit{step}} : & \mathbf{let}\ (x,k) = (V, \lambda x.\, J_c[E[F[x]]]) & \to \mathbf{let}\ (x,k) = (V, \lambda x.\, J_c[E[x]]) \\
 & \mathbf{in}\ J & \quad \mathbf{in}\ J\,[k\ F/k]
\end{array}
$$

Fig. 18 Reductions of call-by-value $\lambda_{\mathcal{C}}^{-}$

The syntax is given in Fig. 17. Subtractions are introduced by terms of the form $(M, \lambda x.\, k\ E[x])$ or $(M, \lambda x.\, \mathbf{let}\ (x', k) = E[x]\ \mathbf{in}\ J)$. We represent them both using the syntactic category J_c and write $(M, \lambda x.\, J_c[E[x]])$. We call the second component of the pair a *jump context*. The use of a jump context in place of a continuation variable is motivated and explained below. The notation $\lambda x.\, J_c[E[x]]$ is basically a writing for the pair of the elementary jump context J_c and of the evaluation context E. The role of the abstraction over x is purely decorative: it is a way to recall that the hole of $J_c[E]$ is bound in the pair $(M, \lambda x.\, J_c[E[x]])$. In particular, in the evaluation context $(E', \lambda x.\, J_c[E[x]])$, the hole is in E', not in E. As an abuse of notation, we write k as an abbreviation for the jump context $\lambda x.k\ x$. Subtractions are eliminated by a jump of the form $\mathbf{let}\ (x,k) = M\ \mathbf{in}\ J$. The notation $E[F]$ is pure syntax: it denotes the pair of E and F. Contrastingly, each of the notations $F[M]$ and $E[M]$ (resp. $J_c[M]$ and $J_c[E[M]]$) that are used later on denote the term (resp. jump) obtained by interpreting the context as a partial term (resp. jump) and by filling the hole of this partial term with M.

The reduction semantics is given in Fig. 18. The $\mathcal{C}_{\mathit{lift}}$ rule lifts a $\mathcal{C}$-expression out of any of the three possible elementary contexts F: these include applicative contexts as before as well as contexts of the form $(\Box, \lambda x.\, J_c[E[x]])$.

Consider the $\lambda_{\mathcal{C}\widehat{tp}}$ term $\mathcal{C}(\lambda k.\, \widehat{tp}\ (\mathcal{I}h\ k\ 2))$. After the invocation of continuation k, control goes back to the current top-level continuation bound to $\widehat{tp}$. If this top-level continuation has value tp, then we can express the example using subtraction by writing $\mathcal{C}(\lambda k.\, k\ (2, tp))$. In $\lambda_{\mathcal{C}}^{-}$, the dynamically-scoped variable $\widehat{tp}$ disappears. Instead, the invocation of k is given the current top-level context which must be invoked next. In this particular case, the jump context is essentially just a continuation variable but things can be more complicated.

As another example, consider the term $\mathcal{C}(\lambda k.\, \widehat{tp}\ (\mathcal{C}(\lambda \widehat{tp}.\, k\ 2) == 3))$. After the invocation of continuation k, control returns to the context $\Box == 3$ and then to the top-level continua-

tion bound to $\widehat{tp}$. If this top-level continuation has value tp, then we can express the example using subtraction by writing $\mathcal{C}(\lambda k.\, k\ (2, \lambda x.\, tp\ (x\ ==\ 3)))$. In this case, the jump context is not simply a continuation variable, indicating that the continuation composes with another context before returning to the top-level.

To see this behaviour simulated by the reduction rules, consider a jump expression which eliminates the subtraction and invokes the continuation:

$$\textbf{let}\ (x, k) = (2, \lambda x.\, tp\ (x\ ==\ 3))\ \textbf{in}\ k\ x$$

Using Sub_v^{step}, the context $\square\ ==\ 3$ is lifted:

$$\begin{array}{ll}\textbf{let}\ (x, k) = (2, \lambda x.\, tp\ (x\ ==\ 3)) \rightarrow & \textbf{let}\ (x, k) = (2, tp) \\ \textbf{in}\ k\ x & \textbf{in}\ k\ (x\ ==\ 3)\end{array}$$

and then using Sub_v^{base} control is transferred to the top-level:

$$\begin{array}{l}\textbf{let}\ (x, k) = (2, tp) \rightarrow tp\ (2\ ==\ 3) \\ \textbf{in}\ k\ (x\ ==\ 3)\end{array}$$

In general the jump context may include an arbitrary nesting of elementary contexts. For example, we can have:

$$\begin{array}{l}\textbf{let}\ (x, k) = (z, \lambda x.\, tp\ (((x\ N_1)\ N_2)\ N_3)) \\ \textbf{in}\ k\ x\end{array}$$

The question arises whether it is more convenient or natural to lift the context $((\square\ N_1)\ N_2)\ N_3$ in one step or in many steps in the style of the rule $\mathcal{C}_{lift}$. We adopt a one by one lifting of $\square\ N_1$, $\square\ N_2$ and $\square\ N_3$ as shown below:

$$\begin{array}{rl} & \textbf{let}\ (x, k) = (z, \lambda x.\, tp\ (((x\ N_1)\ N_2)\ N_3))\ \textbf{in}\ k\ x \\ \rightarrow & \textbf{let}\ (x, k) = (z, \lambda x.\, tp\ ((x\ N_2)\ N_3))\ \textbf{in}\ k\ (x\ N_1) \\ \rightarrow & \textbf{let}\ (x, k) = (z, \lambda x.\, tp\ (x\ N_3))\ \textbf{in}\ k\ ((x\ N_1)\ N_2) \\ \rightarrow & \textbf{let}\ (x, k) = (z, tp)\ \textbf{in}\ k\ (((x\ N_1)\ N_2)\ N_3) \\ \rightarrow & \ tp\ (((z\ N_1)\ N_2)\ N_3)\end{array}$$

The above discussion motivates jump contexts of the form $\lambda x.k\ \ E[x]$. The form $\lambda x.\, \textbf{let}\ (x', k) = E[x]\ \textbf{in}\ J$ is motivated by the following Sub_v^{lift}-reduction:

$$\begin{array}{ll}\textbf{let}\ (x', k) = \mathcal{C}(\lambda q.\, k'\ (2, \lambda x.\, q\ x)) \rightarrow k'\ (2, \lambda x.\ \textbf{let}\ (x', k) = x \\ \textbf{in}\ k\ x' & \textbf{in}\ k\ x'\ )\end{array}$$

Intermezzo 15 In Sect. 2.3, we showed how to embed $\neg_{\perp\!\!\!\perp} T$ into $\neg T$. For the other direction, we need to explicitly consider an elimination rule for $\perp$:

$$\frac{\Gamma \vdash M : \perp}{\Gamma \vdash \mathsf{tp}_\perp\ M : \perp\!\!\!\perp}\ \perp_e$$

The continuation $\mathsf{tp}_\perp$ is the constant tp considered in Sect. 2 restricted to the $\perp$ type: it is the dual of the unique inhabitant of the unit type $\top$. Using $\mathsf{tp}_\perp$, any term M of type $T \rightarrow \perp$ can be turned into the jump context $\lambda x.\, \mathsf{tp}_\perp\ ((\lambda y.M\ y)\ x)$.

Fig. 19 $\Lambda_{\mathcal{C}}^{\rightarrow -}$: type system of $\lambda_{\mathcal{C}}^{-}$

$$b \in \mathit{BaseType}$$
$$A, B, C, T, U \in \mathit{TypeExp} ::= b \mid A \to B \mid A - B$$
$$\Gamma \in \mathit{Contexts} ::= \cdot \mid \Gamma, x : A \mid \Gamma, k : A \to \perp\!\!\!\perp$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ Ax$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to_i \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash M' : A}{\Gamma \vdash M\,M' : B}\ \to_e$$

$$\frac{\Gamma, k : A \to \perp\!\!\!\perp \vdash J : \perp\!\!\!\perp}{\Gamma \vdash \mathcal{C}(\lambda k.J) : A}\ RAA \qquad \frac{\Gamma, k : A \to \perp\!\!\!\perp \vdash M : A}{\Gamma, k : A \to \perp\!\!\!\perp \vdash k\ M : \perp\!\!\!\perp}\ \to_e^k$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : B \vdash J_c[E[x]] : \perp\!\!\!\perp}{\Gamma \vdash (M, \lambda x.\, J_c[E[x]]) : A - B}\ -_i$$

$$\frac{\Gamma \vdash M : A - B \qquad \Gamma, x : A, k : B \to \perp\!\!\!\perp \vdash J : \perp\!\!\!\perp}{\Gamma \vdash \mathbf{let}\ (x, k) = M\ \mathbf{in}\ J : \perp\!\!\!\perp}\ -_e$$

The following abbreviations capture some common patterns:

$$\lambda(x, k).M \triangleq \lambda v.\mathcal{C}(\lambda q.\,\mathbf{let}\ (x, k) = v\ \mathbf{in}\ q\ M) \quad \text{(Abbrev. 4)}$$

$$\mathbf{join}\ M \triangleq \mathbf{let}\ (x, k) = M\ \mathbf{in}\ k\ x \quad \text{(Abbrev. 5)}$$

$$\mathbf{bind}_{q,k}\ M\ \mathbf{in}\ N \triangleq \mathbf{let}\ (f, k) = M\ \mathbf{in}\ q\ (f\ N) \quad \text{(Abbrev. 6)}$$

In the first abbreviation v is a fresh variable. The operator **join** abbreviates the common pattern where the elimination form of the subtractive value immediately throws the value to the jump context. The operator **bind** is similar to the monadic operator of the same name. Both M and N are expected to be terms that evaluate to subtractive values with N containing free occurrences of k and f is a fresh variable: the effects of M are performed to produce a subtractive value which is bound to (f, k) and then f is applied to N.

The typing rules for the complete language are in Fig. 19. The system named $\Lambda_{\mathcal{C}}^{\rightarrow -}$ is completely standard with no effect annotations and not even a global parameter T. Note that the right premise of the rule $-_i$ must be read as "there is a derivation of $\Gamma, x : B \vdash J : \perp\!\!\!\perp$ for J obtained by filling the jump context J_c with $E[x]$ where x is chosen such that it does not occur in J_c nor in E."

Proposition 16 (1) *Subject reduction*: *Given* $\lambda_{\mathcal{C}}^{-}$ *terms* M *and* N *if* $\Gamma \vdash M : A$ *and* $M \to N$ *then* $\Gamma \vdash N : A$.

(2) *Typed* $\lambda_{\mathcal{C}}^{-}$ *terms are strongly normalising*.

Proof Subject reduction is routine verification. The proof of strong normalisation is expanded in detail in Appendix. □

6 Embeddings in $\lambda_{\mathcal{C}}^{-}$

We introduce the full embedding of $\lambda_{\mathcal{C}\widehat{ip}}$ into $\lambda_{\mathcal{C}}^{-}$ and show its correctness. Our starting point is the *effect logic* of Fig. 20 which gives the type judgements of $\lambda_{\mathcal{C}\widehat{ip}}$ with no terms. The aim

$$A, B, T, U ::= X \mid A \;{}_U\!\!\rightarrow_T B$$
$$\Gamma ::= \cdot \mid \Gamma, A \mid \Gamma, A \rightarrow_T \perp\!\!\!\perp$$

$$\frac{}{\Gamma, A; T \vdash A; T}\; Ax$$

$$\frac{\Gamma, A; U \vdash B; T}{\Gamma; T' \vdash : A \;{}_U\!\!\rightarrow_T B; T'}\; \rightarrow_i \qquad \frac{\Gamma; U_1 \vdash A \;{}_{U_2}\!\!\rightarrow_{T_1} B; T_2 \quad \Gamma; T_1 \vdash A; U_1}{\Gamma; U_2 \vdash B; T_2}\; \rightarrow_e$$

$$\frac{\Gamma, A \rightarrow_U \perp\!\!\!\perp \vdash \perp\!\!\!\perp; T}{\Gamma; U \vdash A; T}\; RAA \qquad \frac{\Gamma \vdash \perp\!\!\!\perp; A}{\Gamma; T \vdash A; T}\; RAA^{\widehat{tp}}$$

$$\frac{\Gamma, A \rightarrow_U \perp\!\!\!\perp; U \vdash A; T}{\Gamma, A \rightarrow_U \perp\!\!\!\perp \vdash \perp\!\!\!\perp; T}\; \rightarrow^k_e \qquad \frac{\Gamma; U \vdash U; T}{\Gamma \vdash \perp\!\!\!\perp; T}\; \rightarrow^{\widehat{tp}}_e$$

Fig. 20 Natural deduction with effects

Fig. 21 Embedding of $\lambda_{\mathcal{C}\widehat{tp}}$ terms and jumps into $\lambda^-_{\mathcal{C}}$

$$\begin{aligned}
x^+ &= x \\
(\lambda x.M)^+ &= \lambda(x, tp).M^{tp} \\
\\
V^{tp} &= (V^+, tp) \\
(V\,N)^{tp} &= V^+\,N^{tp} \\
(M\,N)^{tp} &= \mathcal{C}(\lambda q.\,\mathbf{bind}_{q,tp}\,M^{tp}\,\mathbf{in}\,N^{tp}) \quad (M \text{ not a value}) \\
(\mathcal{C}(\lambda k.\,J))^{tp} &= \mathcal{C}(\lambda k.\,J^{tp}) \\
(\mathcal{C}(\lambda \widehat{tp}.\,J))^{tp} &= (\mathcal{C}(\lambda tp.J^{tp}), tp) \\
\\
(k\;M)^{tp} &= k\;M^{tp} \\
(\widehat{tp}\;M)^{tp} &= \mathbf{join}\;M^{tp}
\end{aligned}$$

is to embed such judgements into the judgements of the subtractive logic of Fig. 16, which is a term-free version of the type system of $\lambda^-_{\mathcal{C}}$:

$$\begin{aligned}
(\Gamma; B \vdash A; C)^* &= \Gamma^*, \neg_{\perp\!\!\!\perp} C^* \vdash A^* - B^*, \\
(\Gamma \vdash \perp\!\!\!\perp; C)^* &= \Gamma^*, \neg_{\perp\!\!\!\perp} C^* \vdash \perp\!\!\!\perp .
\end{aligned}$$

The embedding extends the embedding of Fig. 14 which maps $\lambda_{\mathcal{C}\widehat{tp}}$ types and contexts into $\lambda^-_{\mathcal{C}}$.

When extended with a term assignment, the embeddings become:

$$\begin{aligned}
(\Gamma; B \vdash M : A; C)^{tp} &= \Gamma^*, tp : \neg_{\perp\!\!\!\perp} C^* \vdash M^{tp} : A^* - B^*, \\
(\Gamma \vdash J :\perp\!\!\!\perp; C)^{tp} &= \Gamma^*, tp : \neg_{\perp\!\!\!\perp} C^* \vdash J^{tp} :\perp\!\!\!\perp,
\end{aligned}$$

where the value tp of the dynamic top-level continuation is required as a parameter.

Figure 21 summarises the embedding of terms and jumps. The translation in the figure includes a special translation $(\cdot)^+$ on values with the following property. The translation of a value has no free occurrences of the continuation variable used in the embedding. More precisely, every function takes a subtractive value as an argument which specifies the original argument and the top-level continuation resulting from the evaluation of the function and the argument. This allows us to infer for example that in the special case where we jump to the

top-level with a value, we have $(\widehat{tp}\ V)^{tp} \twoheadrightarrow tp\ V^+$. We further discuss some cases of the embedding below.

In the translation of an application of the form MN, for M not a value, the references to the top-level in N^{tp} are bound by the $\mathbf{bind}_{q,tp}$, while the occurrences of the top-level in M^{tp} are free. These relationships mimic the fact that the actual binding for a call to $\widehat{tp}$ in N is the one active when M returns its result, whereas the actual binding for a call to $\widehat{tp}$ in M is the one at the time of evaluating MN. The embedding of $\mathcal{C}(\lambda\widehat{tp}.\,J)$ returns a subtractive term with the current top-level continuation tp as the jump context and a computation which introduces a fresh tp thus simulating the rebinding of the top-level continuation in J. Note that tp does *not* expect a subtractive value as its argument. Thus, the argument passed to a top-level jump is evaluated to produce a value and a jump context, which are consumed before returning to tp.

6.1 Soundness of the embedding of $\lambda_{\mathcal{C}\widehat{tp}}$ into $\lambda^{-}_{\mathcal{C}}$

For tp a fresh continuation variable, the embedding is sound in the sense that it maps judgements of $\Lambda^{\rightarrow\ eff}_{\mathcal{C}\widehat{tp}}$ (Fig. 13) to judgements of $\Lambda^{\rightarrow -}_{\mathcal{C}}$ (Fig. 19).

Proposition 17 (i) *If* $\Gamma; U \vdash M : A; T$ *in* $\Lambda^{\rightarrow\ eff}_{\mathcal{C}\widehat{tp}}$ *then* $(\Gamma; U \vdash M : A; T)^{tp}$ *in* $\Lambda^{\rightarrow -}_{\mathcal{C}}$,
(ii) *If* $\Gamma \vdash J :\!\perp\!\!\!\perp; T$ *in* $\Lambda^{\rightarrow\ eff}_{\mathcal{C}\widehat{tp}}$ *then* $(\Gamma \vdash J :\!\perp\!\!\!\perp; T)^{tp}$ *in* $\Lambda^{\rightarrow -}_{\mathcal{C}}$.

Based on Proposition 11, the soundness argument also extends to the type system $\Lambda^{\rightarrow\ eff_{eq}}_{\mathcal{C}\widehat{tp}}$ (Fig. 12) and $\Lambda^{\rightarrow\ fixed}_{\mathcal{C}\widehat{tp}}$ (Fig. 11) for $\lambda_{\mathcal{C}\widehat{tp}}$.

Proposition 18 (i) *If* $\Gamma \vdash M : A; T$ *in* $\Lambda^{\rightarrow\ eff_{eq}}_{\mathcal{C}\widehat{tp}}$ *then* $(\Gamma_{df}; T_{df} \vdash M : A_{df}; T_{df})^{tp}$ *in* $\Lambda^{\rightarrow -}_{\mathcal{C}}$.
(ii) *If* $\Gamma \vdash M : A; b$ *in* $\Lambda^{\rightarrow\ fixed}_{\mathcal{C}\widehat{tp}}$ *then* $(\Gamma_T; b \vdash M : A_b; b)^{tp}$ *in* $\Lambda^{\rightarrow -}_{\mathcal{C}}$ *for an atomic type* b.

The embedding is also sound with respect to the semantics. More precisely, each reduction step in $\lambda_{\mathcal{C}\widehat{tp}}$ maps to a reduction sequence of at least one step (written $\twoheadrightarrow^{\geq 1}$) in $\lambda^{-}_{\mathcal{C}}$ *up to* some renaming steps which are carried out by the $\mathcal{C}_{idem}$ rule. The reason for these renaming steps is due to the administrative redexes introduced by the translation as shown in the example below. Consider the following reduction in $\lambda_{\mathcal{C}\widehat{tp}}$:

$$\mathcal{C}(\lambda k.\,k\ N)\ M \to \mathcal{C}(\lambda k.\,k\ (N\ M)).$$

For simplicity we assume k does not occur free in N. The embedding of the left-hand side reduces to

$$\mathcal{C}(\lambda q.\,\mathbf{let}\ (f, tp') = N^{tp}\ \mathbf{in}\ q\ (f\ M^{tp'}))$$

whereas the embedding of the right-hand side is:

$$\mathcal{C}(\lambda k.\,k\ \mathcal{C}(\lambda q.\,\mathbf{let}\ (f, tp') = N^{tp}\ \mathbf{in}\ q\ (f\ M^{tp'}))).$$

Before stating the soundness we need a lemma indicating how the embedding behaves with respect to substitution. To that end, we define the following abbreviations:

$$\begin{aligned}(k\ (\Box\ M))^+ &= \mathbf{bind}_{k,tp}\ \Box\ \mathbf{in}\ M^{tp}\\ (k\ (V\ \Box))^+ &= k\ (V^+\ \Box)\\ (\widehat{tp}\ \Box)^+ &= \mathbf{join}\ \Box\end{aligned}$$

Lemma 19 *The following properties hold*:

(i) $M^{tp}[V^+/x] = (M[V/x])^{tp}$,
(ii) $(J[k\ (V\ \Box)/k])^{tp} = J^{tp}[(k\ (V\ \Box))^+/k]$,
(iii) $(J[q\ (\Box\ M)/k])^{tp} \twoheadrightarrow_{\mathcal{C}_{idem}} N \twoheadleftarrow J^{tp}[(q\ (\Box\ M))^+/k]$ *for some* N,
(iv) $(J[\widehat{tp}\ \Box/k])^{tp} = J^{tp}[(\widehat{tp}\ \Box)^+/k]$.

Proof The non-obvious part is item (iii). The translation of each subterm $k\,(N\,M)$ in $J[k\ (\Box\ M)/k]$ depends on whether N is a value or not. If N is some value V, then $k\,(V\,M)$ is translated to $k\,(V^+\,M^{tp})$ which is a contractum of $\mathbf{bind}_{k,tp}\ V^{tp}\ \mathbf{in}\ M^{tp}$. If N is not a value, the translation is $k\ \mathcal{C}(\lambda q.\,\mathbf{bind}_{q,tp'}\ N^{tp}\ \mathbf{in}\ M^{tp'})$ which reduces by $\mathcal{C}_{idem}$ to $\mathbf{bind}_{k,tp}\ N^{tp}\ \mathbf{in}\ M^{tp}$. □

Proposition 20 *If* $M \to N$ *in* $\lambda_{\mathcal{C}\widehat{tp}}$ *then there exists* N' *such that* $M^{tp} \twoheadrightarrow^{\geq 1} N'$ *in* $\lambda_{\mathcal{C}}^{-}$ *and* $N^{tp} \twoheadrightarrow_{\mathcal{C}_{idem}} N'$.

Proof We show one case of the proof:

Case $\mathcal{C}_L$:

$$
\begin{array}{ll}
 & (\mathcal{C}(\lambda k'.\,J)\ N)^{tp} \ =\ \mathcal{C}(\lambda q.\,\mathbf{bind}_{q,tp}\ \mathcal{C}(\lambda k'.\,J^{tp})\ \mathbf{in}\ N^{tp}) \\
\to_{Sub_{\forall}^{lift}} & \mathcal{C}(\lambda q.\,J^{tp}[\mathbf{let}\ (f, tp) = \Box\ \mathbf{in}\ q\ (f\ N^{tp})/k']) \\
= & \mathcal{C}(\lambda q.\,J^{tp}[(q\ (\Box\ N))^+/k']) \\
\to & P \\
\twoheadleftarrow_{\mathcal{C}_{idem}} & (\mathcal{C}(\lambda q.\,J[q\ (\Box\ N)/k']))^{tp} \quad \text{(by Lemma 19(iii))}
\end{array}
$$

□

Remark 21 In trying to find a better simulation of $\lambda_{\mathcal{C}\widehat{tp}}$, we have investigated alternative subtraction rules. In particular, the exact form to give to the elimination rule is not obvious in natural deduction (compared to its definition in sequent calculus [8]). For example, the rule:

$$\frac{\Gamma \vdash M : A - B \quad \Gamma, x : A, k : B \to \perp\!\!\!\perp \vdash M' : C}{\Gamma \vdash \mathbf{let}\ (x, k) = M\ \mathbf{in}\ M' : C}$$

that has been considered in a previous work [3] seems to be equally worthwhile. This however complicates the translation of $\widehat{tp}\,M$, which becomes $(tp\ (\mathbf{join}\,M^{tp}))$, where $\mathbf{join}\ M$ is now defined as $\mathbf{let}\ (x, k) = M\ \mathbf{in}\ \mathcal{T}h\ k\ x$. Due to the outer occurrence of tp, the translation is not stable under reduction. This introduces the need of reasoning up to the name of this extra continuation variable. Another solution is to reason in full classical logic with subtraction (i.e. $\lambda_{\mathcal{C}}^{-\perp}$) instead of minimal classical logic (i.e. $\lambda_{\mathcal{C}}^{-}$) and to define $\widehat{tp}\,M$ as $\mathsf{tp}_{\perp}\,(\mathbf{join}\,M^{tp})$, where $\mathsf{tp}_{\perp}$ witnesses the elimination rule of $\perp$ (see Intermezzo 15). We believe that a true simulation of the reduction would have been obtained with this approach.

As a corollary, we get the following result.

Proposition 22 *If* $M =_{\lambda_{\mathcal{C}\widehat{tp}}} N$ *then* $M^{tp} =_{\lambda_{\mathcal{C}}^{-}} N^{tp}$.

As an emphasis of how the translation works, the following example shows how to capture dynamic substitution of $\lambda_{\mathcal{C}\widehat{tp}}$.

Example 23 Consider Example 5 that was used to illustrate the capture of $\widehat{tp}$ during substitution:

$$\begin{array}{l}(\lambda x.\,(\lambda y.\,\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ (x\ y))))\ (\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y))\\ \rightarrow \lambda y'.\,\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ ((\lambda\_.\mathcal{C}(\lambda\_.\ \widehat{tp}\ y))\ y'))\end{array}$$

The embedding of the left-hand side is as follows:

$$\begin{array}{ll}((\lambda x.\,(\lambda y.\,\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ (x\ y))))\ (\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y)))^{tp} & =\\ (\lambda x.\,(\lambda y.\,\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ (x\ y))))^{+}\ (\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y))^{tp} & =\\ (\lambda(x,tp_1).(\lambda y.\,\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ (x\ y)))^{tp_1})\ ((\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y))^{+},tp) & =\\ ((\lambda(x,tp_1).((\lambda(y,tp_3).\mathcal{C}(\lambda\widehat{tp}.\,\widehat{tp}\ (x\ y))^{tp_3}),\lambda x.\,tp_1\ x))\\ \ ((\lambda\_.\mathcal{C}(\lambda\_.\widehat{tp}\ y))^{+},tp)) & =\\ ((\lambda(x,tp_1).(\lambda(y,tp_3).(\mathcal{C}(\lambda tp_4.\mathbf{join}\ (x\ (y,tp_4))),\lambda x.\,tp_3\ x),\lambda x.\,tp_1\ x))\\ \ ((\lambda(\_,tp_2).\mathcal{C}(\lambda\_.\mathbf{join}\ (y,\lambda x.\,tp_2\ x))),tp))\end{array}$$

This term reduces to:

$$\begin{array}{l}(\lambda(y',tp_3).(\mathcal{C}(\lambda tp_4.\mathbf{join}\ ((\lambda(\_,tp_2).\mathcal{C}(\lambda\_.\mathbf{join}\ (y,\lambda x.\,tp_2\ x)))\\ \qquad\qquad (y',tp_4))),\lambda x.\,tp_3\ x),tp)\end{array}$$

which is the embedding of the right-hand side.

The theory $\lambda^{-}_{\mathcal{C}}$ is richer than $\lambda_{\mathcal{C}\widehat{tp}}$ or $\lambda_{\mathcal{C}\#\mathsf{tp}}$, so that the translation is not complete.

Example 24 Consider the $\lambda_{\mathcal{C}\#\mathsf{tp}}$ term $y\ (\#\ (\mathsf{tp}\ x))$ which is mapped into the $\lambda^{-}_{\mathcal{C}}$ term $y\ (\mathcal{C}(\lambda tp.\ \mathbf{join}(x,tp),tp)$, which reduces as follows:

$$\begin{array}{rl}y\ (\mathcal{C}(\lambda tp.\ \mathbf{join}(x,tp)),tp) & \overset{Sub_v^{base}}{\rightarrow}\ y\ ((\mathcal{C}(\lambda tp.\ tp\ x),tp))\\ & \overset{\mathcal{C}_{lift}}{\rightarrow}\ y\ (\mathcal{C}(\lambda tp.\ tp\ (x,tp)))\\ & \overset{\mathcal{C}_{lift}}{\rightarrow}\ \mathcal{C}(\lambda tp.\ tp\ (y\ (x,tp)))\end{array}$$

The *lift* reductions move the # while maintaining the proper references to the top-level continuation. This move has no counterpart in either $\lambda_{\mathcal{C}\#\mathsf{tp}}$ or $\lambda_{\mathcal{C}\widehat{tp}}$. In $\lambda_{\mathcal{C}\widehat{tp}}$ such a lifting would cause the dynamically-bound top-level continuation $\widehat{tp}$ to be incorrectly captured; in $\lambda^{-}_{\mathcal{C}}$, the top-level continuation is statically-bound so that it is enough to rely on α-conversion to have a safe lifting rule for #. Pulling the lifting rule for # back from $\lambda^{-}_{\mathcal{C}}$ to $\lambda_{\mathcal{C}\widehat{tp}}$ is non trivial, if even possible.

6.2 Which type systems ensure termination?

We are now able to show that the type systems with effects ensure strong normalisation of the underlying λ-calculus while the type system with fixed type does not.

Proposition 25 (i) *If* $\Gamma;U\vdash M:A;T$ *in* $\Lambda^{\overrightarrow{eff}}_{\mathcal{C}\widehat{tp}}$ *or* $\Gamma\vdash M:A;T$ *in* $\Lambda^{\overrightarrow{eff_{eq}}}_{\mathcal{C}\widehat{tp}}$ *then* M *is strongly normalising.*

(ii) *If* $\Gamma\vdash M:A;T$ *in* $\Lambda^{\overrightarrow{fixed}}_{\mathcal{C}\widehat{tp}}$ *and* T *is atomic then* M *is strongly normalising.*

Proof The result follows from Propositions 11, 16, 17, 20 and the fact that $\mathcal{C}_{idem}$ commutes with the other reductions in the following way: if $M \to N$ and $M \to_{\mathcal{C}_{idem}} M'$ then there is N' such that $M' \to N'$ and $N \twoheadrightarrow_{\mathcal{C}_{idem}} N'$, unless $M \to_{\mathcal{C}_{elim}} N = M'$. But since $\mathcal{C}_{elim}$ cannot be applied infinitely many times in a row, any infinite reduction sequence typed in $\Lambda^{\to\ eff}_{\mathcal{C}_{\widehat{tp}}}$ can be mapped to an infinite sequence in $\Lambda^{\to -}_{\mathcal{C}}$. □

Strong normalisation does not generally hold in $\Lambda^{\to\ fixed}_{\mathcal{C}_{\widehat{tp}}}$ if the top-level type is not atomic. Indeed, if T is non atomic then we have to add the annotation T on the arrows of T itself. This requires a definition by fixpoint and the resulting recursive type can be used to type a fixpoint combinator as shown in Example 7.

7 CPS semantics

To complete our investigation, we present a CPS semantics for $\lambda^{-}_{\mathcal{C}}$ and a corresponding double-negation translation for $\Lambda^{\to -}_{\mathcal{C}}$.

7.1 Target CPS calculus

Before presenting the CPS translation from $\lambda^{-}_{\mathcal{C}}$ we first introduce the target language of the translation. This target language $\lambda^{\wedge}$ is an ordinary *call-by-name* λ-calculus with pairs. The semantics of $\lambda^{\wedge}$ is given using the following two rules:

$$\begin{aligned} \beta &: (\lambda x.M)N && \to M\,[N/x] \\ \wedge &: \mathbf{let}\ (x, y) = (M, N)\ \mathbf{in}\ M' && \to M'\,[N/y; M/x] \end{aligned}$$

We also use the following abbreviation:

$$\lambda(x, y).M \triangleq \lambda z.\,\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M \qquad \text{(Abbrev. 7)}$$

The type system of $\lambda^{\wedge}$, named $\Lambda^{\wedge}$, is an ordinary natural deduction system equipped with *tensorial conjunction*. This is a conjunction whose elimination rule extracts both components of the pair simultaneously, similar to what happens for the tensor product of linear logic. The tensorial conjunction will be used to model subtractions in $\lambda^{-}_{\mathcal{C}}$; its introduction and elimination rules are as follows:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \wedge B}\ \wedge_i$$

$$\frac{\Gamma \vdash M : A \wedge B \qquad \Gamma, x : A, y : B \vdash M' : C}{\Gamma \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ M' : C}\ \wedge_e$$

We assume the existence in $\Lambda^{\wedge}$ of a distinguished formula $\bot$ to which no inference rule is associated. We write $\neg A$ as an abbreviation of $A \to \bot$.

7.2 CPS translation of $\lambda^{-}_{\mathcal{C}}$

The CPS translation maps the types, judgements, and terms of $\lambda^{-}_{\mathcal{C}}$ to $\lambda^{\wedge}$. The translation we present extends the call-by-value CPS translation of Fischer [21, 22]. We give the translation in Fig. 22 for the types and judgements and in Fig. 23 for the terms and the contexts. We have:

Fig. 22 Fischer-style call-by-value $\neg\neg$-translation on $\Lambda_{\mathcal{C}}^{\rightarrow -}$

$$
\begin{array}{ll}
b^* & = b \\
(A \to B)^* & = \neg B^* \to \neg A^* \\
(A - B)^* & = \neg B^* \wedge A^* \\
 & \\
(\cdot)^* & = \cdot \\
(\Gamma, x : A)^* & = \Gamma^*, x : A^* \\
(\Gamma, k : A \to \perp\!\!\!\perp)^* & = \Gamma^*, k : \neg A^* \\
 & \\
(\Gamma \vdash M : A)^k & = \Gamma^*, k : \neg A^* \vdash [\![M]\!]\, k : \perp \\
(\Gamma \vdash J : \perp\!\!\!\perp)^* & = \Gamma^* \vdash [\![J]\!] : \perp
\end{array}
$$

$$
\begin{array}{ll}
x^+ & = x \\
(\lambda x.M)^+ & = \lambda k'.\lambda x.[\![M]\!]\, k' \\
(V, \lambda x.\, J_c[E[x]])^+ & = ([\![E]\!]\,[\![J_c]\!], V^+) \\
 & \\
[\![V]\!]\, k & = k\ V^+ \\
[\![V\ N]\!]\, k & = [\![N]\!]\,(V^+\ k) \\
[\![M\ N]\!]\, k & = [\![M]\!]\,(\lambda f.[\![N]\!]\,(f\ k)) \quad (M \text{ not value}) \\
 & \\
[\![\mathcal{C}(\lambda k'.\, J)]\!]\, k & = [\![J]\!][k/k'] \\
[\![(M, \lambda x.\, J_c[E[x]])]\!]\, k & = [\![M]\!]\,(\lambda v.\, k\ ([\![E]\!]\,[\![J_c]\!], v)) \quad (M \text{ not value}) \\
 & \\
[\![\textbf{let}\ (x,k) = M\ \textbf{in}\ J]\!] & = [\![M]\!]\,(\lambda(k,x).[\![J]\!]) \\
[\![k\ M]\!] & = [\![M]\!]\, k \\
 & \\
[\![\Box\ N]\!]\, k & = \lambda f.[\![N]\!]\,(f\ k) \\
[\![V\ \Box]\!]\, k & = V^+\, k \\
[\![(\Box, \lambda x.\, J_c[E[x]])]\!]\, k & = \lambda v.\, k\ ([\![E]\!]\,[\![J_c]\!], v) \\
 & \\
[\![E[F]]\!]\, k & = [\![F]\!]\,([\![E]\!]\, k) \\
[\![\Box]\!]\, k & = k \\
 & \\
[\![k\,\Box]\!] & = k \\
[\![\textbf{let}\ (x,k) = \Box\ \textbf{in}\ J]\!] & = \lambda(k,x).[\![J]\!]
\end{array}
$$

Fig. 23 Fischer-style call-by-value CPS translation of $\lambda_{\mathcal{C}}^{-}$

Proposition 26 *The CPS translation is sound with respect to typing*:

– *If* $\Gamma \vdash M : A$ *in* $\Lambda_{\mathcal{C}}^{\rightarrow -}$ *then* $(\Gamma \vdash M : A)^k$ *in* $\Lambda^\wedge$.
– *If* $\Gamma \vdash J : \perp\!\!\!\perp$ *in* $\Lambda_{\mathcal{C}}^{\rightarrow -}$ *then* $(\Gamma \vdash J : \perp\!\!\!\perp)^*$ *in* $\Lambda^\wedge$.

Before proving the soundness of the translation with respect to reductions, we first state a few useful properties.

Lemma 27

(i) $[\![M]\!]\,([\![F]\!]\, k) = [\![F[M]]\!]\, k$ *if M not a value*

(ii) $[\![V]\!]\,([\![F]\!]\, k) \twoheadrightarrow [\![F[V]]\!]\, k$

$$
\begin{array}{lrcl}
\text{(iii)} & [\![M]\!]\,[\![J_c]\!] & = & [\![J_c[M]]\!] \\
\text{(iv)} & [\![M[V/x]]\!]\,k & = & ([\![M]\!]\,k)[V^+/x] \\
\text{(v)} & [\![J[V/x]]\!] & = & [\![J]\!][V^+/x] \\
\text{(vi)} & [\![F[E]]\!]\,k & = & [\![E]\!]\,([\![F]\!]\,k) \\
\text{(vii)} & [\![J]\!]\,[[\![F]\!]\,k'/k] & \twoheadrightarrow & [\![J[k'\ F/k]]\!] \\
\text{(viii)} & [\![F[\mathcal{C}(\lambda k.\,J)]]\!]\,k' & \twoheadrightarrow & [\![J[k'\ F/k]]\!] \\
\text{(ix)} & [\![J]\!][[\![J_c]\!]/k] & = & [\![J[J_c/k]]\!] \\
\text{(x)} & [\![J_c[\mathcal{C}(\lambda k.\,J)]]\!] & = & [\![J[J_c/k]]\!]
\end{array}
$$

Proof Items (i), (iii), (iv) and (v) can be easily checked. Item (ii) introduces a reduction for the contexts $\Box\ N$ and $(\lambda x.\,J_c[E[x]], \Box)$. Items (vi) is by induction on the length of E. For item (vii), many cases can occur. If k occurs at the head of the second component of a subtractive term, the substitution leaves the term unchanged because of item (vi). If k occurs elsewhere, then it may be applied to a value or not and either (i) or (ii) applies. For item (viii), the result follows from (i) and (vii). Item (ix) follows because k is substitutive when it occurs at the head of the second component of a subtractive term and by item (iii) otherwise. For item (x), the result follows from (iii) and (ix). □

Proposition 28 *The CPS translation is sound with respect to the $\lambda^-_{\mathcal{C}}$ semantics*:

(i) *If $M \to M'$ in $\lambda^-_{\mathcal{C}}$ then $[\![M]\!]\,k =_{\beta,\wedge} [\![M']\!]\,k$ in $\lambda^\wedge$;*
(ii) *If $J \to J'$ in $\lambda^-_{\mathcal{C}}$ then $[\![J]\!] =_{\beta,\wedge} [\![J']\!]$ in $\lambda^\wedge$.*

Proof We consider a few of the cases:

– Sub_v^{lift} : From Lemma 27(x), we directly have:

$$[\![\mathbf{let}\ (x,q) = \mathcal{C}(\lambda k.\,J')\ \mathbf{in}\ J]\!] = [\![J'[\mathbf{let}\ (x,q) = \Box\ \mathbf{in}\ J/k]]\!].$$

– Sub_v^{base} : The left-hand side is:

$$
\begin{array}{rll}
[\![\mathbf{let}\ (x,q) = (V,k')\ \mathbf{in}\ J]\!] = & [\![(V,k')]\!]\,(\lambda(q,x).[\![J]\!]) \\
\twoheadrightarrow_{\beta_v} & (\mathbf{let}\ (q,x) = (k',V^+)\ \mathbf{in}\ [\![J]\!] \\
\to_\wedge & [\![J]\!]\,[k'/q;\,V^+/x]
\end{array}
$$

The right-hand side is $[\![J[k'/q;\,V/x]]\!]$ and the result follows from Lemma 27(v, x).

– Sub_v^{step} : The left-hand side is:

$$
\begin{array}{ll}
[\![\mathbf{let}\ (x,k') = (V,\lambda x.\,J_c[E[F[x]]])\ \mathbf{in}\ J]\!] & = \\
(\lambda k.k\,([\![F]\!]\,([\![E]\!]\,[\![J_c]\!]),V^+))(\lambda(k',x).[\![J]\!]) & \twoheadrightarrow_{\beta_v} \\
\mathbf{let}\ (k',x) = ([\![F]\!]\,([\![E]\!]\,[\![J_c]\!]),V^+)\ \mathbf{in}\ [\![J]\!] & \to_\wedge \\
[\![J]\!][[\![F]\!]\,([\![E]\!]\,[\![J_c]\!])/k';\,V^+/x] & = \\
[\![J]\!][[\![F]\!]\,k''/k'][[\![E]\!]\,[\![J_c]\!]/k'';\,V^+/x] & \text{for } k'' \text{ fresh}
\end{array}
$$

For the right-hand side we have:

$$
\begin{array}{l}
[\![\mathbf{let}\ (x,k') = (V,\lambda x.\,J_c[E[x]])\ \mathbf{in}\ J[k'\ F/k']]\!] \twoheadrightarrow_{\beta_v} \\
\mathbf{let}\ (k',x) = ([\![E]\!]\,[\![J_c]\!],V^+)\ \mathbf{in}\ [\![J[k'\ F/k']]\!]\ \to_\wedge \\
([\![J[k'\ F/k']]\!])[[\![E]\!][\![J_c]\!]/k';\,V^+/x]
\end{array}
$$

We conclude by Lemma 27(vii). □

7.3 Induced CPS translation of $\lambda_{\mathcal{C}\#\mathsf{tp}}$

As $\lambda_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda_{\mathcal{C}\widehat{tp}}$ are isomorphic, we can compose the embeddings from $\lambda_{\mathcal{C}\widehat{tp}}$ to $\lambda_{\mathcal{C}}^{-}$ with the CPS translation from $\lambda_{\mathcal{C}}^{-}$ to $\lambda^{\wedge}$ to produce a CPS translation for $\lambda_{\mathcal{C}\#\mathsf{tp}}$. The result is given in Figs. 24 and 25 (obvious redexes have been contracted), where $\Lambda_{\mathcal{C}\mathsf{tp}}^{\rightarrow\ eff}$ stands for the type-and-effect system of $\lambda_{\mathcal{C}\#\mathsf{tp}}$. This is obtained from the system of Fig. 13 by replacing the $RAA^{\widehat{tp}}$ and $\rightarrow_{e}^{\widehat{tp}}$ with the following typing rules:

$$\frac{\Gamma \vdash J : \perp\!\!\!\perp ; A}{\Gamma ; T \vdash \# J : A ; T} \qquad \frac{\Gamma ; U \vdash M : U ; T}{\Gamma \vdash \mathsf{tp}\ M : \perp\!\!\!\perp ; T}$$

The $\lambda_{\mathcal{C}\#\mathsf{tp}}$ CPS translation shows how such a translation for delimited continuations can be decomposed into a store-passing transformation followed by a standard CPS transformation into the pure λ-calculus. The first transformation deals with the control delimiter; the second transformation deals with the control operator. By putting together Propositions 6, 22 and 28, one then concludes that the CPS semantics preserves the notions of convertibility of $\lambda_{\mathcal{C}\#\mathsf{tp}}$.

Proposition 29 *Given $\lambda_{\mathcal{C}\#\mathsf{tp}}$ terms M and N. If $M = N$ in $\lambda_{\mathcal{C}\#\mathsf{tp}}$ then $[\![M]\!]\, k\, tp =_{\beta,\wedge} [\![N]\!]\, k\, tp$.*

$$\begin{array}{ll}
b^{*} & = b \\
(A\ {}_{U}{\rightarrow}_{T}\ B)^{*} & = \neg(\neg U^{*} \wedge B^{*}) \rightarrow \neg(\neg T^{*} \wedge A^{*}) \\
(A \rightarrow_{U} \perp\!\!\!\perp)^{*} & = \neg(\neg U^{*} \wedge A^{*}) \\
 & \\
(\Gamma ; B \vdash M : A ; C)_{k}^{tp} & = \Gamma^{*}, tp : \neg C^{*}, k : \neg(\neg B^{*} \wedge A^{*}) \vdash [\![M]\!]\, k\, tp : \bot \\
(\Gamma \vdash J : \perp\!\!\!\perp ; C)^{tp} & = \Gamma^{*}, tp : \neg C^{*} \vdash [\![J]\!]\, tp : \bot
\end{array}$$

Fig. 24 Derived call-by-value $\neg\neg$-state-transformation on $\Lambda_{\mathcal{C}\mathsf{tp}}^{\rightarrow\ eff}$

$$\begin{array}{lll}
x^{+} & = x & \\
(\lambda x.\, M)^{+} & = \lambda k.\, \lambda(tp, x).\, [\![M]\!]\, k\, tp & \\
 & & \\
[\![V]\!]\, k\, tp & = k\ (tp, V^{+}) & \\
[\![V\ N]\!]\, k\, tp & = [\![N]\!]\ (V^{+}\ k)\ tp & \\
[\![M\ N]\!]\, k\, tp & = [\![M]\!]\ (\lambda(tp', f).\, [\![N]\!]\ (f\ k)\ tp')\ tp & M \text{ not a value} \\
[\![\mathcal{C}(\lambda k'.\, J)]\!]\, k\, tp & = ([\![J]\!]\ tp)[k/k'] & \\
[\![\# J]\!]\, k\, tp & = [\![J]\!]\ \lambda v.\, k\ (tp, v) & \\
[\![\mathsf{tp}\ M]\!]\ tp & = [\![M]\!]\ (\lambda(k, x).\, k\ x)\ tp & \\
[\![k\ M]\!]\ tp & = [\![M]\!]\, k\, tp &
\end{array}$$

Fig. 25 Derived call-by-value CPS translation of $\lambda_{\mathcal{C}\#\mathsf{tp}}$

7.4 The operators *shift* and *reset*

Based on the embeddings of *shift* and *reset* given in Remark 3, their translations in $\lambda^{\wedge}$ are:

$$\begin{aligned} [\![\langle M\rangle]\!]\, k\, tp &= [\![M]\!]\, I'\, (\lambda v.\, k\, (tp, v)) \\ [\![\mathcal{S}\ (\lambda q.\ M)]\!]\, k\, tp &= [\![M]\!]\, [reifyS\ k/q]\ I'\ tp \end{aligned}$$

$$\begin{aligned} \text{where } I' &= \lambda(k, x).\, k\, x \\ reifyS\ k &= \lambda k'.\, \lambda(tp, x).\, k\ (\lambda y.\, k'\ (tp, y), x) \end{aligned}$$

which correspond, up to currying and η-conversion, to the definition of *shift* and *reset* in the original paper by Danvy and Filinski [9].

Let $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ be the subset of $\lambda_{\mathcal{C}\#\mathsf{tp}}$ without continuation variables and where every occurrence of tp and $\mathcal{C}$ only occurs as part of the definitions of $\mathcal{S}$ and $\langle\_\rangle$, but still equipped with the same equational theory as $\lambda_{\mathcal{C}\#\mathsf{tp}}$. What we have shown is that the following diagram commutes:

Theorem 30

$$\begin{array}{ccc} \lambda'_{\mathcal{C}\#\mathsf{tp}} & \xrightarrow{[\![\_]\!]'\,k} & \lambda_v \\ \downarrow \_\ tp & & \downarrow [\![\_]\!]\ tp \\ \lambda_{\mathcal{C}}^{-} & \xrightarrow{\mathsf{cur} \circ [\![\_]\!]\,k} & \lambda \end{array}$$

where

- λ_v *is Plotkin's call-by-value* λ*-calculus and* λ *is usual* λ*-calculus,*
- cur *is the currying combinator,*
- $[\![\_]\!]'\,k$ *is the same as* $[\![\_]\!]\,k$ *from Fig. 23 but extended with the clauses*

$$\begin{aligned} [\![\#\ J]\!]'\, k &= k\ [\![J]\!]' \\ [\![\mathsf{tp}\ M]\!]' &= [\![M]\!]'\ \lambda x.x \end{aligned}$$

  so that the "macros" $[\![\langle M\rangle]\!]'\,k$ *and* $[\![\mathcal{S}(\lambda q.\, M)]\!]'\,k$ *are translated as in* [10],
- $[\![\_]\!]\ tp$ *is as in Fig. 23 but written with the name tp in place of* k*, so that the up and right sides of the square correspond to the two-steps CPS translation from the syntax of* $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ *to* λ [10].

7.5 Completeness of the CPS translation

Thanks to Kameyama and Hasegawa [35] axiomatisation of $\lambda'_{\mathcal{C}\#\mathsf{tp}}$, we can give a proof of completeness of our translation. Let $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$ be the extension of $=_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$ obtained by adding the following reductions:

$$\begin{aligned} \#_{idem} :\ & \mathsf{tp}\ (\#J) && \to' J && \\ \#_{tail} :\ & \#\ (l\ ((\lambda x.M)\ (\#\ J))) && \to' (\lambda x.\#\ (l\ M))\ (\#\ J) && \\ \mathcal{C}_{tail} :\ & (\lambda x.\mathcal{C}(\lambda k.l\ M))\ N && \to' \mathcal{C}(\lambda k.l\ ((\lambda x.M)\ N)) && \\ \eta_v :\ & \lambda x.(V\ x) && \to' V && \text{where } x \notin FV(V) \\ \beta_\Omega :\ & (\lambda x.E[x])\ M && \to' E[M] && \text{where } x \notin FV(E) \end{aligned}$$

where l is either k or tp, F is either $V\ \Box$ or $\Box\ N$ and E is either $\Box$ or $E[F]$. The following equations (respectively consequences of β_Ω and $\#_{idem}$ conjugated with $\mathcal{C}_{idem}$ and $\mathcal{C}_{tail}$, of $\mathcal{C}_{tail}$, $\mathcal{C}_{idem}$ and β_Ω, and of β_Ω and β_v) will be useful:

$$\begin{array}{lll} \#_{let}: & \mathsf{tp}\ ((\lambda x.\#\ (k\ x))\ M) & ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}\ k\ M \\ \mathcal{T}h_{let}: & \mathsf{tp}\ ((\lambda x.\mathcal{T}h\ (k\ x))\ M) & ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}\ k\ M \\ let_{lift}: & F[(\lambda x.N)\ M] & ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}\ (\lambda x.F[N])\ M\,. \end{array}$$

Kameyama and Hasegawa's axioms of $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ are listed below. Since $\mathcal{S}$ is a stand-alone constant in Kameyama and Hasegawa [35], we need to slightly adapt their axiomatic[1]:

$$\begin{array}{lllll} \beta_v: & (\lambda x.M)\ V & =_{KH}\ M\ [V/x] & & \\ \mathcal{S}\text{-}reset: & \langle E[\mathcal{S}(\lambda k.\,M)]\rangle & =_{KH}\ \langle(\lambda k.M)\lambda x.\langle E[x]\rangle\rangle & & \\ \mathcal{S}\text{-}elim: & \mathcal{S}(\lambda k.\,k\ M) & =_{KH}\ M & \text{where } k \notin FV(M) & \\ \mathcal{S}\text{-}tail: & (\lambda x.\mathcal{S}(\lambda k.\,M))\ N & =_{KH}\ \mathcal{S}(\lambda k.\,(\lambda x.M)\ N) & & \\ reset\text{-}value: & \langle V\rangle & =_{KH}\ V & & \\ reset\text{-}lift: & \langle(\lambda x.M)\ \langle N\rangle\rangle & =_{KH}\ (\lambda x.\langle M\rangle)\ \langle N\rangle & & \\ reset\text{-}\mathcal{S}: & \mathcal{S}(\lambda k.\,\langle M\rangle) & =_{KH}\ \mathcal{S}(\lambda k.\,M) & & \\ \eta_v: & \lambda x.(V\ x) & =_{KH}\ V & \text{where } x \notin FV(V) & \\ \beta_\Omega: & (\lambda x.E[x])\ M & =_{KH}\ E[M] & \text{where } x \notin FV(E) & \end{array}$$

On $\lambda'_{\mathcal{C}\#\mathsf{tp}}$, Kameyama and Hasegawa's axiomatisation is derivable from the equational theory $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$:

Proposition 31 *Let M and N be in $\lambda'_{\mathcal{C}\#\mathsf{tp}}$. If $M =_{KH} N$ then $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N$.*

Proof The rules β_v, η_v, β_Ω and *reset-value* are also in $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$; $\mathcal{S}$-*reset* is a combination of $\mathcal{C}_L$, $\mathcal{C}_R$ and $\mathcal{C}_{idem}$; $\mathcal{S}$-*elim* is a consequence of $\mathcal{C}_{elim}$ using first β_v, $\#_{let}$ and $\#_{idem}$; $\mathcal{S}$-*tail* is a consequence of $\mathcal{C}_{tail}$ and let_{lift}; *reset-lift* derives from $\#_{tail}$ for l being tp; finally, *reset-*$\mathcal{S}$ derives from $\#_{idem}$ conjugated with β_v. □

Now, since Kameyama and Hasegawa's axiomatisation is complete with respect to the CPS translation,[2] we get ($\eta_\wedge$ is the equation $M = (\mathbf{let}\ (x, y) = M\ \mathbf{in}\ x, \mathbf{let}\ (x, y) = M\ \mathbf{in}\ y)$):

Corollary 32 *Given $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ terms M and N, if $[\![M]\!]\ k\ tp =_{\beta,\eta,\wedge,\eta_\wedge} [\![N]\!]\ k\ tp$ then $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N$.*

[1] The original system has the rule $\mathcal{S}$-*reset*: $\langle E[\mathcal{S}\ M]\rangle =_{KH} \langle M\lambda x.\langle E[x]\rangle\rangle$ and no rule $\mathcal{S}$-*tail*. The full strength of the original $\mathcal{S}$-*reset* is not expressible in $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ since $\mathcal{S}$ in there is constrained to appear with the form $\mathcal{S}(\lambda k.M)$. The original $\mathcal{S}$-*reset* implies $\mathcal{S}_\eta: \mathcal{S}\ M = \mathcal{S}(\lambda k.M\ k)$, so, obviously, it is equivalent to the combination of $\mathcal{S}_\eta$ and of the modified $\mathcal{S}$-*reset*. Thanks to β_Ω and η_v, $\mathcal{S}_\eta$ is provably equivalent to $\mathcal{S}$-*tail*, hence the modified system is equivalent to the original one on $\lambda'_{\mathcal{C}\#\mathsf{tp}}$. We leave open the question whether $\mathcal{S}$-*tail* is not redundant in the modified system.

[2] The CPS translation in [35] is the composition of the Plotkin-style variants of the Fisher-style call-by-value translations $[\![\_]\!]'\ k$ and $[\![\_]\!]_{tp}$. Both styles of translation are isomorphic. By Theorem 30, they are also isomorphic to $[\![\_]\!]\ k\ tp$ and the equations are the same.

We then show that any term M in $\lambda_{\mathcal{C}\#\mathsf{tp}}$ has a representative in $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ that behaves the same with respect to $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$. Let $\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}$ be the subset of $\lambda_{\mathcal{C}\#\mathsf{tp}}$ with no free continuation variables. The following embedding, that relies on the definition of $\mathcal{S}$ and $\langle\,\rangle$ in Remark 3, is a map from $\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}$ to $\lambda'_{\mathcal{C}\#\mathsf{tp}}$:

$$\begin{array}{ll} x^\bullet & = x \\ (\lambda x.\, M)^\bullet & = \lambda x.\, M^\bullet \\ (M\ N)^\bullet & = M^\bullet\ N^\bullet \\ (\mathcal{C}(\lambda k.\, J))^\bullet & = \mathcal{S}(\lambda k.\, J^\bullet) \\ (\#\ J)^\bullet & = \langle J^\bullet \rangle \end{array} \qquad \begin{array}{l} (\mathsf{tp}\, M)^\bullet = M^\bullet \\ (k\, M)^\bullet\ = k\ M^\bullet \end{array}$$

Notice that $(k\ M)^\bullet$, when in position of subterm, is necessarily surrounded by tp. Thanks to the following equation:

$$\begin{array}{rll} (\mathcal{C}(\lambda k.\, J))^\bullet = & & \mathcal{C}(\lambda k.\mathsf{tp}\ ((\lambda k.J^\bullet)\ \lambda x.\#\ (k\ x))) \\ & =_{\beta_v} & \mathcal{C}(\lambda k.\mathsf{tp}\ J^\bullet[\lambda x.\#\ (k\ x)/k]) \\ & = & \mathcal{C}(\lambda k.\mathsf{tp}\ J[\mathsf{tp}\ (\lambda x.\#\ (k\ x))\ \Box/k]^\bullet) \\ & =_{\#_{let}} & \mathcal{C}(\lambda k.\mathsf{tp}\ J^\bullet) \end{array}$$

we get:

Proposition 33 *For all M in $\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}$, $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} M^\bullet$.*

Now, take M in $\lambda_{\mathcal{C}\#\mathsf{tp}}$. We embed M into $\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}$ by replacing each free continuation variable k in M by the context $\mathsf{tp}\ (y_k\ \Box)$ for y_k a fresh variable associated to k. Let ρ be this substitution and ρ' the substitution that maps each y_k to $\lambda x.\mathcal{T}h\ k\ x$. The term $(M[\rho])^\bullet$ is in $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ and, using $\mathcal{T}h_{let}$, we can show that $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} (M[\rho])^\bullet[\rho']$.

A simple verification shows that the equation $\#_{let}$ used in the proof of Proposition 33 is sound with respect to $[\![\ ]\!]\, k\, tp$. Moreover, Fig. 23 shows that $[\![\ ]\!]\, k\, tp$ is stable by substitution of continuation variables. Hence we get:

Theorem 34 (Completeness) *Given $\lambda_{\mathcal{C}\#\mathsf{tp}}$ terms M and N, from $[\![M]\!]\, k\, tp =_{\beta,\eta,\wedge,\eta_\wedge} [\![N]\!]\, k\, tp$, we get $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N$.*

7.6 More on *shift* and *reset*

We now make the connection with Kameyama and Hasegawa axiomatics more precise. Let $\lambda_{\mathcal{S}}$ be the language defined by

$$M ::= x \mid \lambda x.M \mid M\ M \mid \mathcal{S}(\lambda k.M) \mid \langle M \rangle,$$

where $\mathcal{S}$ and $\langle\,\rangle$ are primitive symbols. Clearly, passing from $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ to λ_S is just a move from macro-defined $\mathcal{S}$ and $\langle\,\rangle$ to primitive $\mathcal{S}$ and $\langle\,\rangle$. We can check that all equations in $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$ not already checked valid for $=_{\beta,\eta,\wedge,\eta_\wedge}$ through the CPS translation are so. Hence, we get:

Theorem 35 (Isomorphism between $\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda_{\mathcal{S}}$) *We have the following chain of isomorphisms*:

$$(\lambda^{-}_{\mathcal{C}\#\mathsf{tp}}, ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}) \simeq (\lambda'_{\mathcal{C}\#\mathsf{tp}}, ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}) \simeq (\lambda'_{\mathcal{C}\#\mathsf{tp}}, =_{KH}) \simeq (\lambda_{\mathcal{S}}, =_{KH}).$$

Proof The first isomorphism is by Proposition 33. The second is by Proposition 31 and by composition of CPS soundness of $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$ and CPS completeness of $=_{KH}$. □

To extend the isomorphism to $\lambda_{\mathcal{C}\#\mathsf{tp}}$, that has free continuation variables, we need to extend $\lambda_{\mathcal{S}}$ with one new constant c_k for each continuation variable k in $\lambda_{\mathcal{C}\#\mathsf{tp}}$. If each of this constant were constrained to occur in contexts of the form $\mathcal{S}(\lambda k'.\, c_k\ M)$ or $\#\ (k\ M)$, we would directly get a correspondence with $\lambda_{\mathcal{C}\#\mathsf{tp}}$. If we otherwise allow these constants to occur anywhere in terms, then we need to assign a default semantics (abortive or functional) to them.

Let $\lambda^+_{\mathcal{S}}$ be the extension of $\lambda_{\mathcal{S}}$ with the constants c_k. To assign an abortive semantics, we extend $=_{KM}$ with the parametric equation $C_{\mathcal{A}} : c_k = \lambda x.\mathcal{S}(\lambda\_.\, c_k\ x)$. We also extend $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ to allow free occurrences of continuation variables subject to the proviso that they occur in subterms of the form $\lambda x.\mathcal{T}h\ k\ x$. Let $\lambda'^{+\mathcal{A}}_{\mathcal{C}\#\mathsf{tp}}$ be this extension. The interpretation of $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ as a calculus of macro-definitions for $\lambda_{\mathcal{S}}$ extends to $\lambda'^{+\mathcal{A}}_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda^+_{\mathcal{S}}$ by setting $c_k = \lambda x.\mathcal{T}h\ k\ x$. We extend $^\bullet$ on $\lambda_{\mathcal{C}\#\mathsf{tp}}$ by setting $(k\ M)^\bullet = (\lambda x.\mathcal{T}h\ k\ x)\ M^\bullet$ for k free. Because $(k\ M)^\bullet$ always occurs surrounded by tp in the translation of terms, we have $k\ M = \mathsf{tp}\ ((\lambda x.\mathcal{T}h\ k\ x)\ M)$ by $\mathcal{T}h_{let}$. Hence $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} M^\bullet$ still holds and $^\bullet$ is an isomorphism. Expressed in $\lambda'^{+\mathcal{A}}_{\mathcal{C}\#\mathsf{tp}}$, $C_{\mathcal{A}}$ is derivable thanks to β_v and $\mathcal{C}_{idem}$.

Alternatively, we can assign a functional semantics by replacing $\lambda x.\mathcal{T}h\ k\ x$ by $\lambda x.\#\ k\ x$ in the constructions above. We call $\lambda'^{+\#}_{\mathcal{C}\#\mathsf{tp}}$ the image of $\lambda^+_{\mathcal{S}}$ where c_k is interpreted as $\lambda x.\#\ k\ x$. The new equation on $\lambda^+_{\mathcal{S}}$ is $C_{\#} : c_k = \lambda x.\#\ (c_k\ x)$. The equation $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} M^\bullet$ holds thanks to $\#_{let}$ and $C_{\#}$ holds in $\lambda'^{+\#}_{\mathcal{C}\#\mathsf{tp}}$ by β_v and $\#_{idem}$.

Theorem 36 (Isomorphism between $\lambda_{\mathcal{C}\#\mathsf{tp}}$ and $\lambda^+_{\mathcal{S}}$) *We have the following chain of isomorphisms*:

$$\begin{aligned}(\lambda_{\mathcal{C}\#\mathsf{tp}}, ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}) &\simeq (\lambda'^{+A}_{\mathcal{C}\#\mathsf{tp}}, ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}) \simeq (\lambda'^{+A}_{\mathcal{C}\#\mathsf{tp}}, =_{KH+C_{\mathcal{A}}}) \simeq (\lambda^+_{\mathcal{S}}, =_{KH+C_{\mathcal{A}}})\\ &\simeq (\lambda'^{+\#}_{\mathcal{C}\#\mathsf{tp}}, ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}) \simeq (\lambda'^{+\#}_{\mathcal{C}\#\mathsf{tp}}, =_{KH+C_{\#}}) \simeq (\lambda^+_{\mathcal{S}}, =_{KH+C_{\#}}).\end{aligned}$$

Proof It remains to prove that from $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N$ in $\lambda'^{+\mathcal{A}}_{\mathcal{C}\#\mathsf{tp}}$, we can get $M =_{KH+C_{\mathcal{A}}} N$ and similarly for $\lambda'^{+\#}_{\mathcal{C}\#\mathsf{tp}}$ and $=_{KH+C_{\#}}$. Both cases are similar and we treat only the first one. We proceed as in the proof of Theorem 34. Let ρ be the substitution that maps each free continuation variables k in M and N to $\mathsf{tp}\ (y_k\ \square)$ and ρ' be the substitution that maps each y_k to c_k (i.e. to $\lambda x.\mathcal{T}h\ k\ x$). From $M ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N$, we get $M[\rho] ='_{\lambda_{\mathcal{C}\#\mathsf{tp}}} N[\rho]$. By composition on $\lambda'_{\mathcal{C}\#\mathsf{tp}}$ of CPS soundness of $='_{\lambda_{\mathcal{C}\#\mathsf{tp}}}$ and CPS completeness of $=_{KH}$, we get $M[\rho] =_{KH} N[\rho]$ and hence $M[\rho][\rho'] =_{KH} N[\rho][\rho']$. It remains to connect $M[\rho][\rho']$ to M and $N[\rho][\rho']$ to N. Since the free occurrences of k in M and N occur as part of subterms of the form $\lambda x.\mathcal{T}h\ k\ x$, this amounts to connect c_k to $\lambda x.\mathcal{T}h\ \mathsf{tp}\ (c_k\ x)$. This is a direct consequence of $C_{\mathcal{A}}$ and β_v. □

Example 37 Let $M = \#\ k\ (f\ 1) + \#\ k\ (\#\ \mathsf{tp}\ (g\ 1)) + \mathcal{T}h\ k\ (h\ 1)$. The image of M by $^\bullet$ is $\#\ (c_k\ (f\ 1)) + \#\ (c_k\ \#\ (g\ 1)) + \mathcal{S}(\lambda\_.\, c_k\ (h\ 1))$: at this point, the interpretation of c_k does not matter since we have $\mathsf{tp}\ ((\lambda x.\mathcal{T}h\ k\ x)\ N) =_{\mathcal{T}h_{let}} k\ N =_{\#_{let}} \mathsf{tp}\ ((\lambda x.\#\ k\ x)\ N)$.

In $\lambda'^{+\mathcal{A}}_{\mathcal{C}\#\mathsf{tp}}$ and in $\lambda^+_{\mathcal{S}}$ equipped with $=_{KH+C_{\mathcal{A}}}$, $M^\bullet$ can be shortened to $\#\ (c_k\ (f\ 1)) + \#\ (c_k\ \#\ (g\ 1)) + c_k\ (h\ 1)$: the abortive semantics of $\mathcal{S}(\lambda\_.\ )$ can be left implicit because $\mathcal{C}_{tail}$ holds.

In $\lambda^{'+\#}_{\mathcal{C}\#\mathsf{tp}}$ and in $\lambda^{+}_{\mathcal{S}}$ equipped with $=_{KH+C\#}$, $M^{\bullet}$ can be shortened to $\#\ (c_k\ (f\ 1))\ +\ c_k\ \#\ (g\ 1)\ +\ \mathcal{S}(\lambda\_.\, c_k\ (h\ 1))$: the second # has been removed thanks to $\#_{tail}$ but the first one cannot be hidden in general.

Conversely, the expression $c_k\ (f\ 1)\ +\ \#\ c_k\ (g\ 1)$ in $\lambda^{+}_{\mathcal{S}}$ is not explicit enough to express the semantics of the first occurrence of c_k. With an abortive semantics, it will be equivalent in $\lambda_{\mathcal{C}\#\mathsf{tp}}$ to the expression $\mathcal{T}h\ k\ (f\ 1)\ +\ \#\ k\ (g\ 1)$ while with a functional semantics, it will be equivalent to $(\lambda x.\#\ k\ x)\ (f\ 1)\ +\ \#\ k\ (g\ 1)$. Note again that $(\lambda x.\#\ k\ x)\ (f\ 1)$ does not a priori simplify into $\#\ k\ (f\ 1)$ while $(\lambda x.\mathcal{T}h\ k\ x)\ (f\ 1)$ does simplify to $\mathcal{T}h\ k\ (f\ 1)$. Also, both $\#\ \mathsf{tp}\ (\lambda x.\#\ k\ x)\ (g\ 1)$ and $\#\ \mathsf{tp}\ (\lambda x.\mathcal{T}h\ k\ x)\ (g\ 1)$ do simplify to $\#\ k\ (g\ 1)$.

8 Conclusions

We have focused exclusively on delimited continuations obtained with $\mathcal{C}$ and # (or equivalently *shift* and *reset*). We briefly review and classify some of the other control operators in the literature and discuss them based on our work. We also discuss the question of interpreting delimited continuations along the lines of the Curry-Howard correspondence.

8.1 A short history

Early proposals for delimited continuations had only a *single* control delimiter [9, 13, 16]. The control operation for capturing the continuation implicitly refers to the most recent occurrence of this delimiter.

After the limitations of the single control delimiter became apparent, later proposals generalised the single delimiter by allowing hierarchies of delimiters and control operators like $reset_n$ and $shift_n$ [10, 46]. At about the same time, a different proposal *spawn* allowed new delimiters to be generated dynamically [29, 30]. In this system, the base of each delimited continuation is rooted at a different #. The action of creating the # returns a specialised control operator for accessing occurrences of this particular #; this specialised control operator can then be used for capturing (and aborting) the particular delimited continuation rooted at the newly generated # (and only that one). This is more expressive and convenient than either single delimiters or hierarchies of delimiters and allows arbitrary nesting and composition of continuation-based abstractions. A later proposal by Gunter et al. [24] separated the operation for *creating* new delimiters from the control operator *using* that #.

8.2 Control delimiters and extent

The issues related to hierarchies of control delimiters or the dynamic generation of new names for control delimiters, appear orthogonal to our analysis. Indeed, the presence of multiple delimiters does not change the fundamental point about the dynamic behaviour of each individual #.

However, our analysis fundamentally relies on a subtle issue related to the *extent* of delimiters [39]. More precisely, there is no question that the # delimits the part of the context that a control operator gets to capture, but given that constraint there are still *four* choices to consider with very different semantics [12]:

$$E[\#\ (E_{\uparrow}[\mathcal{F}_1\ M])] \mapsto E[M\ (\lambda x.E_{\uparrow}[x])]$$
$$E[\#\ (E_{\uparrow}[\mathcal{F}_2\ M])] \mapsto E[\#\ (M\ (\lambda x.E_{\uparrow}[x]))]$$
$$E[\#\ (E_{\uparrow}[\mathcal{F}_3\ M])] \mapsto E[M\ (\lambda x.\#\ (E_{\uparrow}[x]))]$$
$$E[\#\ (E_{\uparrow}[\mathcal{F}_4\ M])] \mapsto E[\#\ (M\ (\lambda x.\#\ (E_{\uparrow}[x])))]$$

All four variants have been proposed in the literature: $\mathcal{F}_1$ is like the operator *cupto* [24]; $\mathcal{F}_2$ is Felleisen's $\mathcal{F}$ operator [13]; $\mathcal{F}_3$ is like a *spawn* controller [29]; and $\mathcal{F}_4$ is *shift* [10].

It turns out that the inclusion of the # in the reified continuation (variants $\mathcal{F}_3$ and $\mathcal{F}_4$) simplifies the semantics considerably. For example, when a continuation captured by $\mathcal{F}_3$ or $\mathcal{F}_4$ is invoked, the included # can be used to provide the required top-level context for that invocation. In the case of $\mathcal{F}_1$ or $\mathcal{F}_2$, there is no included #; so when the continuation is invoked, we must "search" for the # required to denote the top-level. In general, the # can be located arbitrarily deep in the calling context. Since the representation of contexts as delimited continuations does not support operations for "searching for delimiters," Felleisen et al. [17] developed a special model based on an *algebra of contexts* which supports the required operations. In a recent investigation of control operators for delimited continuations, Dybvig et al. [12] show however that it is possible to use standard continuation semantics to explain all the four variants of operators above: the trick is to augment the model with a state variable containing a *sequence of continuations and delimiters*. Chung-Chieh Shan [45] has also recently shown a similar result using a different encoding for the operators in terms of *shift* and *reset*. We believe that these encodings can be adapted as the basis for an embedding of the operators in a variant (or extension) of $\lambda_{\mathcal{C}}^{-}$.

8.3 The Curry-Howard correspondence

The Curry-Howard correspondence relates proofs to programs, propositions to types, and proof normalisation to program normalisation. Typically, the simply-typed $\lambda_{\mathcal{C}\mathsf{tp}}$-calculus (alternatively Parigot's $\lambda\mu_{\mathsf{tp}}$-calculus) coincides with classical natural deduction as soon as tp is assigned the type $\bot$ [1]. Of course, this is a restriction from the point of view of any computationally interesting type system for control for which one would expect the top-level to be of an inhabited type (e.g. the type of integers). But this is really where the Curry-Howard correspondence holds, on the fragment of Fig. 6 obtained by setting T to $\bot$.

We mapped the effect-based type systems within subtractive logic but we did not answer the question of a *direct* syntactic Curry-Howard-style interpretation.

A possible approach is the following: in the same way as the logical interpretation of tp expects it has type $\bot$, let's constrain $\widehat{tp}$ to take arguments of type $\bot$. By this way, the type system $\Lambda_{\mathcal{C}\widehat{tp}}^{\overrightarrow{fixed}}$ collapses to classical implicational logic and similarly for the type systems $\Lambda_{\mathcal{C}\widehat{tp}}^{\overrightarrow{eff_{eq}}}$ and $\Lambda_{\mathcal{C}\widehat{tp}}^{\overrightarrow{eff}}$ when, in addition, the effects annotations, implicitly set to $\bot$, are omitted. Along this interpretation, delimiters provide no extra logical expressiveness. This is somehow disappointing but can we really get more? After all, even if it does not enrich the logic, it still enriches the proof language. And this may well be related to completeness questions, since $\lambda\mu$-calculus is known not to satisfy Böhm theorem [11] while adding delimiters is known to provide some completeness properties [19, 47].

Acknowledgements We thank Olivier Danvy, Matthias Felleisen, Yukiyoshi Kameyama, and Hayo Thielecke for the discussions and help they provided in understanding their results. We would also like to thank the ICFP and HOSC reviewers who provided corrections and extensive comments on the presentation of both the conference and journal versions.

Appendix Proof of strong normalisation of $\lambda_{\mathcal{C}}^{-}$

The $\lambda_{\mathcal{C}}^{-}$ calculus is defined by the reduction rules in Fig. 18. We show that its simply-typed fragment, as described in Fig. 19, is *strongly normalisable* (SN).

We begin by establishing a closure property of SN that is needed in the rest of the proof. We then generalise the notion of SN to that of *reducibility* and show that all typed terms are reducible.

In the following we say that M (respectively E or $J_c[E]$) is SN if all its immediate reducts are. This is an inductive definition so that we can reason by structural induction on it.

Properties of SN

As the following establishes, strong normalisability is preserved by head expansion.

Lemma 38 (1) *If V and $J_c[E[M[V/x]]]$ are SN, then $J_c[E[(\lambda x.\, M)V]]$ is SN.*

(2) *If $J_c[E]$, V and $J[V/y][J_c[E]/k]$ are SN then we also have that*: **let** $(y,k) = (V, \lambda x.\, J_c[E[x]])$ **in** J *is SN.*

(3) *If $J_c[E]$ and $J[J_c[E]/k]$ are SN then $J_c[E[\mathcal{C}(\lambda k.\, J)]]$ is SN.*

Proof (1) We reason by induction on the SN proofs. If a reduction step occurs in $J_c[E]$ leading to $J'_c[E']$ then $J_c[E[M[V/x]]] \to J'_c[E'[M[V/x]]]$ and the induction hypothesis on $J'_c[E'[M[V/x]]]$ SN applies. Similarly for a reduction step in M. If a reduction step occurs in V leading to V' then $J_c[E[M[V/x]]] \twoheadrightarrow J_c[E[M[V'/x]]]$. Hence, the jump $J_c[E[M[V'/x]]]$ is SN and the induction hypothesis on V' SN applies. Finally, if $J_c[E[(\lambda x.M)V]] \to J_c[E[M[V/x]]]$, the result is SN by hypothesis.

(2) Let $J' = J[V/x][J_c[E]/k]$. We reason by induction on the structure of E, then by induction on the proofs of SN for $J_c[E]$, V, and J'. If $E = \Box$, then a reduction step in **let** $(x,k) = (V, \lambda x.\, J_c[x])$ **in** J occurs either in J_c, J, V or it yields J'. In the first cases, we apply the induction hypothesis as above. In the latter case, the contractum is directly SN by hypothesis. Otherwise, if E is some $E'[F]$, we consider the different reduction steps that can occur in **let** $(x,k) = (V, \lambda x.\, J_c[E'[F[x]]])$ **in** J:

- If the reduction step occurs in V (respectively $J_c[E'[F]]$) leading to V' (respectively $J''_c[E'']$), then $J' \twoheadrightarrow J[V'/x][J_c[E]/k]]$ (respectively $J' \twoheadrightarrow J[V/x][J''_c[E'']/k]]$) so that the reducts of J' and V (respectively $J_c[E'[F]]$) are SN and the induction hypothesis applies.
- If the reduction step occurs in J leading to some jump J'', then $J' \to J''[V/x][J_c[E]/k]]$ so that the latter reduct is SN and the induction hypothesis applies.
- If the reduction gives:
  let $(x,k) = (V, \lambda x.\, J_c[E'[x]])$ **in** $J[kF/k]$ then J' can be rewritten to $J[kF/k][V/x][J_c[E']/k]$ so that the induction hypothesis on E applies (taking E' and $J[kF/k]$ for E and J respectively).

(3) As above, we reason by induction on the structure of $J_c[E]$, then by induction on the proofs of SN for $J_c[E]$, and $J[J_c[E]/k]$. If $E = \Box$, then there are two interesting cases:

- If $J_c[\mathcal{C}(\lambda k.\, J)] \to J[J_c/k]$ (i.e. $\mathcal{C}_{idem}$ or Sub_v^{lift}), then SN follows from the hypothesis.
- If J is some kM with k not free in M and $J_c[\mathcal{C}k.J] \to J_c[M]$ using $\mathcal{C}_{elim}$, then the reasoning is the same since $J_c[M] = J[J_c/k]$.

If E is some $E'[F]$, then there are also two interesting cases:

- If $J_c[E'[F[\mathcal{C}(\lambda k.\, J)]]] \to J_c[E'[\mathcal{C}(\lambda k.J[kF/k])]]$, then SN follows by induction hypothesis on E' since $J[J_c[E'[F]]/k]$ can be rewritten into $J[kF/k][J_c[E']/k]$.

– If J is some kM with k not free in M and $J_c[E[\mathcal{C}k.J]] \to J_c[E[M]]$ using $\mathcal{C}_{elim}$, then again, the reasoning is the same since $J_c[E[M]] = J[J_c[E]/k]$.

□

Reducibility

We define the property "*reducible of a given type*" for terms and jump contexts. The definition is by induction on the type, then by mutual induction on values, jump contexts, and non-value terms, with priority given first to values, then to jump contexts, and finally to non-values. The definition is a straightforward adaptation of the notion of reducibility developed by Herbelin [28] for proving strong normalisation of the call-by-name $\overline{\lambda}\mu\tilde{\mu}$-calculus.

Definition 39 (Reducible of a given type)

– A value V is reducible of type A iff:
  • V is a variable.
  • V is $\lambda x.N$ and A is some $B \to C$ such that for all value V' reducible of type B, $V\ V'$ is reducible of type C.
  • V is $(V', \lambda x.\, J_c[E[x]])$ and A is some $B - C$ and V' is reducible of type B and $J_c[E]$ is reducible of type C.
– A jump context $J_c[E]$ is reducible of type A iff for all value V reducible of type A, $J_c[E[V]]$ is SN.
– A non-value term M is reducible of type A iff for all jump contexts $J_c[E]$ reducible of type A, $J_c[E[M]]$ is SN.

Remark 1 Since reducible jump contexts are SN against any reducible value, the *if* part of the last clause of the definition of reducibility also holds for values.

Properties of reducibility

Lemma 40 *Reducible values, non-value terms and contexts are SN*:

(1) *For all $J_c[E]$ reducible of type A, $J_c[E]$ is SN.*
(2) *For all M reducible of type A, M is SN.*
(3) *For all k of type A, $k\,\Box$ is a reducible jump context.*

Proof Let $J_c[E]$ be a reducible jump context. Take a variable x. It is a reducible value, hence $J_c[E[x]]$ is SN. Especially $J_c[E]$ is SN.

We show the last two items conjointly by induction on A. We first show that M reducible is SN if it is a value.

– If M is a variable, it is SN.
– If M is $\lambda x.N$ of type $B \to C$, then applying it to x (which is reducible of type B) yields a reducible term of type C identical to N. By induction hypothesis on C, N is SN so that M is SN too.
– If M is (V, kE) of type $B - C$ with kE reducible of type C and V reducible of type B then kE is SN, and V is SN by induction hypothesis, so that M is SN.

We then show that all jump contexts of the form $k\,\Box$ are reducible of type A. Let V be a reducible value of type A. We have to show that $k\,V$ is SN. We just showed that V was SN. Since the only redexes of $k\,V$ are redexes of V, we get $k\,V$ SN.

We can now extend the result to any non-value M. Let k be a continuation variable of type A. Since $k\,\Box$ is reducible of type A, we obtain that $k\,M$ is SN. Especially, M is SN. □

The following lemma shows that reducibility of jump contexts is preserved by context construction.

Lemma 41 *We have*:

(1) *If V is reducible of type $A \to B$ and $J_c[E]$ reducible of type B then $J_c[E[V\,\Box]]$ is reducible of type A.*
(2) *If M is reducible of type A and $J_c[E]$ reducible of type B then $J_c[E[\Box\,M]]$ is reducible of type $A \to B$.*
(3) *If $J_c[E]'$ is reducible of type A and $J_c[E]$ reducible of type $B - A$ then $J_c[E[(\Box, \lambda x.\, J'_c[E'[x]])]]$ is reducible of type B.*
(4) *If $J[V/x][J_c[E]/k']$ is SN for any reducible V of type B and $J_c[E]$ of type C then* **let** $(x, k') = \Box$ **in** J *is a reducible jump context of type $B - C$.*

Proof (1) We have to show that $J_c[E[V\,V']]$ is SN for any V' reducible of type A. If V is some variable x then redexes of $J_c[E[x\,V']]$ are either in $J_c[E]$ or in V' which are SN by Lemma 40. If V is some $\lambda x.M$ then, by definition of its reducibility, $V\,V'$ is reducible. Combined with the reducibility of $J_c[E]$, we get $J_c[E[V\,V']]$ SN.

(2) We have to show that $J_c[E[V\,M]]$ is SN for any V reducible of type $A \to B$. By the previous item, $J_c[E[V\,\Box]]$ is a reducible context of type A. Hence, $J_c[E[V\,M]]$ is SN, by reducibility of M.

(3) We have to show that $J_c[E[(V, \lambda x.\, J'_c[E'[x]])]]$ is SN for any V reducible of type B. The reducibility of $J'_c[E']$ and V implies the reducibility of the value $(V, \lambda x.\, J'_c[E'[x]])$. Hence, the reducibility of $J_c[E]$ implies $J_c[E[(V, \lambda x.\, J'_c[E'[x]])]]$ SN.

(4) We have to show that **let** $(x, k') = V$ **in** J is SN for any V reducible of type $B - C$. If V is some variable y then the only redexes of **let** $(x, k') = y$ **in** J are in J. Since $J = J[x/x][k'\Box/k']$ which is SN by hypothesis, the whole expression is SN too. If V is some value of the form $(V', \lambda x.\, J''_c[E''[x]])$, then by hypothesis, $J[V'/x][J''_c[E'']/k']$ is SN. By Lemma 38(2), we conclude that **let** $(x, k') = (V', J''_c[E''])$ **in** J is SN too. □

Adequacy lemma

Finally, we show the main lemma that all typed terms are reducible.

Lemma 42 *Let Γ be an ordered context of declarations of the form either $x_i : A_i$ or $k_i : B_i \to \perp\!\!\!\perp$. Let V_i be instances for the variables x_i and $J^i_c[E_i]$ be instances for the variables k_i such that the free variables of V_i and $J^i_c[E_i]$ are among the x_j and k_j for $j \leq i$. The V_i are reducible values of respective types A_i and the $J^i_c[E_i]$ are reducible jump contexts of respective types B_i. We write $[\sigma]$ for the ordered substitution mixing the substitution $[V_i/x_i]$ and $[J^i_c[E_i]/k_i]$. We have*:

– $\Gamma, \Delta \vdash M : A$ *implies* $M[\sigma]$ *reducible of type* A,
– $\Gamma, \Delta \vdash J :\perp\!\!\!\perp$ *implies* $J[\sigma]$ *SN.*

Proof We reason by induction on the derivation of $\Gamma, \Delta \vdash M : A$ or $\Gamma, \Delta \vdash J :\perp\!\!\!\perp$.

– Rule Ax with $M = x_i$: this is direct by reducibility of V_i.

- Rule $\to_e$ with $M = M_1 M_2$ with M_1 of type $B \to A$ and M_2 of type B. Let $J_c[E]$ a reducible jump context of type A. Since $M_2' = M_2[\sigma]$ is reducible of type B by induction hypothesis, $J_c[E[\Box\ M_2']]$ is reducible of type $B \to A$ by Lemma 41(2). Since $M_1' = M_1[\sigma]$ is reducible by induction hypothesis, we have that $J_c[E[M_1' M_2']]$ is SN. Hence $M[\sigma] = M_1' M_2'$ is reducible.
- Rule $\to_i$ with $M = \lambda x.N : A \to B$. Let V be a reducible value of type A and $J_c[E]$ a reducible jump context of type B. By induction hypothesis, we have $N[\sigma][V/x]$ reducible of type B hence $J_c[E[N[\sigma][V/x]]]$ is SN. By Lemma 40, V is SN. By Lemma 38(1), we get $J_c[E[(\lambda x.(N[\sigma]))V]]$ SN so that $(\lambda x.N)[\sigma] = \lambda x.(N[\sigma])$ is reducible.
- Rule $\to_e^k$ with $J = k_{i_0} M$ and M of type A and k_{i_0} of type $A \to \perp\!\!\!\perp$. By induction hypothesis we have $M' = M[\sigma]$ reducible of type A. Since $J_c^{i_0}[E_{i_0}]$ is reducible (of type A), we get $J[\sigma] = J_c^{i_0}[E_{i_0}][M']$ SN.
- Rule *RAA* with $M = \mathcal{C}(\lambda k.J)$. Let $J_c[E]$ a reducible jump context of type A. By induction hypothesis we have $J' = J[\sigma][J_c[E]/k]$ SN. By Lemma 40, $J_c[E]$ is SN so that we get $J_c[E[\mathcal{C}(\lambda k.J)]]$ SN by Lemma 38(3).
- Rule $-_e$ with $J_0 = (\mathbf{let}\ (y,k) = N\ \mathbf{in}\ J)$ with N of type $B - C$. We have to show that $J_0[\sigma]$ is SN. By induction hypothesis we already know that $N' = N[\sigma]$ is reducible of type $B - C$. Let $J' = J[\sigma]$. By induction hypothesis, we have that $J'[V/y][J_c[E]/k]$ is SN for every reducible V of type B and $J_c[E]$ of type C. Hence, by Lemma 41(4), $\mathbf{let}\ (y,k) = \Box\ \mathbf{in}\ J'$ is a reducible jump context of type $B - C$. By reducibility of N', we conclude that $J_0[\sigma] = \mathbf{let}\ (y,k) = N'\ \mathbf{in}\ J'$ is SN.
- Rule $-_i$ with $M = (M, \lambda x.\, J_c[E[x]])$ and $A = B - C$. The typability of $J_c[E]$ states that $\Gamma, \Delta, x : C \vdash J_c[E[x]] : \perp\!\!\!\perp$ for some fresh variable x. By induction hypothesis, $((J_c[E])[\sigma])[V] = J_c[E[x]][\sigma][V/x]$ is SN for any reducible V of type C so that $J_c'[E'] = (J_c[E])[\sigma]$ is reducible of type C. By induction hypothesis, $N' = N[\sigma]$ is reducible too, of type B. If N' is a value, we get that $M[\sigma] = (N', \lambda x.\, J_c'[E'[x]])$ is reducible. Otherwise, we have to show that $J_c''[E''[(N', \lambda x.\, J_c'[E'[x]])]]$ is SN for every reducible jump context $J_c''[E'']$ of type $B - C$. By Lemma 41(3), $J_c''[E''[(\Box, \lambda x.\, J_c'[E'[x]])]]$ is reducible of type B. Hence, by reducibility of N', we conclude that $J_c''[E''[(N', \lambda x.\, J_c'[E'[x]])]]$ is SN.

□

Combining the above with Lemma 40(1 and 2), and because variables and variable-based jump contexts are reducible (respectively by definition and by Lemma 40(3)), we finally get the strong normalisability of $\lambda_{\mathcal{C}}^{-}$.

Theorem 43 *Typed $\lambda_{\mathcal{C}}^{-}$ is strongly normalisable.*

References

1. Ariola, Z.M., Herbelin, H.: Minimal classical logic and control operators. In: Thirtieth International Colloquium on Automata, Languages and Programming, ICALP'03, Eindhoven, The Netherlands, June 30–July 4, 2003. Lecture Notes in Comput. Sci., vol. 2719, pp. 871–885. Springer, New York (2003)
2. Ariola, Z.M., Herbelin, H.: Control reduction theories: the benefit of structural substitution. J. Funct. Program. (2007, to appear)
3. Ariola, Z.M., Herbelin, H., Sabry, A.: A type-theoretic foundation of continuations and prompts. In: ACM SIGPLAN International Conference on Functional Programming, pp. 40–53. ACM Press, New York (2004)
4. Baba, K., Hirokawa, S., Fujita, K.: Parallel reduction in type free $\lambda\mu$-calculus. Electron. Notes Theor. Comput. Sci. **42**, 52–66 (2001)

5. Barbanera, F., Berardi, S.: Extracting constructive content from classical logic via control-like reductions. In: Bezem, M., Groote, J.F. (eds.) Proceedings 1st Intl. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993. Lecture Notes in Comput. Sci., vol. 664, pp. 45–59. Springer, Berlin (1993)
6. Crolard, T.: Subtractive logic. Theor. Comput. Sci. **254**(1–2), 151–185 (2001)
7. Crolard, T.: A formulae-as-types interpretation of subtractive logic. J. Log. Comput. **14**(4), 529–570 (2004) (Special issue on Modalities in Constructive Logics and Type Theories)
8. Curien, P.-L., Herbelin, H.: The duality of computation. In: ACM SIGPLAN International Conference on Functional Programming, pp. 233–243. ACM Press, New York (2000)
9. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark (1989)
10. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, pp. 151–160. ACM Press, New York (1990)
11. David, R., Py, W.: Lambda-mu-calculus and Böhm's theorem. J. Symb. Log. **66**(1), 407–413 (2001)
12. Dybvig, R.K., Peyton-Jones, S., Sabry, A.: A monadic framework for subcontinuations. J. Funct. Program. (2007, to appear). http://journals.cambridge.org/action/displayIssue?iid=168229
13. Felleisen, M.: The theory and practice of first-class prompts. In: Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL '88), pp. 180–190. ACM Press, New York (1988)
14. Felleisen, M.: On the expressive power of programming languages. In: Jones, N. (ed.) ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark. Lecture Notes in Comput. Sci., vol. 432, pp. 134–151. Springer, New York (1990)
15. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992)
16. Felleisen, M., Friedman, D., Kohlbecker, E.: A syntactic theory of sequential control. Theor. Comput. Sci. **52**(3), 205–237 (1987)
17. Felleisen, M., Wand, M., Friedman, D.P., Duba, B.F.: Abstract continuations: a mathematical semantics for handling full functional jumps. In: Conference on LISP and Functional Programming, Snowbird, Utah, pp. 52–62. ACM Press, New York (1988)
18. Filinski, A.: Declarative continuations: an investigation of duality in programming language semantics. In: Category Theory and Computer Science, Manchester, UK, September 5–8, 1989, Proceedings. Lecture Notes in Comput. Sci., vol. 389, pp. 224–249. Springer, New York (1989)
19. Filinski, A.: Representing monads. In: Conf. Record 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994, pp. 446–457. ACM Press, New York (1994)
20. Filinski, A.: Representing layered monads. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 175–188. ACM Press, New York (1999)
21. Fischer, M.J.: Lambda-calculus schemata. In: Proc. ACM Conference on Proving Assertions About Programs. SIGPLAN Notices, vol. 7(1), pp. 104–109. ACM Press, New York (1972)
22. Fischer, M.J.: Lambda-calculus schemata. Lisp Symb. Comput. **6**(3/4), 259–288 (1993). http://www.brics.dk/~hosc/vol06/03-fischer.html. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, vol. 7, no. 1, January 1972
23. Griffin, T.G.: The formulae-as-types notion of control. In: Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990, pp. 47–57. ACM Press, New York (1990)
24. Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Functional Programming & Computer Architecture. ACM Press, New York (1995)
25. Guzmán, J., Suárez, A.: An extended type system for exceptions. In: Record of the fifth ACM SIGPLAN workshop on ML and its Applications. Also appeared as Research Report 2265, INRIA, BP 105-78153, Le Chesnay Cedex, France (1994)
26. Haynes, C.T.: Logic continuations. In: Proceedings of the Third International Conference on Logic Programming. Lecture Notes in Comput. Sci., vol. 225, pp. 671–685. Springer, Berlin (1986)
27. Haynes, C.T., Friedman, D., Wand, M.: Obtaining coroutines from continuations. J. Comput. Lang. **11**, 143–153 (1986)
28. Herbelin, H.: Explicit substitutions and reducibility. J. Log. Comput. **11**(3), 431–451 (2001)
29. Hieb, R., Dybvig, R.K.: Continuations and concurrency. In: PPoPP '90, Symposium on Principles and Practice of Parallel Programming, Seattle, Washington, March 14–16. SIGPLAN Notices, vol. 25(3), pp. 128–136. ACM Press, New York (1990)
30. Hieb, R., Dybvig, K., Anderson, C.W. III: Subcontinuations. Lisp Symb. Comput. **7**(1), 83–110 (1994)

31. Hofmann, M.: Sound and complete axiomatisations of call-by-balue control operators. Math. Struct. Comput. Sci. **5**(4), 461–482 (1995)
32. Howard, W.: The formulae-as-types notion of construction. In: Hindley, J.R., Seldin, J.P. (eds.) To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic, New York (1980)
33. Kameyama, Y.: A type-theoretic study on partial continuations. In: IFIP TCS, pp. 489–504 (2000)
34. Kameyama, Y.: Towards logical understanding of delimited continuations. In: Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01) (2001)
35. Kameyama, Y., Hasegawa, M.: A sound and complete axiomatization of delimited continuations. In: Proc. of 8th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'03, Uppsala, Sweden, 25–29 Aug. 2003. SIGPLAN Notices, vol. 38(9), pp. 177–188. ACM Press, New York (2003)
36. Lillibridge, M.: Unchecked exceptions can be strictly more powerful than Call/CC. Higher-Order Symb. Comput. **12**(1), 75–104 (1999)
37. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science, pp. 14–23. IEEE Press (1989)
38. Moreau, L.: A syntactic theory of dynamic binding. Higher-Order Symb. Comput. **11**(3), 233–279 (1998)
39. Moreau, L., Queinnec, C.: Partial continuations as the difference of continuations. A duumvirate of control operators. In: International Conference on Programming Language Implementation and Logic Programming (PLILP'94), Madrid, Spain, pp. 182–197. Springer, Berlin (1994)
40. Murthy, C.: Control operators, hierarchies, and pseudo-classical type systems: a—translation at work. In: ACM workshop on Continuations, pp. 49–71 (1992)
41. Ong, C.-H.L., Stewart, C.A.: A Curry-Howard foundation for functional computation with control. In: Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan. 1997, pp. 215–227. ACM Press, New York (1997)
42. Parigot, M.: Lambda-mu-calculus: an algorithmic interpretation of classical natural deduction. In: Logic Programming and Automated Reasoning: International Conference LPAR '92 Proceedings, St. Petersburg, Russia, pp. 190–201. Springer, Berlin (1992)
43. Rauszer, C.: Semi-boolean algebras and their application to intuitionistic logic with dual connectives. Fundam. Math. **83**, 219–249 (1974)
44. Riecke, J.G., Thielecke, H.: Typed exceptions and continuations cannot macro-express each other. In: Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Comput. Sci., vol. 1644, pp. 635–644. Springer, Berlin (1999)
45. Shan, C.: Shift to control. In: Shivers, O., Waddell, O. (eds.) Proceedings of the 5th Workshop on Scheme and Functional Programming, pp. 99–107. Technical report, Computer Science Department, Indiana University (2004)
46. Sitaram, D., Felleisen, M.: Control delimiters and their hierarchies. Lisp Symb. Comput. **3**(1), 67–99 (1990a)
47. Sitaram, D., Felleisen, M.: Reasoning with continuations II: full abstraction for models of control. In: LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 161–175. ACM Press, New York (1990b)
48. Thielecke, H.: On exceptions versus continuations in the presence of state. In: Proceedings of the Ninth European Symposium On Programming (ESOP). Lecture Notes in Comput. Sci., vol. 1782, pp. 397–411. Springer, Berlin (2000)
49. Thielecke, H.: Contrasting exceptions and continuations. Available from http://www.cs.bham.ac.uk/~hxt/research/exncontjournal.pdf (2001)
50. Thielecke, H.: Comparing control constructs by double-barrelled CPS. Higher-Order Symb. Comput. **15**(2/3), 119–136 (2002)
51. Wadler, P.: Monads and composable continuations. Lisp Symb. Comput. **7**(1), 39–56 (1994)
52. Wand, M.: Continuation-based multiprocessing. Higher-Order Symb. Comput. **12**(3), 285–299 (1999), http://www.brics.dk/~hosc/vol12/3-wand.html. Reprinted from the Proceedings of the 1980 Lisp Conference, with a foreword