A Type-Theoretic Foundation of Continuations and Prompts

Zena M. Ariola*
University of Oregon
ariola@cs.uoregon.edu

Hugo Herbelin
INRIA-Futurs
Hugo.Herbelin@inria.fr

Amr Sabry*
Indiana University
sabry@indiana.edu

Abstract

There is a correspondence between classical logic and programming language calculi with first-class continuations. With the addition of control delimiters (prompts), the continuations become composable and the calculi are believed to become more expressive. We formalise that the addition of prompts 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. From a logic perspective, the effect annotations can be understood in a standard logic extended with the dual of implication, namely subtraction.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational Semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms: Languages, Theory

Keywords: callcc, continuation, monad, prompt, reset, shift, subcontinuation, subtraction

1 Introduction

Programming practice suggests that control operators add expressive power to purely functional languages. For example, control operators permit the implementation of backtracking [18], coroutines [19], and lightweight processes [41], which go beyond pure functional programming. Of course, any *complete program* that uses these abstractions can be *globally transformed* to a purely functional program, but this misses the point. As Felleisen [9] 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 way to formalise the additional expressive power of control operators that is based on the Curry-Howard isomorphism [22]. The pure λ-calculus corresponds to intuitionistic logic; extending it with the control operator $\mathcal{C}$ [8] makes it equivalent to classical logic [15], which is evidently more expressive. Felleisen also showed that the control operator *callcc* is less expressive than $\mathcal{C}$ [9]. Ariola and Herbelin provided the logical explanation [1]: *callcc* corresponds to *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.

This logic-based expressiveness is arguably a simpler approach but it has so far only been applied to a small family of control operators, and the situation for the other control operators is much less understood. For example, the operators *shift* and *reset* [6] are widely believed to be more expressive than $\mathcal{C}$ but it is not clear how to formalise this belief for at least three reasons:

1. Several incompatible type systems and type-and-effect systems have been proposed for these operators [23, 30, 24, 5], and it is not apparent which of these systems is the "right" one even when moving to the world of continuation-passing style [40]. 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. Under the Curry-Howard isomorphism, reductions rules correspond to proof normalisation steps and hence we expect the reductions rules for the control operators to be confluent and, in the simply-typed case, strongly normalising. But although the semantics of many control operators (including *shift* and *reset* [25]) can be given using local reduction rules, many of the systems are not confluent, and under none of the type-and-effect systems are the typed terms known to be strongly normalising.
3. The operators *shift* and *reset* can simulate a large number of other computational effects like state and exceptions [12]. 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. Thielecke *et. al.* [38, 37, 33] have formalised the expressive power of various combinations of continuations, exceptions and state but their results need to be generalised and explained using standard type systems and logics.

*Supported by NSF grant number CCR-0204389.

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

Based on the technical results in this paper, we propose a calculus $\lambda_{\mathcal{C}}^{\rightleftarrows -}$ which corresponds to classical *subtractive logic* [32, 2], as a foundation in which to reason about "all" control operators and their relative expressive power. Restrictions of classical subtractive logic have been independently (and recently) shown to correspond to coroutines and exceptions [3] but our connection to the more expressive world of first-class continuations and prompts is novel. In more detail, we establish that the additional expressiveness of *shift* and *reset* comes from the fact that *reset* essentially corresponds to a dynamically-scoped variable, and in the presence of dynamic scope, continuations can model other effects like state and exceptions. Logically this is apparent from the following equivalences that hold *classically*: $A \wedge \neg T \to B = A \to B \vee T = A \wedge \neg T \to B \wedge \neg T$. In other words, in the presence of control operators, a function that takes a dynamically-scoped environment entry of type $\neg T$ is the same as a function that throws an exception of type T, which is the same as a function that manipulates a state variable of type $\neg T$. Each of these views leads to a different type-and-effect system for *shift* and *reset*, which can all be embedded in the type $A - T \to B - U$ that uses the subtraction connective. We also note that *shift* and *reset* only use a limited amount of the expressive power of classical subtractive logic. This leaves room for investigating other control operators that are believed to be even more expressive.

The next three sections review some background material and set up the stage for our technical development. Section 2 reviews some of the basic control operators and their connection to classical logic discovered by Griffin. Section 3 improves on Griffin's original formulation by using the $\lambda_{\mathcal{C}_{\mathsf{tp}}}^{\rightleftarrows}$-calculus from our previous work. Section 4 reviews the semantics of *shift* and *reset* and their equivalent formulation using $\mathcal{C}$ and *prompt*. It explains that an essential ingredient of the semantics is the dynamic nature of *prompt*.

Our contributions are explained in detail in the remaining sections. The main point of Section 5 is that it is possible to explain the semantics of *shift* and *reset* in the context of $\lambda_{\mathcal{C}_{\mathsf{tp}}}^{\rightleftarrows}$ by generalising the calculus to include just one dynamically-scoped variable. The section formalises this point by giving the semantics and establishing its soundness and completeness with respect to the original semantics. Section 6 investigates various ways to extend the type system of $\lambda_{\mathcal{C}_{\mathsf{tp}}}^{\rightleftarrows}$ to accommodate the dynamically-scoped variable. We present three systems (which are closely related to existing type-and-effect systems) and reason about their properties. Sections 7 and 8 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 $\lambda_{\mathcal{C}}^{\rightleftarrows -}$. The second focuses on using the subtractive type for explaining and managing the effect annotations. Section 9 concludes with a discussion of other control operators.

2 Control and Classical Logic

We review the semantics of continuation-based control operators and their connection to classical logic.

2.1 Operational Semantics

Figure 1 introduces a call-by-value calculus extended with the operators *abort* ($\mathcal{A}$), *callcc* ($\mathcal{K}$), and $\mathcal{C}$. The semantics of the control operators can be described most concisely using the following three operational rules, which rewrite complete programs:

$$\begin{array}{rcl} E[\mathcal{A}\,M] & \mapsto & M \\ E[\mathcal{K}\,M] & \mapsto & E[M\ (\lambda x.\mathcal{A}\,E[x])] \\ E[\mathcal{C}\,M] & \mapsto & M\ (\lambda x.\mathcal{A}\,E[x]) \end{array}$$

$$\begin{array}{lcl} x,a,v,f,c \in \mathit{Vars} & & \\ M,N \in \mathit{Terms} & ::= & x \mid \lambda x.M \mid MN \mid \mathcal{A}\,M \mid \mathcal{K}\,M \mid \mathcal{C}\,M \\ V \in \mathit{Values} & ::= & x \mid \lambda x.M \\ E \in \mathit{EvCtxt} & ::= & \square \mid E\,M \mid V\,E \end{array}$$

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

In each of the rules, the entire program is split into an evaluation context E representing the continuation, and a current redex to rewrite. The operator $\mathcal{A}$ aborts the continuation returning its subexpression to the top-level; the other two operators capture the evaluation context E and reify it as a function $(\lambda x.\mathcal{A}\,E[x])$. When invoked, this function aborts the evaluation context at the point of invocation, and installs the captured context instead.

The rules show that $\mathcal{C}$ differs from $\mathcal{K}$ in that it does not duplicate the evaluation context. This difference makes $\mathcal{C}$ at least as expressive as both $\mathcal{A}$ and $\mathcal{K}$; it can be used to define them as follows:

$$\begin{array}{rcll} \mathcal{A}\,M & \triangleq & \mathcal{C}\ (\lambda\_.\ M) & \text{(Abbrev. 1)} \\ \mathcal{K}\,M & \triangleq & \mathcal{C}\ (\lambda c.\ c\ (M\ c)) & \text{(Abbrev. 2)} \end{array}$$

We use _ to refer to an anonymous variable.

We will therefore focus on $\mathcal{C}$ in the remainder of the paper, but still occasionally treat $\mathcal{A}$ as a primitive control operator to provide more intuition.

EXAMPLE 1. (A $\lambda_{\mathcal{C}}$-term and its evaluation) We have:

$$\mathcal{C}\ (\lambda c.\ 1\ +\ c\ 2) \mapsto (\lambda c.\ 1\ +\ c\ 2)\ (\lambda x.\mathcal{A}\,x) \mapsto\!\!\!\!\to 1\ +\ \mathcal{A}\,2 \mapsto 2$$

The invocation of the continuation c abandons the context $1\ +\ [\,]$.

2.2 Reduction Rules

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

Figure 2. Reductions of call-by-value $\lambda_{\mathcal{C}}$

Instead of presenting the semantics of $\lambda_{\mathcal{C}}$ as a relation on complete programs, it is possible to give local reduction rules that are applicable anywhere in a term and in arbitrary order. (See Figure 2.) Instead of capturing the entire evaluation context at once, the rules allow one to *lift* the control operation step-by-step until it reaches another control operator. At any point, it is possible to use $\mathcal{C}_{top}$ to start applying M to part of the captured context and then continue lifting the outer $\mathcal{C}$ to accumulate more of the context. The rules are in correspondence with the operational semantics in the sense that there is a standard reduction sequence that reaches an answer which is almost identical to the answer produced by the operational semantics [10, Th. 3.17]. For the term in Example 1, the standard reduction sequence produces $\mathcal{C}\ (\lambda c.c\ 2)$ instead of 2.

2.3 Griffin's Type System

Griffin introduced a type system for $\lambda_{\mathcal{C}}$ where types can also be read as propositions [15]. The set of basic types is assumed to include a special type $\perp$ which represents an empty type or the proposition "false." The type system is given in Figure 3. In the rule

$$\begin{array}{rcl} b \in \mathit{BaseType} & = & \{\ \bot, \dots\ \} \\ A,B \in \mathit{TypeExp} & ::= & b \mid A \to B \end{array}$$

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

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash M' : A}{\Gamma \vdash MM' : B}\ \to_e$$

$$\frac{\Gamma \vdash M : (A \to \bot) \to \bot}{\Gamma \vdash \mathcal{C}M : A}\ \neg\neg_E$$

Figure 3. Type system of $\lambda_{\mathcal{C}}$ (Griffin)

$\neg\neg_E$, a continuation accepting a value of type A is given the type $A \to\ \bot$ (which corresponds to the negation $\neg A$). Thus, the closed term $\lambda x.\mathcal{C}x$ provides a proof of the double negation elimination rule $\neg\neg A \to A$.

PROPOSITION 1 (GRIFFIN). *A formula A is provable in classical logic iff there exists a closed $\lambda_{\mathcal{C}}$ term M such that $\vdash M : A$.*

3 The $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus

Our previous work [1] introduced a refinement of the $\lambda_{\mathcal{C}}$-calculus called the $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus, which is better behaved both as a reduction system and as a logical system. We introduce this calculus and use it in the remainder of the paper as the basis for developing a uniform framework for reasoning about control operators.

3.1 The Top-Level Continuation

When reasoning about continuations, the issues related to the "top-level" are often swept under the rug. In Section 2.1, we stated that the reduction rules are in correspondence with the operational semantics dismissing the spurious context $\mathcal{C}\ (\lambda c.c\ \square)$ which surrounds the answer.

Griffin notes a similar situation in the type system of Figure 3. According to the operational semantics, a complete program $E[\mathcal{C}\ M]$ would rewrite to $(M\ (\lambda x.\mathcal{A}\ E[x]))$. For the right-hand side to type-check $E[x]$ must have type $\bot$, which implies that the rule is only applicable when the program $E[\mathcal{C}\ M]$ has type $\bot$. In Griffin's words, "since there are no closed terms of this type, the rule is useless!" Griffin addressed this problem by requiring that every program M be wrapped in the context $\mathcal{C}\ (\lambda c.\ c\ \square)$, which provides an explicit binding for the top-level continuation, and by modifying the operational rules to only apply in the context provided by the top-level continuation. In particular, the rule used above becomes:

$$\mathcal{C}\ (\lambda c.E[\mathcal{C}\ M]) \mapsto \mathcal{C}\ (\lambda c.M\ (\lambda x.\mathcal{A}\ E[x]))$$

Since the issue of the "top-level" becomes critical for functional continuations, it is well-worth an explicit and formal treatment. The $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus provides such a treatment by introducing a special constant tp denoting the top-level continuation.

3.2 Syntax and Semantics

The syntax of $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$ is in Figure 4. We distinguish between regular variables, continuation variables, and the special continuation constant tp. We also restrict the use of $\mathcal{C}$ such that the argument is always a λ-abstraction which binds a continuation variable and

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

Figure 4. Syntax of $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$

immediately performs a jump. The restriction does not lose expressiveness but imposes a useful structure on the terms. Indeed the $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus is isomorphic to Parigot's $\lambda\mu$-calculus [31] extended with a tp continuation constant.

With the presence of tp, the grammar distinguishes two kinds of jumps: a jump to the top-level (which aborts the program execution) and a jump to a previously defined point (which throws a value to a continuation). We abbreviate the terms which abstract over these jumps as follows:

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

$$\mathit{throw}\ k\ M \ \triangleq\ \mathcal{C}^-(\lambda\_.k\ M) \qquad \text{(Abbrev. 4)}$$

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

Figure 5. Reductions of call-by-value $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$

The reduction rules of the $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus are in Figure 5. In addition to the regular substitution operation $M[V/x]$, the rules use structural substitutions of the form $J[k\ (M\ N)/k\ M]$ from the $\lambda\mu$-calculus. Such substitutions can be read as: "for every free occurrence of k, replace the jump $(k\ M)$ by the jump $(k\ (M\ N))$. The reductions rules are closely related to the ones of $\lambda_{\mathcal{C}}$ in Figure 2 with small differences: there is a new variant of $\mathcal{C}^-_{idem}$ dealing with a regular continuation variable, there is a new rule $\mathcal{C}^-_{elim}$ that eliminates a superfluous jump whose target is the current continuation, and the rule $\mathcal{C}_{top}$ is no longer needed.

$$\begin{array}{rcl} (\mathcal{C}\ M)^\circ & = & \mathcal{C}^-(\lambda k.\mathsf{tp}\ (M^\circ\ (\lambda x.\mathit{throw}\ k\ x))) \\ (\mathcal{A}\ M)^\circ & = & \mathcal{A}^-\ M^\circ \end{array}$$

Figure 6. Embedding of $\lambda_{\mathcal{C}}$ in $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$ (interesting clauses)

There is a natural embedding of the terms of the $\lambda_{\mathcal{C}}$-calculus into the $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$-calculus, under which the reductions of one system are sound and complete with respect to the other [1]. Figure 6 shows the interesting clauses of the embedding: for all other term constructors the embedding is homomorphic. The embedding of $\mathcal{C}$ looks similar to a $\mathcal{C}_{top}$-reduction. The embedding of $\mathcal{A}$ is calculated using Abbrev. 1 followed by a simplification rule. The term in Example 1 is embedded as $\mathcal{C}^-(\lambda k.\mathsf{tp}\ ((\lambda c.\ 1\ +\ c\ 2)\ (\lambda x.\mathit{throw}\ k\ x)))$ which reduces to $\mathcal{C}^-(\lambda k.\mathsf{tp}\ (1\ +\ \mathit{throw}\ k\ 2))$.

3.3 Type System

The set of base types in $\lambda^{\rightarrow}_{\mathcal{C}^-\mathsf{tp}}$ includes $\bot$ which represents an empty type or the proposition "false" as before but it no longer plays a special role in the judgements. Instead, the judgements refer to a

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

$$
\frac{}{\Gamma, x : A \vdash x : A; T}\ Ax \qquad \frac{\Gamma, x : A \vdash M : B; T}{\Gamma \vdash \lambda x.M : A \to B; T}\ \to_i
$$

$$
\frac{\Gamma \vdash M : A \to B; T \quad \Gamma \vdash M' : A; T}{\Gamma \vdash MM' : B; T}\ \to_e
$$

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

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

Figure 7. Type system of $\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$

distinct 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.

It is possible [2] with a bit of juggling to inject a continuation of type $T \to \perp\!\!\!\perp$ into the type of *functions* $T \to \perp$ used in the Griffin's system:

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

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. 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."

In the original presentation of the type system [1], the continuation constant tp was given the type $\perp \to \perp\!\!\!\perp$. This is because the purpose in there was to characterise the computational content of (non-minimal) classical logic. A common and useful generalisation is to give the top-level continuation the type $T \to \perp\!\!\!\perp$ for some arbitrary but fixed type T [30]. We adopt this generalisation and modify the system to keep track of the special type T. Instead of providing an explicit signature for the type of the constant tp, we keep the type T on the right-hand side of the judgements where it acts as a global parameter to the type system.

The axioms and inference rules of the type system are in Figure 7. The rule *RAA* (*Reductio Ad Absurdum*) is similar to the double-negation rule $\neg\neg_E$ in Griffin's system of Figure 3 except that it uses the special type $\perp\!\!\!\perp$ instead of $\perp$. The special top-level continuation can only be invoked with a value of the distinguished type T.

The presence of the top-level type may give more information about the type of a term. For example, this judgement:

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

accurately predicts that the expression returns a bool or jumps to the top-level with an int. But in the following judgement:

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

the presence of the type string is actually restrictive and worse, misleading as we explain in Sections 6.1 and 6.2 where we talk about generalisations of the system.

The $\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$-calculus refines the $\lambda_{\mathcal{C}}$-calculus and has all the right properties.

PROPOSITION 2 (ARIOLA AND HERBELIN).

(i) *$\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$ is confluent and typed terms are strongly normalising.*

(ii) *Subject reduction: Given $\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$ terms M and N, if $\Gamma \vdash M : A; T$ and $M \to N$, then $\Gamma \vdash N : A; T$.*

(iii) *A formula A is provable in minimal classical logic iff there exists a closed $\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$ term M such that $\vdash M : A; B$ is provable without using rule $\to_e^{\mathsf{tp}}$.*

(iv) *A formula A is provable in classical logic iff there exists a closed $\lambda_{\mathcal{C}^{-}\mathsf{tp}}^{\to}$ term M such that $\vdash M : A; \perp$.*

4 Functional Continuations

We aim to extend the results for $\mathcal{C}$ to the more expressive control operators associated with delimited functional continuations. We focus on three of the basic operators: *shift* which captures a functional continuation [6], and the two identical operators *prompt* and *reset* (the first introduced by Felleisen [8] and the second introduced by Danvy and Filinski [6]) which delimit continuations.

4.1 The Operators *shift* and *reset*

The operational semantics of *shift* ($\mathcal{S}$) and *reset* (#) in the context of a call-by-value calculus is given below:

$$
\begin{array}{rcl}
x, a, v, f, c \in \mathit{Vars} & & \\
M \in \mathit{Terms} & ::= & x \mid \lambda x.M \mid MM \mid \mathcal{S}\ M \mid \#\ M \\
V \in \mathit{Values} & ::= & x \mid \lambda x.M \\
E \in \mathit{EvCtxt} & ::= & \square \mid E\ M \mid V\ E \mid \#\ E \\
\\
E[\#\ V] & \mapsto & E[V] \\
E_{\downarrow}[\#\ (E_{\uparrow}[\mathcal{S}\ M])] & \mapsto & E_{\downarrow}[\#\ (M\ (\lambda x.\#\ E_{\uparrow}[x]))]
\end{array}
$$

The first rule indicates that the role of a *reset* terminates when its subexpression is evaluated; indeed the point of *reset* is to *delimit* the continuation captured during the evaluation of its subexpression. When compared to the semantics of $\mathcal{C}$ or $\mathcal{K}$ given in Section 2.1, we note at least three points about the semantics of $\mathcal{S}$:

1. Unlike either $\mathcal{C}$ or $\mathcal{K}$, the control operator $\mathcal{S}$ only captures part of the surrounding evaluation context. This surrounding context is split into three parts: a part $E_{\uparrow}$ which extends to the *closest* occurrence of a *reset*, the occurrence of the *reset* itself, and the rest of the evaluation context $E_{\downarrow}$ surrounding the first *reset* and which may include other occurrences of *reset*. It is an error to use $\mathcal{S}$ in a context that is not surrounded by a *reset*.
2. Like $\mathcal{C}$ but unlike $\mathcal{K}$, the action of capturing the context by $\mathcal{S}$ also aborts it. In particular, the context $E_{\uparrow}$ is removed and "shifted" inside the captured continuation.

3. Unlike $\mathcal{C}$ or $\mathcal{K}$, the captured context is reified into a function which does not include an $\mathcal{A}$. Indeed, the reified continuation is a *functional continuation* whose invocation returns to the point of invocation just like an ordinary function application.

The term in Example 1 rewritten with *shift* and *reset* evaluates as follows:

$$\begin{array}{lll} \#\,\mathcal{S}\,(\lambda c.1\,+\,c\;2) & \mapsto & \#\,((\lambda c.1\,+\,c\;2)\;(\lambda x.\#\,x)) \\ & \mapsto\!\!\!\!\rightarrow & \#\,(1\,+\,\#\,2) \\ & \mapsto\!\!\!\!\rightarrow & 3 \end{array}$$

The invocation of the continuation c does not abort, but returns to the context $1\,+\,[\;]$. In general, functional continuations are *composable*. For example, $\#\;(\mathcal{S}\;(\lambda c.\;c\;(c\;1))\;+\;2)$ returns 5 but the same term with a $\mathcal{C}$ returns 3.

4.2 The Operators $\mathcal{C}$ and *prompt*

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

Figure 8. Syntax of $\lambda_{\mathcal{C}\#}$

There is an equivalent formulation of *shift* and *reset* using $\mathcal{C}$ and *prompt* (#). This representation is more suitable for our purposes since we already have all the machinery to deal with $\mathcal{C}$.

The syntax of $\lambda_{\mathcal{C}\#}$ is given in Figure 8: it consists of adding *prompt*-expressions to the syntax of $\lambda_{\mathcal{C}}$ in Figure 1. The intention is that the continuation captured by $\mathcal{C}$ will be delimited by the *prompt*. No other changes to the semantics of $\mathcal{C}$ is assumed. In particular, the continuation captured by $\mathcal{C}$ still includes an $\mathcal{A}$. Since $\mathcal{A}$ is an abbreviation for a particular use of $\mathcal{C}$, its action is also delimited by the *prompt*. This also means that it is an error to use $\mathcal{C}$ in a context that is not surrounded by a *prompt*.

$$\begin{array}{llll} \beta_v: & (\lambda x.M)V & \rightarrow & M[V/x] \\ \mathcal{C}_L: & (\mathcal{C}M)N & \rightarrow & \mathcal{C}\;(\lambda c.M\;(\lambda f.\mathcal{A}\;(c\;(f\;N)))) \\ \mathcal{C}_R: & V(\mathcal{C}M) & \rightarrow & \mathcal{C}\;(\lambda c.M\;(\lambda x.\mathcal{A}\;(c\;(Vx)))) \\ \mathcal{C}_{idem}: & \mathcal{C}\;(\lambda c.\mathcal{C}M) & \rightarrow & \mathcal{C}\;(\lambda c.M(\lambda x.\mathcal{A}\,x)) \\ \#_{\mathcal{C}} & \#(\mathcal{C}M) & \rightarrow & \#(M\;(\lambda x.\mathcal{A}\,x)) \\ \#_v & \#V & \rightarrow & V \end{array}$$

Figure 9. Reductions of call-by-value $\lambda_{\mathcal{C}\#}$

Formally, we give the semantics using local reduction rules in Figure 9. The reduction rules extend those of Figure 2 with rules for the *prompt* but omit the $\mathcal{C}_{top}$ rule. All uses of the omitted rule can be simulated with the new $\#_{\mathcal{C}}$ rule, except for the erroneous occurrences not surrounded by a *prompt*.

The use of $\lambda_{\mathcal{C}\#}$ to study functional continuations defined by *shift* and *reset* is justified by the following facts. A *prompt* is just a synonym for *reset* and the operators $\mathcal{S}$ and $\mathcal{C}$ can be mutually simulated as follows [12]:

$$\begin{array}{lll} \mathcal{S}\,M & \triangleq & \mathcal{C}\;(\lambda c.M\;(\lambda v.\#\;(c\;v))) \\ \mathcal{C}\,M & \triangleq & \mathcal{S}\;(\lambda c.M\;(\lambda x.\mathcal{A}\;(c\;x))) \end{array}$$

The mutual simulation is based on the facts that $\mathcal{A}M = \mathcal{S}\;(\lambda\_.M)$ and that for any value V, $\mathcal{A}\;(\#\,V) = \mathcal{A}\,V$ and $\#\;(\mathcal{A}\,V) = \#\,V$ [25].

4.3 Dynamic Scope of *prompt*

One of the original motivations for introducing the notion of a control delimiter is that it provides an explicit "top-level" for its subexpression [8]. It is therefore clear that the original $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathrm{tp}}$ which has a *constant* corresponding to the top-level continuation needs to be extended by making the top-level continuation a *variable*. In this section, we explain why this variable cannot be a statically-scoped variable.

Consider a possible embedding of the $\lambda_{\mathcal{C}\#}$ expression $\#M$ in a generalisation of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathrm{tp}}$ where the constant tp is replaced by a class of regular statically-scoped continuation variables $\{\;tp, tp', \ldots\;\}$:

$$\begin{array}{lll} (\#\,M)^{\circ} & = & \mathcal{C}^{-}(\lambda tp.\;tp\;M^{\circ}) \\ (\mathcal{A}\,M)^{\circ} & = & \mathcal{C}^{-}(\lambda\_.\;tp\;M^{\circ}) \end{array}$$

The embedding uses a single continuation variable tp to denote the current top-level continuation: a *prompt* expression rebinds this variable, and an $\mathcal{A}$-expression invokes it. For simple examples, this has the correct semantics but as soon as things get more complicated we run into problems.

EXAMPLE 2. (Static *prompt* vs. Dynamic *prompt*) Consider the term $\#(\#(\lambda\_.\;\mathcal{A}\;(\lambda\_.\;3))\;(\lambda\_.\;\mathcal{A}\;4))$ which reduces as follows using the $\lambda_{\mathcal{C}\#}$ reductions:

$$\rightarrow \#((\lambda\_.\;\mathcal{A}\;(\lambda\_.\;3))\;(\lambda\_.\;\mathcal{A}\;4)) \rightarrow \#\mathcal{A}\;(\lambda\_.\;3) \rightarrow\!\!\!\!\rightarrow (\lambda\_.\;3)$$

Using the suggested embedding, the corresponding $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathrm{tp}}$ term is:

$$\begin{array}{l} \mathcal{C}^{-}(\lambda tp.\;tp\;((\mathcal{C}^{-}(\lambda tp'.\;tp'\;(\lambda\_.\;\mathcal{C}^{-}(\lambda\_.\;tp'\;(\lambda\_.\;3)))))) \\ \qquad\qquad (\lambda\_.\;\mathcal{C}^{-}(\lambda\_.\;tp\;4)))) \end{array}$$

where we have α-renamed one of the occurrences of the statically-scoped variable tp. Then, if we adopt the reduction rules of the original $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathrm{tp}}$-calculus, treating tp as a regular continuation variable, we have:

$$\begin{array}{ll} \rightarrow\!\!\!\!\rightarrow_{\mathcal{C}_L^{-},\beta} & \mathcal{C}^{-}(\lambda tp.\;tp\;(\mathcal{C}^{-}(\lambda tp'.tp'\;(\mathcal{C}^{-}(\lambda\_.\;tp'\;3))))) \\ \rightarrow_{\mathcal{C}^{-}_{idem}} & \mathcal{C}^{-}(\lambda tp.\;tp\;(\mathcal{C}^{-}(\lambda tp'.\;tp'\;3))) \\ \rightarrow_{\mathcal{C}^{-}_{idem}} & \mathcal{C}^{-}(\lambda tp.\;tp\;3) \\ \rightarrow_{\mathcal{C}^{-}_{elim}} & 3 \end{array}$$

which is inconsistent with the result given in $\lambda_{\mathcal{C}\#}$.

The source of the inconsistency is the α-renaming step. Indeed, in the original $\lambda_{\mathcal{C}\#}$ term, the first occurrence of $\mathcal{A}$ is statically associated with the second occurrence of #. However, after one reduction step this occurrence of $\mathcal{A}$ is actually associated with the first occurrence of #. In $\lambda_{\mathcal{C}\#}$, when a control operation is evaluated, the evaluation refers to the *dynamically-closest* occurrence of a *prompt*. We formalise this idea in the next section.

INTERMEZZO 3. The approach of using static scope for delimiters has been explored further by Thielecke [39] and Kameyama [24]. Thielecke considers several variations of control operators with different scope rules. Using his notation, we have $\#M =$ **here** M and $\mathcal{A}\,M =$ **go** M. He presents a system with static scope $\vdash_{\mathbf{s}}$ and shows that it corresponds to classical logic. Note that β-reduction is not fully definable in his system: it has only a single **here/go** pair, so that the renaming of static **here/go** bindings that may be necessary to avoid a capture of **go** is not expressible. Kameyama also develops a type system and a semantics for a *static* variant of *shift* and *reset* and proves type soundness. His type system formalises the fact that one needs to maintain a sequence of top-level continuation variables in the judgements. Indeed, an expression might be evaluated in the scope of several occurrences of the *prompt*, each represented by a statically-scoped (α-renamed) continuation variable.

5 The $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$-Calculus: Syntax and Semantics

To maintain the proper semantics of functional continuations, a *prompt* should be mapped to a use of $\mathcal{C}^{-}$ but with the receiver being a *dynamic* λ-abstraction. We formalise this idea using an extension of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathsf{tp}}$ called the $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$-calculus.

5.1 Syntax

$$
\begin{array}{lcl}
\multicolumn{3}{l}{x,a,v,f,c \in \mathit{Vars}} \\
\multicolumn{3}{l}{k \in \mathit{StaticKVars}} \\
\multicolumn{3}{l}{\mathit{DynKVars} = \{\ \widehat{tp}\ \}} \\
M,N \in \mathit{Terms} & ::= & x \mid \lambda x.M \mid MN \mid \mathcal{C}^{-}(\lambda k.J) \mid \mathcal{C}^{-}(\lambda\widehat{tp}.\ J) \\
V \in \mathit{Values} & ::= & x \mid \lambda x.M \\
J \in \mathit{Jumps} & ::= & k\ M \mid \widehat{tp}\ M
\end{array}
$$

Figure 10. Syntax of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$

As the syntax in Figure 10 shows, we have 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 that is just like $\mathcal{A}^{-}M$ except that it refers to the special dynamic variable $\widehat{tp}$:

$$\widehat{\mathcal{A}^{-}}M \quad \triangleq \quad \mathcal{C}^{-}(\lambda\_.\widehat{tp}\ M) \qquad \text{(Abbrev. 5)}$$

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.

$$
\begin{array}{lcl}
(\#\ M)^{\circ} & = & \mathcal{C}^{-}(\lambda\widehat{tp}.\ \widehat{tp}\ M^{\circ}) \\
(\mathcal{C}\ M)^{\circ} & = & \mathcal{C}^{-}(\lambda k.\widehat{tp}\ (M^{\circ}\ (\lambda x.\mathit{throw}\ k\ x))) \\
(\mathcal{A}\ M)^{\circ} & = & \widehat{\mathcal{A}^{-}}\ M^{\circ}
\end{array}
$$

Figure 11. Embedding of $\lambda_{\mathcal{C}\#}$ into $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ (interesting clauses)

The embedding of $\lambda_{\mathcal{C}\#}$ in $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ we aim for is given in Figure 11. The embedding of a *prompt*-expression is similar to what we attempted in the last section but modified to take the dynamic nature of the *prompt* into account. The embeddings of $\mathcal{C}$ and $\mathcal{A}$ are just like the ones from Figure 6 except that they refer to $\widehat{tp}$ instead of tp.

5.2 Reductions

In order to define the semantics of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ we first need an understanding of the semantics of dynamic scope in the absence of control operators [28]. 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 $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ in Figure 12 look essentially like the reduction rules of the $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathsf{tp}}$-calculus. We discuss these rules informally here and defer the correctness argument until the next section. The first group of reductions is a copy of the corresponding reductions in $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathsf{tp}}$ with occasional dynamic annotations. As implied by DS2, the meta-operation of substitution must allow for the capture of $\widehat{tp}$. The substitution operations $M\ \widehat{[V/x]}$ and $J\ \widehat{[k\ (M\ N)/k\ M]}$ do not rename $\widehat{tp}$ but are otherwise standard. For example, if $V = (\lambda\_.\mathcal{C}^{-}(\lambda\_.\widehat{tp}\ y))$ then:

$$
\begin{array}{ll}
 & (\lambda y.\mathcal{C}^{-}(\lambda\widehat{tp}.\ \widehat{tp}\ (x\ y)))\ \widehat{[V/x]} \\
= & \lambda y'.\mathcal{C}^{-}(\lambda\widehat{tp}.\ \widehat{tp}\ ((\lambda\_.\mathcal{C}^{-}(\lambda\_.\ \widehat{tp}\ y))\ y'))
\end{array}
$$

which captures $\widehat{tp}$ but not y. The second group of rules arises because $\mathcal{C}^{-}$-abstractions come in two flavors, so we essentially need to consider every rule that is applicable to expressions of the form $\mathcal{C}^{-}(\lambda k.J)$ and modify it, if appropriate, to apply to expressions of the form $\mathcal{C}^{-}(\lambda\widehat{tp}.J)$. *We only modify and include three rules:* two *idem* rules which look as usual, and the rule $\mathcal{C}^{-}_{elim'}$ which explicitly allows occurrences of $\widehat{tp}$ in V to escape as suggested by DS1 above. The other rules are not needed to simulate the semantics of $\lambda_{\mathcal{C}\#}$.

5.3 Properties of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ Reductions

We have two semantics for $\lambda_{\mathcal{C}\#}$: one given by the reduction rules of Figure 9 and one given by the embedding in $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ and the reduction rules in Figure 12. We need to verify that the two semantics are consistent. A simple way to perform this verification would be to check that the reduction rules on one side can be simulated by the reduction rules on the other side, but this very strong correspondence does not hold in our case. What does hold is a correspondence up to *operational equivalence*.

Given a reduction relation X, and two terms M and N, possibly containing free variables, we say $M \simeq_X N$ if for every context P which binds all the free variables of M and N, $P[M] \twoheadrightarrow_X V$ iff $P[N] \twoheadrightarrow_X V$. For example, any two terms that are convertible using the reduction relation are operationally equivalent. Two non-convertible terms may still be equivalent if no sequence of reductions in any context can invalidate their equivalence. An example of this kind is the equivalence $(\lambda x.x)\ (y\ z) \simeq_{\mathcal{C}\#} (y\ z)$.

To show that the two semantics are consistent up to operational equivalence, we need to consider the following two cases. If we evaluate a $\lambda_{\mathcal{C}\#}$-program using the original semantics and that evaluation produces an answer V, then the embedding of the program in $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ should be operationally equivalent to the embedding of V. Similarly, if the evaluation of a $\lambda_{\mathcal{C}\#}$-program is erroneous or di-

$$
\begin{array}{llcll}
\beta_v: & (\lambda x.M)V & \to & M\,\hat{[}V/x\hat{]} & \\
C^-_L: & (C^-(\lambda k.J))\,N & \to & C^-(\lambda k'.J\,\hat{[}k'\,(MN)/k\,M\hat{]}) & \\
C^-_R: & V\,(C^-(\lambda k.J)) & \to & C^-(\lambda k'.J\,\hat{[}k'\,(VM)/k\,M\hat{]}) & \\
C^-_{idem'}: & C^-(\lambda k.k''\,(C^-(\lambda k'.J))) & \to & C^-(\lambda k.J\,[k''/k']) & \\
C^-_{idem}: & C^-(\lambda k.\widehat{tp}\,(C^-(\lambda k'.J))) & \to & C^-(\lambda k.J\,\hat{[}\widehat{tp}/k'\hat{]}) & \\
C^-_{elim}: & C^-(\lambda k.k\,M) & \to & M & \text{where } k \notin FV(M) \\
 & & & & \\
C^-_{idem_{\widehat{tp}}'}: & C^-(\lambda\widehat{tp}.\,k\,(C^-(\lambda k'.J))) & \to & C^-(\lambda\widehat{tp}.\,J\,[k/k']) & \\
C^-_{idem_{\widehat{tp}}}: & C^-(\lambda\widehat{tp}.\,\widehat{tp}\,(C^-(\lambda k'.J))) & \to & C^-(\lambda\widehat{tp}.\,J\,\hat{[}\widehat{tp}/k'\hat{]}) & \\
C^-_{elim'}: & C^-(\lambda\widehat{tp}.\,\widehat{tp}\,V) & \to & V & \textit{even if } \widehat{tp} \in V
\end{array}
$$

Figure 12. Reductions of call-by-value $\lambda^{\rightarrow}_{C^-\widehat{tp}}$

verges in the original semantics, then it remains undefined after the embedding. Turning the implication around, if evaluating a $\lambda_{C\#}$-program by embedding it in $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ and using the reduction rules of Figure 12 produces a value V, then the original program is operationally equivalent to the mapping of V back to $\lambda_{C\#}$. The interesting clauses of this mapping are defined as follows:

$$
\begin{array}{rcl}
(C^-(\lambda k.J))^\bullet & = & C\,(\lambda k.J^\bullet) \\
(C^-(\lambda\widehat{tp}.J))^\bullet & = & \#\,J^\bullet \\
(\widehat{tp}\,M)^\bullet & = & \mathcal{A}\,M^\bullet
\end{array}
$$

The only non-trivial clause is the first one where we silently allow continuation variables to occur in $\lambda_{C\#}$. The problem as we shall see is that continuations variables are special in $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ and the translation loses information about their special status.

PROPOSITION 3. *Let M be a* closed *$\lambda_{C\#}$-term:*

- *If $M \twoheadrightarrow_{\lambda_{C\#}} V$ then $M^\circ \simeq_{C^-\widehat{tp}} V^\circ$.*
- *If $M^\circ \twoheadrightarrow_{\lambda^{\rightarrow}_{C^-\widehat{tp}}} V$ then $M \simeq_{C\#} V^\bullet$.*

The proof of the first clause reduces to checking that embedding both sides of every $\lambda_{C\#}$-reduction produces semantically-equivalent terms in $\lambda^{\rightarrow}_{C^-\widehat{tp}}$. For some of the cases, like the reduction $\#_v$, the embedded terms are related by a corresponding reduction in $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ and hence are obviously semantically-equivalent. For the C_{idem} case, the embedded left-hand side does not reduce to the embedded right-hand side, but both can reduce to a common term, and hence are again semantically-equivalent. The lifting rules C_L are C_R introduce a complication: proving the equivalence of the embedded terms requires using the following equivalence:

$$(\lambda x.\mathit{throw}\;k\;x)\;M \quad \simeq_{C^-\widehat{tp}} \quad \mathit{throw}\;k\;M$$

even when M is not a value. This happens because in contrast to the regular substitution operation, structural substitutions can replace arbitrary jumps $(k\;M)$ by $(k\;(V\;M))$ even when M is not a value. The mismatch reflects more the design choices of $\lambda_{C\#}$ and $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ rather than an inconsistency. Indeed it would be possible to design a variant of $\lambda_{C\#}$ in which one could throw a term M to a continuation instead of a value, and it would be possible to design a variant of $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ where all arguments to jumps are restricted to be values. In general, requiring that all jump arguments be values forces one to evaluate the argument to the jump in some continuation and then erasing this continuation, instead of the equivalent but more efficient choice of first erasing the continuation and then evaluating the argument to the jump [14].

The proof of the second clause reduces to proving:

1. For all $\lambda_{C\#}$-terms M, we have $M \simeq_{C\#} M^{\circ\bullet}$
2. For every $\lambda^{\rightarrow}_{C^-\widehat{tp}}$-reduction $M \to N$, we have $M^\bullet \simeq_{C\#} N^\bullet$.

The proof of the first statement is straightforward. Note however that proving that $C\;M$ is equivalent to $(C\;M)^{\circ\bullet}$ requires using C_{top} which is not a reduction rule but otherwise a valid operational equivalence.

When attempting to prove the second statement, we encounter a problem related to free continuation variables. Even though programs are closed terms, reductions can happen anywhere including under binders and hence it is possible for a $\lambda^{\rightarrow}_{C^-\widehat{tp}}$-reduction to manipulate an open term. In particular, consider $C^-_{idem_{\widehat{tp}}'}$ where the continuation variable k is free. The right-hand side maps to the $\lambda_{C\#}$-term $(\#J[k/k']^\bullet)$ but the left-hand side is equivalent to the $\lambda_{C\#}$-term $(\#J[(\lambda x.\mathcal{A}\;(k\;x))/k']^\bullet)$. Since the variable k is not special in $\lambda_{C\#}$ it could, as far as the $\lambda_{C\#}$-theory is concerned, be substituted with 5 and hence it is definitely not the case that one can assume that k and $(\lambda x.\mathcal{A}\;(k\;x))$ are operationally equivalent. This assumption would be correct if we could somehow guarantee that k will be substituted by a continuation. In a complete program, this is clearly the case as the left-hand side must occur in a context $\ldots(C^-(\lambda k.\ldots\Box\ldots)\ldots$ which binds k to a continuation variable. We just need to make this information explicit in the statement of the proposition [34, Lemma 19]:

2'. Let $M \to N$ be a $\lambda^{\rightarrow}_{C^-\widehat{tp}}$-reduction, and let $k_1,\ldots,k_n$ be the free continuation variables in M, then we have the equivalence $C\;(\lambda k_1\ldots C\;(\lambda k_n.M^\bullet)) \simeq_{C\#} C\;(\lambda k_1\ldots C\;(\lambda k_n.N^\bullet))$.

The proof of the modified clause proceeds by cases. For the reduction C^-_{elim} we use the fact that even though C_{elim} is not a reduction of $\lambda_{C\#}$, it is a valid equivalence. Two of the $\lambda^{\rightarrow}_{C^-\widehat{tp}}$-reductions refer to substitutions using $\widehat{tp}$ in isolation. These occurrences are mapped to $\lambda x.\mathcal{A}\;x$ in $\lambda_{C\#}$ and we use the equivalence $((\lambda x.\mathcal{A}\;x)\;M) \simeq_{C\#} \mathcal{A}\;M$ with M not necessarily a value as appropriate. Finally, the reductions C^-_L and C^-_R require the following equivalences in $\lambda_{C\#}$ where k is a continuation variable and M may not be a value:

$$
\begin{array}{rcl}
(\lambda x.k\;(V\;x))\;M & \simeq_{C\#} & k\;(V\;M) \\
(\lambda x.k\;(x\;N))\;M & \simeq_{C\#} & k\;(M\;N)
\end{array}
$$

These again allow one to jump with a non-value. All the required $\lambda_{C\#}$ equivalences are known to be valid [34, 25].

6 The $\lambda^{\rightarrow}_{C^-\widehat{tp}}$-Calculus: Types

There are several natural type (and effect) systems for $\lambda^{\rightarrow}_{C^-\widehat{tp}}$ that have appeared in the literature of control operators and prompts in one form or another. We investigate three of these systems and reason about their properties. The first system has no effect annotations which we show are essential for strong normalisation.

6.1 System I: Fixed Answer Type

Even though the status of the top-level continuation changes from a constant to a dynamic variable, the type system of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\mathsf{tp}}$ in Figure 7 is essentially a sound type system for $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$. This type system can be explained on the $\lambda_{\mathcal{C}\#}$ side as an extension of Griffin's type system where the judgements have an additional type T on the right-hand side. This is identical to Murthy's type system [30] and the type system one gets when defining the control operations on top of a continuation monad with a fixed answer type T [40]. In all these systems, the rule for typing *prompt* is:

$$\frac{\Gamma \vdash M : T; T}{\Gamma \vdash \# M : T; T}\ \#$$

and the type of $\mathcal{A}$ is $T \to A$, which show that the control operators can only be used in contexts that agree with the top-level type. Despite this restriction, these types may be adequate for some applications [12] but logically they are unacceptable as they are not strong enough to guarantee strong normalisation.

EXAMPLE 4. (Loss of SN) 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 expressions:

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

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

$$f\ s \quad \twoheadrightarrow \quad (\# (g\ (\mathcal{A}\ f)))\ s \quad \twoheadrightarrow \quad (\# f)\ s \quad \twoheadrightarrow \quad f\ s$$

One can confirm this behaviour using Filinski's ML library [12] which provides *shift* and *reset* on top of a continuation monad with a fixed answer type by running the transliteration of our term:

```
let val g = fn _ => fn _ => 0
    val f = fn x => reset (fn () =>  g (x 0)) x
in f (fn _ => shift (fn _ => f ))
end
```

6.2 System II: Dynamic Scope as Effects

The requirement that all occurrences of $\widehat{tp}$ (or equivalently, all occurrences of the *prompt*) are typed with the same fixed top-level type T is overly restrictive. Each introduction of $\widehat{tp}$ can be given a different type as shown in rule $RAA^{\widehat{tp}}$ in Figure 13: 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, a judgement $\Gamma \vdash M : A; T$ can be read as term M returns a value of type A or requires its enclosing *prompt* to be of type T. On the $\lambda_{\mathcal{C}\#}$ side, this corresponds to the following rule:

$$\frac{\Gamma \vdash M : A; A}{\Gamma \vdash \# M : A; T}\ \#$$

This modification is unsound by itself as it changes the type of $\widehat{tp}$ without taking into account that it is dynamically bound. Simply adding $RAA^{\widehat{tp}}$ to the type system of Figure 7 produces a type system for the static variant of prompts: the term considered in Example 2 which evaluates to $(\lambda\_.3)$ when prompts are dynamic and to 3 when prompts are static would be given the type int.

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

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

$$\frac{\Gamma \vdash M : A \to_T B; T \quad \Gamma \vdash M' : A; T}{\Gamma \vdash MM' : B; T}\ \to_e$$

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

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

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

Figure 13. A type-and-effect system of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$

Therefore, in addition to having the 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 [17], and to add an effect annotation on *every* arrow type to pass around the type of $\widehat{tp}$.

Using the system with dynamic effect annotations, the term in Example 2 has type $A \to_B \mathsf{int}$ which is consistent with the value $\lambda\_.3$. Also the term $\lambda x.\ x + \mathcal{A}$ `"Hello"` discussed in Section 3.3 has the type $\mathsf{int} \to_{\mathsf{string}} \mathsf{int}$ with the string constraint correctly moved to when the function is *called*.

PROPOSITION 4. *Subject reduction: given $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ terms M and N if $\Gamma \vdash M : A; T$ and $M \to N$ then $\Gamma \vdash N : A; T$.*

On the $\lambda_{\mathcal{C}\#}$ side, the corresponding type-and-effect system (which we do not present) addresses the loss of strong normalisation discussed in Example 4. 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 [26] where unchecked exceptions can be used to violate strong normalisation.

More generally, we can prove that under the type-and-effect system of Figure 13, typed $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$-terms are strongly normalising. (See Proposition 11.)

INTERMEZZO 5. In the approach discussed above, different occurrences of the symbol # in a program may have different types. Gunter *et al.* [16] 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. This proposal shares with the type-and-effect system of Figure 13 that different occurrences of prompts can have different types and that the type of the prompt 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 prompt of a given name must have the same type. Moreover, since the type system has no effect annotations, Example 4 still typechecks and loops, and well-typed control operations may refer to non-existent prompts.

6.3 Understanding the Dynamic Annotations

This section presents an analysis of the effect annotations. We write $\neg T$ for the type of a continuation variable expecting an argument of type T, leaving for Section 7.1 the question of how to concretely represent this type in our language of types.

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 [28]. At the level of types, the environment-passing transformation (written *) maps $A \rightarrow_T B$ to $A^* \wedge \neg T^* \rightarrow B^*$, which means that every function is passed the (unique) environment binding as an additional argument. Judgements $\Gamma \vdash A;T$ are mapped to $\Gamma^*, \neg T^* \vdash A^*$ 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 type rule *RAA* would produce a continuation whose *input type* is $A^* \wedge \neg T^*$, 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 prompt 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 [28] and with dynamic continuation variables [16]. According to the standard monadic interpretation of exceptions [27], 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 prompt. 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 [38, 37, 33], 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 *prompt*, and later invoked under another *prompt*, should calls to $\widehat{tp}$ refer to the first *prompt* or the second? It is clear that our semantics requires the second choice. Therefore, our semantics is consistent with the SML/NJ control operators `capture` and `escape` [36]. In combination with exceptions, these control operators can simulate state [38, 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 $A^* \wedge \neg T^* \rightarrow B^* \wedge \neg T^*$ and judgements $\Gamma \vdash A;T$ are mapped to $\Gamma^*, \neg T^* \vdash A^* \wedge \neg T^*$. 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 [12, 13]. The entire analysis is also consistent with the fact that in *classical logic*, the following formulae are all equivalent:

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

6.4 System III: State as Effects

$$\begin{array}{rcl} b \in \mathit{BaseType} & = & \{\, \perp, \ldots \,\} \\ A,B,T,U \in \mathit{TypeExp} & ::= & b \mid A \;{}_U\!\!\rightarrow_T B \\ \Gamma \in \mathit{Contexts} & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, k : A \rightarrow_T \perp\!\!\!\perp \end{array}$$

$$\frac{}{\Gamma, x : A; T \vdash x : A; T}\ Ax \qquad \frac{\Gamma, x : A; U \vdash M : B; T}{\Gamma; T' \vdash \lambda x.M : A \;{}_U\!\!\rightarrow_T B; T'}\ \rightarrow_i$$

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

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

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

$$\frac{\Gamma, k : A \rightarrow_U \perp\!\!\!\perp; U \vdash M : A; T}{\Gamma, k : A \rightarrow_U \perp\!\!\!\perp \vdash k\, M : \perp\!\!\!\perp; T}\ \rightarrow_e^k \qquad \frac{\Gamma; U \vdash M : U; T}{\Gamma \vdash \widehat{tp}\, M : \perp\!\!\!\perp; T}\ \rightarrow_e^{\widehat{tp}}$$

Figure 14. Another type-and-effect system of $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$

Our type-and-effect system for $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ in Figure 13, which was motivated by the understanding of $\widehat{tp}$ as a dynamic variable, is sound but too restrictive. As the analysis in the previous section shows, $\widehat{tp}$ can also 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 of Figure 14, which is essentially identical to the one developed by Danvy and Filinski as early as 1989 [5].

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 $A \;{}_U\!\!\rightarrow_T B$ 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 $\rightarrow_e$ assumes the function is evaluated before the argument. The new system is sound.

PROPOSITION 5. *Subject reduction: given $\lambda^{\rightarrow}_{\mathcal{C}^{-}\widehat{tp}}$ terms M and N, if $\Gamma; U \vdash M : A; T$ and $M \rightarrow N$ then $\Gamma; U \vdash N : A; T$.*

Let C_{df} be the operation of changing each arrow $A \rightarrow_T B$ into $A \;{}_T\!\!\rightarrow_T B$ in the single-effect formula C. Let Γ_{df} be the extension of this operation to Γ. The new type-and-effect system generalises the previous system.

PROPOSITION 6. *If $\Gamma \vdash M : A;T$ (resp $\Gamma \vdash J : \perp\!\!\!\perp; T$) in Figure 13 then $\Gamma_{df}; T_{df} \vdash M : A_{df}; T_{df}$ (resp $\Gamma_{df} \vdash J : \perp\!\!\!\perp; T_{df}$) in Figure 14.*

The added expressiveness of the type system is illustrated with the embedding of $\#\,(1 + \mathcal{S}(\lambda c.\, 2 == c\, 3))$ which evaluates as follows:

$$\begin{array}{ll} & \mathcal{C}^{-}(\lambda \widehat{tp}.\, \widehat{tp}\ (1 + \mathcal{C}^{-}(\lambda k.\, \widehat{tp}\ (2 == \mathcal{C}^{-}(\lambda \widehat{tp}.\, k\, 3))))) \\ \rightarrow & \mathcal{C}^{-}(\lambda \widehat{tp}.\, \widehat{tp}\ (\mathcal{C}^{-}(\lambda k.\, \widehat{tp}\ (2 == \mathcal{C}^{-}(\lambda \widehat{tp}.\, k\, (1+3)))))) \\ \rightarrow & \mathcal{C}^{-}(\lambda \widehat{tp}.\, \widehat{tp}\ (2 == \mathcal{C}^{-}(\lambda \widehat{tp}.\, \widehat{tp}\ (1+3)))) \\ \twoheadrightarrow & \mathcal{C}^{-}(\lambda \widehat{tp}.\, \widehat{tp}\ (2 == 4)) \twoheadrightarrow \mathsf{false} \end{array}$$

This term is rejected by the system in Figure 13 but accepted in the system of Figure 14. 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.

7 The $\lambda_{\mathcal{C}}^{\rightarrow -}$-Calculus

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

7.1 Subtraction

Our analysis of Section 6.3 together with our understanding of the type system of Figure 14 suggest we interpret the more general effect annotations as follows:

$$\begin{array}{rcl} (A \; {}_U{\rightarrow}_T \; B)^* & = & A^* \wedge \neg T^* \rightarrow B^* \wedge \neg U^* \\ (\Gamma; U \vdash A; T)^* & = & \Gamma^*, \neg T^* \vdash A^* \wedge \neg U^* \end{array}$$

This is possible, but it can be refined as we discuss next.

First, the type of a continuation variable is really of the form $T^* \rightarrow \perp\!\!\!\perp$. While this type can be injected into $T^* \rightarrow \perp$ as shown in Section 3.3, doing so loses important information by injecting the special continuations into the domain of regular functions.

Second, the formula $A \wedge \neg T$ is classically the same as $\neg(\neg A \vee T) = \neg(A \rightarrow T)$, *i.e.*, it represents the *dual of a function type*. This *subtractive type* $A - T$ has been previously studied by Rauszer [32] and Crolard [2], and has been integrated by Curien and Herbelin [4] 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). In many cases, the dual of a function type appears as a technical formality. Here it arises as the natural type for pairing a term and a continuation and we prefer to use it as a more "abstract" representation of this information. Indeed in an intuitionistic setting where continuations are not first-class values, it would still make sense to use the type $A - T$ and in that case the type would *not* be equivalent to $A \wedge \neg T$ [3]. Dually, the types $A \rightarrow B$ and $\neg A \vee B$ are classically (but not intuitionistically) equivalent, and we generally prefer to describe functions using the type $A \rightarrow B$ which does not rely on the presence of first-class continuations.

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

Figure 15. Interpreting the effect annotations

The actual interpretation of effects we use in Figure 15 incorporates both concerns. The interpretation shows that a continuation k has type $A - T \rightarrow \perp\!\!\!\perp$. Understanding $A - T$ as the type $A \wedge \neg T$ means that, as Danvy and Filinski explain, every functional continuation must be given a value and another continuation (called the meta-continuation in their original article [6] and referring to the top-level continuation as we explain). The type $A - T$ is also equivalent to $\neg(\neg T \rightarrow \neg A)$ and with that view, a functional continuation is more like a continuation transformer, which is an idea closely related to Queinnec and Moreau's formalisation of functional continuations as the difference between two continuations [29].

7.2 Syntax, Semantics, and Type System

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

Figure 16. Syntax of $\lambda_{\mathcal{C}}^{\rightarrow -}$

We formalise the $\lambda_{\mathcal{C}}^{\rightarrow -}$-calculus by extending $\lambda_{\mathcal{C}\,\mathsf{tp}}^{\rightarrow}$ 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 [2].

The syntax is given in Figure 16. Terms include two new forms that introduce and eliminate the subtractive type. As expected from the equivalence $A - T = A \wedge \neg T$, the introduction form $(M, k\, E)$ is the pairing of a term and a continuation. The continuation is not a simple variable k however: it is a more general *jump context* $(k\, E)$ where E is an evaluation context accumulated by the reduction rules of Figure 17 during evaluation. Evaluation contexts E are defined using nestings of elementary contexts F ending with a hole. This formulation simplifies the presentation of reduction rules: for example, instead of having individual rules which lift $\mathcal{C}^-$ across each elementary context as in the cases of $\mathcal{C}_L^-$ and $\mathcal{C}_R^-$ before, we express all the lifting rules using the single rule $\mathcal{C}_{\mathit{lift}}^-$.

The elimination form **let** $(x,k) = M$ **in** M' takes apart a subtractive value M as follows. When the term M evaluates to something of the shape $(V, k'E)$, the jump context $k'E$ is bound to k and V is bound to x, and the evaluation proceeds with M'. The two reduction rules $\mathit{Sub}_v{}^{\mathit{base}}$ and $\mathit{Sub}_v{}^{\mathit{step}}$ express this semantics one elementary context at a time. For example, we have the following reduction sequence:

$$\begin{array}{ll} & \textbf{let}\ (x,k) = (V, k'\ (F_1[F_2[F_3\Box]]))\ \textbf{in}\ \mathit{throw}\ k\ x \\ \rightarrow & \textbf{let}\ (x,k) = (V, k'\ (F_1[F_2\Box]))\ \textbf{in}\ \mathit{throw}\ k\ (F_3[x]) \\ \rightarrow & \textbf{let}\ (x,k) = (V, k'\ (F_1\Box))\ \textbf{in}\ \mathit{throw}\ k\ (F_2[F_3[x]]) \\ \rightarrow & \textbf{let}\ (x,k) = (V, k'\ \Box)\ \textbf{in}\ \mathit{throw}\ k\ (F_1[F_2[F_3[x]]]) \\ \rightarrow & \mathit{throw}\ k'\ (F_1[F_2[F_3[V]]]) \end{array}$$

To help readability, we use the following abbreviations:

$$\begin{array}{rcll} \lambda(x,k).M & \triangleq & \lambda v.\textbf{let}\ (x,k) = v\ \textbf{in}\ M & \text{(Abbrev. 6)} \\ \textbf{join}\ M & \triangleq & \textbf{let}\ (x,k) = M\ \textbf{in}\ \mathit{throw}\ k\ x & \text{(Abbrev. 7)} \\ \textbf{bind}_k\ M\ \textbf{in}\ N & \triangleq & \textbf{let}\ (f,k) = M\ \textbf{in}\ f\ N & \text{(Abbrev. 8)} \end{array}$$

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.

β_v :	$(\lambda x.M)V$	$\rightarrow$	$M[V/x]$	
$\mathcal{C}^-_{lift}$:	$F[\mathcal{C}^-(\lambda k.J)]$	$\rightarrow$	$\mathcal{C}^-(\lambda k.J\ [k\ (F[M])/k\ M;\ \ k\ (F[E])/k\ E])$	
$\mathcal{C}^-_{elim}$:	$\mathcal{C}^-(\lambda k.k\ M)$	$\rightarrow$	M	where $k \notin FV(M)$
$\mathcal{C}^-_{idem}$:	$\mathcal{C}^-(\lambda k.k''\ (\mathcal{C}^-(\lambda k'.J)))$	$\rightarrow$	$\mathcal{C}^-(\lambda k.J\ [k''/k'])$	
$Sub_v{}^{base}$	$\textbf{let}\ (x,k) = (V, k'\Box)\ \textbf{in}\ M$	$\rightarrow$	$M[k'/k][V/x]$	
$Sub_v{}^{step}$	$\textbf{let}\ (x,k) = (V, k'(E[F]))\ \textbf{in}\ M$	$\rightarrow$	$\textbf{let}\ (x,k) = (V, k'E)\ \textbf{in}\ M[k\ (F[M])/k\ M;\ \ k\ (F[E])/k\ E]$	

Figure 17. Reductions of call-by-value $\lambda_{\mathcal{C}^-}^{\rightarrow -}$

The typing rules for the complete language are in Figure 18. The system is completely standard with no effect annotations and not even a global parameter T. To type the introduction rule of subtraction, we need to type evaluation contexts E with rules essentially identical to the rules for typing terms. We use the notation $\Gamma, \Box : B \vdash k\,E : \perp\!\!\!\perp$ to denote any judgement of the form $\Gamma, x : B \vdash k(E[x]) : \perp\!\!\!\perp$ up to the name x which is a fresh variable.

$$\begin{array}{rcl} b \in BaseType & = & \{\ \perp, \dots\ \} \\ A,B,C,T,U \in TypeExp & ::= & b \mid A \rightarrow B \mid A - B \\ \Gamma \in Contexts & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, k : A \rightarrow \perp\!\!\!\perp \end{array}$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ Ax \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \rightarrow B}\ \rightarrow_i$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash M' : A}{\Gamma \vdash MM' : B}\ \rightarrow_e$$

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

$$\frac{\Gamma \vdash M : A \quad \Gamma, \Box : B \vdash k\,E : \perp\!\!\!\perp}{\Gamma \vdash (M, k\,E) : A - B}\ -_i$$

$$\frac{\Gamma \vdash M : A - C \qquad \Gamma, x : A, k : C \rightarrow \perp\!\!\!\perp \vdash M' : B}{\Gamma \vdash \textbf{let}\ (x,k) = M\ \textbf{in}\ M' : B}\ -_e$$

Figure 18. Type system of $\lambda_{\mathcal{C}^-}^{\rightarrow -}$

Using standard proof techniques, we get

PROPOSITION 7.

1. *Subject reduction: Given $\lambda_{\mathcal{C}^-}^{\rightarrow -}$ terms M and N if $\Gamma \vdash M : A$ and $M \rightarrow N$ then $\Gamma \vdash N : A$.*
2. *Typed $\lambda_{\mathcal{C}^-}^{\rightarrow -}$ terms are strongly normalising.*

8 Embeddings in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$

We show that the interpretation of effect annotations using the subtractive type is correct. First, we embed the terms of $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$ and show that the embedding respects the semantics of $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ and hence the semantics of $\lambda_{\mathcal{C}\#}$. Second, we embed $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$-judgements in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$-judgements and check that the embedding is type-preserving.

8.1 Embedding $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ and $\lambda_{\mathcal{C}\#}$ terms in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$

We first consider the embedding of $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ terms in Figure 19. The embedding of terms and jumps is parameterised by a single fresh *statically-scoped* continuation name tp in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$ which occurs free in the right-hand side. The dynamic variable $\widehat{tp}$ of $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ is eliminated in the translation which arranges for tp to always refer to the current top-level continuation. The translation of a value has no free occurrences of the continuation variable used in the embedding: the translation of every λ-expression introduces a new binding for tp which is used in the translation of its body. 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. Because translated values have no free occurrences of tp, it is easy to verify that $M[V/x]^{tp} = M^{tp}[V^+/x]$.

The main difficulty in defining the embedding of terms and jumps is to correctly propagate tp. A value V is embedded as a subtractive value whose jump context is the current top-level continuation. In the embedding of MN, the occurrences of tp in N^{tp} are bound by the $\textbf{bind}_{tp}$, while the occurrences of tp in M^{tp} are free. These relationships mimic the facts 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. For the embedding of $\mathcal{C}^-(\lambda k.J)$, we note that because all the right-hand sides of the embedding return terms of the subtractive type, a continuation k in $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ is translated to a continuation which expects a subtractive value as its argument. Hence, jumps to k return not only a regular result but also transfer the current binding of the top-level continuation to the context surrounding $\mathcal{C}^-(\lambda k.J)$.

$$\begin{array}{rcl} V^{tp} & = & (V^+, tp\,\Box) \\ (MN)^{tp} & = & \textbf{bind}_{tp}\ M^{tp}\ \textbf{in}\ N^{tp} \\ (\mathcal{C}^-(\lambda k.J))^{tp} & = & \mathcal{C}^-(\lambda k.J^{tp}) \\ (\mathcal{C}^-(\lambda \widehat{tp}.J))^{tp} & = & (\mathcal{C}^-(\lambda tp.J^{tp}), tp\,\Box) \\ & & \\ x^+ & = & x \\ (\lambda x.M)^+ & = & \lambda(x,tp).M^{tp} \\ & & \\ (k\,M)^{tp} & = & k\,M^{tp} \\ (\widehat{tp}\,M)^{tp} & = & tp\ (\textbf{join}\ M^{tp}) \end{array}$$

Figure 19. Embedding of $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ terms in $\lambda_{\mathcal{C}^-}^{\rightarrow -}$ terms

The two most interesting clauses are the ones involving $\widehat{tp}$. For the embedding of $\mathcal{C}^-(\lambda\widehat{tp}.J)$, we return 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. Because of the context in which $\mathcal{C}^-(\lambda tp \dots)$ occurs in the right-hand side it is clear 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. In the special case, where we jump to the top-level with a value, we have $(\widehat{tp}\,V)^{tp} \rightarrow tp\,V^+$.

Combining the embedding from $\lambda_{\mathcal{C}\#}$ to $\lambda_{\mathcal{C}\widehat{tp}}^{\rightarrow}$ (Figure 6) with the em-

$$
\begin{array}{lcl}
x^+ & = & x \\
(\lambda x.M)^+ & = & \lambda(x,tp).M^{tp} \\
 & & \\
V^{tp} & = & (V^+, tp) \\
(MN)^{tp} & = & \mathbf{bind}_{tp}\ M^{tp}\ \mathbf{in}\ N^{tp} \\
(\mathcal{C}\ M)^{tp} & = & \mathcal{C}^-(\lambda k.\ tp\ (\mathbf{join} \\
 & & \qquad\qquad (\mathbf{bind}_{tp'}\ M^{tp} \\
 & & \qquad\qquad \mathbf{in}\ (\lambda(x,tp).throw\ k\ (x, tp\ \square) \\
 & & \qquad\qquad\quad , tp'\ \square)))) \\
(\#\ M)^{tp} & = & (\mathcal{C}^-(\lambda tp.\ tp\ (\mathbf{join}\ M^{tp})), tp\ \square)
\end{array}
$$

Figure 20. Embedding of $\lambda_{\mathcal{C}\#}$ terms in $\lambda_{\mathcal{C}}^{\rightarrow -}$ terms

bedding we just defined, produces a direct embedding from $\lambda_{\mathcal{C}\#}$ into $\lambda_{\mathcal{C}}^{\rightarrow -}$ shown in Figure 20. The translation of $\mathcal{C}\ M$ is the most complicated: the continuation k is first captured and then we jump to the top-level with a computation that first evaluates M. Once M produces a value f and perhaps a new top-level continuation tp', we apply f to a subtractive value consisting of the reified continuation and tp'. When the reified continuation is invoked on an argument P, it is given the value and top-level continuation resulting from the evaluation of P which it passes to k aborting its context.

Given the equivalence of $\mathcal{C}$ and $\mathcal{S}$, it is easy to calculate that the embedding of $\mathcal{S}$ differs only in the details of the reified continuation which becomes:

$$\lambda(x,tp).\mathcal{C}^-(\lambda tp''.k\ (x, tp''\ (\square, tp\square)))$$

Thus instead of ignoring the top-level continuation tp'' at the call site like in the case of $\mathcal{C}$, this continuation composes the top-level continuations, first returning to tp and then returning to tp''.

EXAMPLE 6. (Embedding in $\lambda_{\mathcal{C}}^{\rightarrow -}$) If the top-level continuation is tp, the embedding of the $\lambda_{\mathcal{C}\#}$ term $(y\ (\#\ x))$ which corresponds to the $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ term $(y\ (\mathcal{C}^-(\lambda\widehat{tp}.\ \widehat{tp}\ x)))$ is:

$$
\begin{array}{lcl}
(y\ (\mathcal{C}^-(\lambda\widehat{tp}.\ \widehat{tp}\ x)))^{tp} & \rightarrow & y\ ((\mathcal{C}^-(\lambda\widehat{tp}.\ \widehat{tp}\ x))^{tp}) \\
 & \rightarrow & y\ ((\mathcal{C}^-(\lambda tp'.tp'\ x), tp\ \square))
\end{array}
$$

The current top-level continuation is "saved" using the subtraction introduction and a new top-level continuation is used for receiving the value of x.

The embedding of $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ into $\lambda_{\mathcal{C}}^{\rightarrow -}$ is sound in the sense that embedding a $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ reduction yields a reduction sequence in $\lambda_{\mathcal{C}}^{\rightarrow -}$. This does *not* imply however that there is any relationship between $\lambda_{\mathcal{C}\#}$ reductions and $\lambda_{\mathcal{C}}^{\rightarrow -}$ reductions under the combined embedding of Figure 20 because the intermediate Proposition 3 only provides soundness up to operational equivalence. The notation $\twoheadrightarrow^{\geq 1}$ indicates a reduction sequence of at least one step.

PROPOSITION 8. *A reduction $M \rightarrow N$ in $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ implies a sequence of reductions $M^{tp} \twoheadrightarrow^{\geq 1} N^{tp}$ in $\lambda_{\mathcal{C}}^{\rightarrow -}$ for any fresh continuation variable tp.*

The theory $\lambda_{\mathcal{C}}^{\rightarrow -}$ is richer than $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$, so that the translation is not complete. Indeed, the dynamically-bound continuation $\widehat{tp}$ is turned into an ordinary statically-bound continuation: the limitations we imposed on lifting $\mathcal{C}^-(\lambda\widehat{tp}\ldots)$ in Section 5.2 can no longer be enforced. For instance, the term in Example 6 can be further reduced as follows:

$$
\begin{array}{cl}
 & y\ ((\mathcal{C}^-(tp'.tp'\ x), tp\ \square)) \\
\overset{\mathcal{C}_{lift}}{\rightarrow} & y\ (\mathcal{C}^-(tp'.tp'\ (x, tp\ \square))) \\
\overset{\mathcal{C}_{lift}}{\rightarrow} & \mathcal{C}^-(tp'.tp'\ (y\ (x, tp\ \square)))
\end{array}
$$

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

8.2 Embedding $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ Judgements

To reason about type-safety we present the embedding of $\lambda^{\rightarrow}_{\mathcal{C}^-\widehat{tp}}$ judgements into $\lambda_{\mathcal{C}}^{\rightarrow -}$. We present it for the more general system of Figure 14 since the less general system of Figure 13 comes by requiring both annotations to be the same. In the translation, the type of the dynamically-bound *prompt* is the type of the fresh variable tp used for the translation and the type of the *prompt* returned by the main expression (if any) is made explicit thanks to the subtraction connective:

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

Let us write $\Lambda^{\rightarrow\ eff_{eq}}_{\mathcal{C}^-\widehat{tp}}$, $\Lambda^{\rightarrow\ eff}_{\mathcal{C}^-\widehat{tp}}$ and $\Lambda_{\mathcal{C}}^{\rightarrow -}$ for the type systems on Figures 13, 14 and 18 respectively. For tp a fresh continuation variable, the translation above is sound.

PROPOSITION 9.

(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 $\Lambda_{\mathcal{C}}^{\rightarrow -}$,

(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 $\Lambda_{\mathcal{C}}^{\rightarrow -}$.

We can also consider the extension from Section 6.1 of the type system of Figure 7: let $\Lambda^{\rightarrow\ fixed}_{\mathcal{C}^-\widehat{tp}}$ be the system of Figure 7 with tp changed into the dynamic continuation variable $\widehat{tp}$ that can be abstracted by the rule

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

Let A_T be the operation of adding twice the same *atomic* effect T on the occurrences of $\rightarrow$ in the effect-free formula A. Let Γ_T be the extension of this operation to an effect-free Γ. We have:

PROPOSITION 10. *Let T be atomic. If $\Gamma \vdash M : A; T$ in $\Lambda^{\rightarrow\ fixed}_{\mathcal{C}^-\widehat{tp}}$ then $\Gamma_T; T \vdash M : A_T; T$ in $\Lambda^{\rightarrow\ eff}_{\mathcal{C}^-\widehat{tp}}$. Hence $(\Gamma_T; T \vdash M : A_T; T)^{tp}$ in $\Lambda_{\mathcal{C}}^{\rightarrow -}$.*

Collecting the previous results with Proposition 7 and 8, we have:

PROPOSITION 11.

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

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

Especially, the last item says that we cannot enforce strong normalisation as soon as 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 4.

9 Other Control Operators

We have focused exclusively on functional continuations obtained with *shift* (or $\mathcal{C}$) and *reset*. We briefly review and classify some of the other control operators in the literature and discuss them based on our work.

9.1 A Short History

As we have seen, early proposals for functional continuations [10, 8, 5] had only a *single* control delimiter. The operation like *shift* for capturing the functional continuation implicitly refers to the most recent occurrence of this delimiter.

The limitations of the single control delimiter however became quickly apparent, and later proposals generalised the single delimiter by allowing hierarchies of prompts and control operators like $reset_n$ and $shift_n$ [6, 35]. At about the same time, a different proposal *spawn* allowed new prompts to be generated dynamically [21, 20]. In this system, the base of each functional continuation is rooted at a different *prompt*. The action of creating the *prompt* returns a specialised control operator for accessing occurrences of this particular *prompt*; this specialised control operator can then be used for capturing (and aborting) the particular functional continuation rooted at the newly generated *prompt* (and only that one). This is more expressive and convenient than either single prompts or hierarchies of prompts and allows arbitrary nesting and composition of continuation-based abstractions. A later proposal by Gunter *et. al.* [16] separated the operation for *creating* new prompts from the control operator *using* that *prompt*.

9.2 Prompts and Extent

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

However, our analysis fundamentally relies on a subtle issue related to the *extent* of prompts [29]. More precisely, there is no question that the *prompt* 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 [7]:

$$\begin{array}{lcl}
E_\downarrow[\#\,(E_\uparrow[\mathcal{F}_1\,M])] & \mapsto & E_\downarrow[M\,(\lambda x.E_\uparrow[x])] \\
E_\downarrow[\#\,(E_\uparrow[\mathcal{F}_2\,M])] & \mapsto & E_\downarrow[\#\,(M\,(\lambda x.E_\uparrow[x]))] \\
E_\downarrow[\#\,(E_\uparrow[\mathcal{F}_3\,M])] & \mapsto & E_\downarrow[M\,(\lambda x.\#\,(E_\uparrow[x]))] \\
E_\downarrow[\#\,(E_\uparrow[\mathcal{F}_4\,M])] & \mapsto & E_\downarrow[\#\,(M\,(\lambda x.\#\,(E_\uparrow[x])))]
\end{array}$$

All four variants have been proposed in the literature: $\mathcal{F}_1$ is like *cupto* [16]; $\mathcal{F}_2$ is Felleisen's $\mathcal{F}$ operator [10]; $\mathcal{F}_3$ is like a *spawn* controller [21]; and $\mathcal{F}_4$ is *shift* [6].

It turns out that the inclusion of the *prompt* 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 *prompt* 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 *prompt*; so when the continuation is invoked, we must "search" for the *prompt* required to denote the top-level. In general, the *prompt* can be located arbitrarily deep in the calling context. Since the representation of contexts as functional continuations does not support operations for "searching for prompts," Felleisen *et al.* [11] developed a special model based on an *algebra of contexts* which supports the required operations. In a recent investigation of control operators for functional continuations, Dybvig *et al.* [7] 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 prompts*. They in fact present an implementation in Haskell that we believe can be adapting as the basis for an embedding in a variant (or extension) of $\lambda_{\mathcal{C}}^{\rightarrow -}$.

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 reviewers who provided corrections and extensive comments on the presentation.

10 References

[1] Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *LNCS*, pages 128–136, Eindhoven, The Netherlands, June 30 - July 4, 2003. Springer-Verlag,

[2] T. Crolard. Subtractive logic. *Theor. Comput. Sci.*, 254(1-2):151–185, 2001.

[3] T. Crolard. A formulae-as-types interpretation of subtractive logic. *Journal of Logic and Computation (Special issue on Modalities in Constructive Logics and Type Theories)*, 2004. To appear.

[4] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the 5th International Conference on Functional Programming*, pages 233–243, 2000.

[5] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

[6] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 151–160, New York, NY, 1990. ACM.

[7] R. K. Dybvig, S. Peyton-Jones, E. Moggi, A. Sabry, and O. Waddell. Monadic functional subcontinuations. Unpublished manuscript, 2004.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 180–190, Jan 1988.

[9] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, New York, N.Y., 1990.

[10] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[11] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Conference on LISP and Functional Programming, Snowbird, Utah*, pages 52–62. ACM, 1988.

[12] A. Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'94) , Portland, OR, USA, 17–21 Jan. 1994*, pages 446–457, New York, 1994. ACM Press.

[13] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'99)*, San Antonio, Texas, pages 175–188. ACM Press, 1999.

[14] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proceedings of the 4th ACM SIGPLAN international conference on Functional programming*, pages 18–27. ACM Press, 1999.

[15] T. G. Griffin. The formulae-as-types notion of control. In *Proceedings of the 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, S an Francisco, CA, USA, 17–19 Jan 1990*, pages 47–57, New York, 1990. ACM Press.

[16] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming & Computer Architecture*, New York, 1995. ACM Press.

[17] J. Guzmán and A. Suárez. An extended type system for exceptions. In *Record of the 5th ACM SIGPLAN workshop on ML and its Applications*, June 1994. Also appears as Research Report 2265, INRIA, BP 105 - 78153 Le Chesnay Cedex, France.

[18] C. T. Haynes. Logic continuations. In *Proceedings of the 3th International Conference on Logic Programming*, volume 225 of *LNCS*, pages 671–685, Berlin, July 1986. Springer-Verlag.

[19] C. T. Haynes, D. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11:143–153, 1986.

[20] R. Hieb, K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[21] R. Hieb and R. K. Dybvig. Continuations and Concurrency. In *PPoPP '90, Symposium on Principles and Practice of Parallel Programming*, volume 25(3) of *SIGPLAN NOTICES*, pages 128–136, Seattle, Washington, March 14–16, 1990. ACM Press.

[22] W. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[23] Y. Kameyama. A type-theoretic study on partial continuations. In *IFIP TCS*, pages 489–504, 2000.

[24] Y. Kameyama. Towards logical understanding of delimited continuations. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, 2001.

[25] Y. Kameyama and M. Hasegawa. 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*, volume 38(9) of *SIGPLAN Notices*, pages 177–188. ACM Press, New York, 2003.

[26] M. Lillibridge. Unchecked exceptions can be strictly more powerful than call/cc. *Higher-Order and Symbolic Computation*, 12(1):75–104, Apr. 1999.

[27] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press, 1989.

[28] L. Moreau. A syntactic theory of dynamic binding. *Higher Order Symbol. Comput.*, 11(3):233–279, 1998.

[29] L. Moreau and C. Queinnec. Partial Continuations as the Difference of Continuations. A Duumvirate of Control Operators. In *International Conference on Programming Language Implementation and Logic Programming (PLILP'94)*, number 844 in *LNCS* pages 182–197, Madrid, Spain, Sept. 1994. Springer-Verlag.

[30] C. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *ACM workshop on Continuations*, pages 49–71, 1992.

[31] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of International Conference on Logic Programming and Automated Reasoning (LPAR '92)*, St. Petersburg, Russia, pages 190–201. Springer-Verlag, 1992.

[32] C. Rauszer. Semi-boolean algebras and their application to intuitionistic logic with dual connectives. *Fundamenta Mathematicae*, 83:219–249, 1974.

[33] J. G. Riecke and H. Thielecke. Typed exceptions and continuations cannot macro-express each other. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 635–644, Berlin, 1999. Springer-Verlag.

[34] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp Symb. Comput.*, 6(3-4):289–360, 1993.

[35] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[36] The SML/NJ library. Available from `http://smlnj.org/`.

[37] H. Thielecke. On exceptions versus continuations in the presence of state. In *Proceedings of the 9th European Symposium On Programming (ESOP)*, volume 1782 of *LNCS*, pages 397–411, Berlin, 2000. Springer-Verlag.

[38] H. Thielecke. Contrasting exceptions and continuations. Version available from `http://www.cs.bham.ac.uk/~hxt/research/exncontjournal.pdf`, 2001.

[39] H. Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):119–136, 2002.

[40] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, Jan. 1994.

[41] M. Wand. Continuation-based multiprocessing revisited. *Higher-Order and Symbolic Computation*, 12(3):285–299, Oct. 1999. Reprinted from the proceedings of the 1990 Lisp Conference, with a foreword.