Electronic Notes in Theoretical Computer Science 45 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume45.html 15 pages

Comparing Control Constructs by Double-barrelled CPS Transforms

Hayo Thielecke

H.Thielecke@cs.bham.ac.uk
School of Computer Science
University of Birmingham
Birmingham B15 2TT
United Kingdom

Abstract

We investigate continuation-passing style transforms that pass two continuations. Altering a single variable in the translation of λ-abstraction gives rise to different control operators: first-class continuations; dynamic control; and (depending on a further choice of a variable) either the `return` statement of C; or Landin's **J**-operator. In each case there is an associated simple typing. For those constructs that allow upward continuations, the typing is classical, for the others it remains intuitionistic, giving a clean distinction independent of syntactic details.

1 Introduction

Control operators come in bewildering variety. Sometimes the same term is used for distinct constructs, as with `catch` in early Scheme or `throw` in Standard ML of New Jersey, which are very unlike the `catch` and `throw` in Lisp whose names they borrow. On the other hand, this Lisp `catch` is fundamentally similar to exceptions despite their dissimilar and much more ornate appearance.

Fortunately it is sometimes possible to glean some high-level "logical" view of a programming language construct by looking only at its type. Specifically for control operations, Griffin's discovery [3] that `call/cc` and related operators can be ascribed classical types gives us the fundamental distinction between languages that have such classical types and those that do not, even though they may still enjoy some form of control. This approach complements comparisons based on contextual equivalences [10,14].

Such a comparison would be difficult unless we blot out complication. In particular, exceptions are typically tied in with other, fairly complicated features of the language which are not relevant to control as such: in ML with the datatype mechanism, in Java with object-orientation. In order to simplify, we first strip down control operators to the bare essentials of labelling and jumping, so that there are no longer any distracting syntactic differences between them. The grammar of our toy language is uniformly this:

$$M ::= x \mid \lambda x.M \mid MM \mid \mathtt{here}\, M \mid \mathtt{go}\, M.$$

The intended meaning of `here` is that it labels a "program point" or expression without actually naming any particular label—just uttering the demonstrative "here", as it were. Correspondingly, `go` jumps to a place specified by a `here`, without naming the "to" of a `goto`.

Despite the simplicity of the language, there is still scope for variation: not by adding bells and whistles to `here` and `go`, but by varying the meaning of λ-abstraction. Its impact can be seen quite clearly in the distinction between exceptions and first-class continuations. The difference between them is as much due to the meaning of λ-abstraction as due to the control operators themselves, since λ-abstraction determines what is statically put into a closure and what is passed dynamically. Readers familiar with, say, Scheme implementations will perhaps not be surprised about the impact of what becomes part of a closure. But the point of this paper is twofold:

- small variations in the meaning of λ completely change the meaning of our control operators;
- we can see these differences at an abstract, logical level, without delving into the innards of interpreters.

We give meaning to the λ-calculus enriched with `here` and `go` by means of continuations in Section 2, examining in Sections 3–5 how variations on λ-abstraction determine what kind of control operations `here` and `go` represent. For each of these variations we present a simple typing, which agrees with the transform (Section 6). We conclude by explaining the significance of these typings in terms of classical and intuitionistic logic (Section 7).

2 Double-barrelled CPS

Our starting point is a continuation-passing style (CPS) transform. This transform is double-barrelled in the sense that it always passes *two* continuations. Hence the clauses start with $\lambda kq. \ldots$ instead of $\lambda k. \ldots$. Other than that, this CPS transform is in fact a very mild variation on the usual call-by-value one [8]. As indicated by the $\boxed{?}$, we leave one variable, the extra continuation passed to the body of a λ-abstraction, unspecified.

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{?}x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{?}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\, M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

The extra continuation may be seen as a jump continuation, in that its manipulation accounts for the labelling and jumping. This is done symmetrically: `here` makes the jump continuation the same as the current one k, whereas `go` sets the current continuation of its argument to the jump continuation q. The clauses for variables and applications do not interact with the additional jump continuation: the former ignores it, while the latter merely distributes it into the operator, the operand and the function call.

Only in the clause for λ-abstraction do we face a design decision. Depending on which continuation (static s, dynamic d, or the return continuation r) we fill in for "?" in the clause for λ, there are three different flavours of λ-abstraction.

$$\begin{aligned}
[\![\lambda_{\mathsf{s}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s})\\
[\![\lambda_{\mathsf{d}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d})\\
[\![\lambda_{\mathsf{r}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r})
\end{aligned}$$

The lambdas are subscripted to distinguish them, and the box around the last variable is meant to highlight that this is the crucial difference between the transforms. Formally there is also a fourth possibility, the outer continuation k, but this seems less meaningful and would not fit into simple typing.

For all choices of λ, the operation `go` is always a jump to a place specified by a `here`. For example, for any M, the term `here` $((\lambda x.M)($`go` $N))$ should be equivalent to N, as the `go` jumps past the M. But in more involved examples than this, there may be different choices *where* `go` can go to among several occurrences of `here`. In particular, if s is passed as the second continuation argument to M in the transform of $\lambda x.M$, then a `go` in M will refer to the `here` that was in scope at the point of definition (unless there is an intervening `here`, just as one binding of a variable x can shadow another). By contrast, if d is passed to M in $\lambda x.M$, then the `here` that is in scope at the point of definition is forgotten; instead `go` in M will refer to the `here` that is in scope at the point of call when $\lambda x.M$ is applied to an argument. In fact, depending upon the choice of variable in the clause for λ as above, `here` and `go` give rise to different control operations:

- first-class continuations like those given by `call/cc` in Scheme [4];
- dynamic control in the sense of Lisp, and typeable in a way reminiscent of checked exceptions;
- a `return`-operation, which can be refined into the **J**-operator invented by Landin in 1965 and ancestral to `call/cc` [4,6,7,13].

We examine these constructs in turn, giving a simple type system in each case. An unusual feature of these type judgements is that, because we have two continuations, there are two types in the succedent on the right of the turnstile, as in

$$\Gamma \vdash M : A, B.$$

The first type on the right accounts for the case that the term returns a value; it corresponds to the current continuation. The second type accounts for the

Fig. 1. Typing for static `here` and `go`

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{s}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathtt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathsf{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} M : B, C}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : A \to B, C \qquad \Gamma \vdash_{\mathsf{s}} N : A, C}{\Gamma \vdash_{\mathsf{s}} MN : B, C}$$

jump continuation. In logical terms, the comma on the right may be read as a disjunction. It makes a big difference whether this disjunction is classical or intuitionistic. That is our main criterion of comparing and contrasting the control constructs.

3 First-class continuations

The first choice of which continuation to pass to the body of a function is arguably the cleanest. Passing the static continuation s gives control the same static binding as ordinary λ-calculus variables. In the static case, the transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{\mathsf{s}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathsf{go}\, M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

We type our source language with `here` and `go` as in Figure 1.

In logical terms, both `here` and `go` are a combined right weakening and contraction. By themselves, weakening and contraction do not amount to much; but it is the combination with the rule for →-introduction that makes the calculus "classical", in the sense that there are terms whose types are propositions of classical, but not of intuitionistic, minimal logic.

To see how →-introduction gives classical types, consider λ-abstracting over `go`.

$$\frac{x : A \vdash_{\mathsf{s}} \mathsf{go}\, x : B, A}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\mathsf{go}\, x : A \to B, A}$$

If we read the comma as "or", and $A \to B$ for arbitrary B as "not A", then this judgement asserts the classical excluded middle, "not A or A". We build on the classical type of $\lambda_{\mathsf{s}} x.\mathsf{go}\, x$ for another canonical example: Scheme's `call-with-current-continuation` (`call/cc` for short) operator [4]. It is syntactic sugar in terms of static `here` and `go`:

$$\texttt{call/cc} = \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\mathsf{go}\,x))).$$

As one would expect [3], the type of `call/cc` is Peirce's law "if not A implies A, then A". We derive the judgement

$$\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\mathsf{go}\,x))) : ((A \to B) \to A) \to A, C$$

as follows. Let Γ be the context $f : (A \to B) \to A$. Then we derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{s}} f : (A \to B) \to A, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{s}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{s}} \mathsf{go}\,x : B, A}}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\mathsf{go}\,x : A \to B, A}}{\Gamma \vdash_{\mathsf{s}} (f\,(\lambda_{\mathsf{s}} x.\mathsf{go}\,x)) : A, A}}{\Gamma \vdash_{\mathsf{s}} \texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\mathsf{go}\,x)) : A, C}}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\mathsf{go}\,x))) : ((A \to B) \to A) \to A, C}$$

As another example, let Γ be any context, and assume we have $\Gamma \vdash_{\mathsf{s}} M : A, B$. Right exchange is derivable in that we can also derive $\Gamma \vdash_{\mathsf{s}} M' : B, A$ for some M'.

In the typing of `call/cc`, a `go` is (at least potentially, depending on f) exported from its enclosing `here`. Conversely, in the derivation of right exchange, a `go` is imported into a `here` from without. What makes everything work is static binding.

4 Dynamic control

Next we consider the dynamic version of `here` and `go`. The word "dynamic" is used here in the sense of dynamic binding and dynamic control in Lisp. Another way of phrasing it is that with a dynamic semantics, the `here` that is in scope at the point where a function is *called* will be used, as opposed to the `here` that was in scope at the point where the function was *defined*—the latter being used for the static semantics.

In the dynamic case, the transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{\mathsf{d}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\texttt{here}\,M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathsf{go}\,M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

In this transform, the jump continuation acts as a handler continuation; since it is passed as an extra argument on each call, the dynamically enclosing handler is chosen. Hence under the dynamic semantics, `here` and `go` become a stripped-down version of Lisp's `catch` and `throw` with only a single catch

Fig. 2. Typing for dynamic `here` and `go`

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{d}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \texttt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \texttt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{d}} M : B, C}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.M : A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} N : A, C}{\Gamma \vdash_{\mathsf{d}} MN : B, C}$$

tag. These `catch` and `throw` operation are themselves a no-frills version of exceptions with only identity handlers. We can think of `here` and `go` as a special case of these more elaborate constructs:

$$\texttt{here}\, M \equiv \texttt{(catch 'e }\; M\texttt{)}$$

$$\texttt{go}\, M \equiv \texttt{(throw 'e }\; M\texttt{)}$$

Because the additional continuation is administered dynamically, we cannot fit it into our simple typing without annotating the function type. So for dynamic control, we write the function type as $A \to B \vee C$. Syntactically, this should be read as a single operator with the three arguments in mixfix. We regard the type system as a variant of intuitionistic logic in which $\to$ and $\vee$ always have to be introduced or eliminated together.

This annotated arrow can be seen as an idealization of the Java `throws` clause in method definitions, in that $A \to B \vee C$ could be written as

$$B\texttt{(}A\texttt{) throws }C$$

in a more Java-like syntax. A function of type $A \to B \vee C$ may throw things of type C, so it may only be called inside a `here` with the same type. Our typing for the language with dynamic `here` and `go` is presented in Figure 2.

We do not attempt to idealize the ML way of typing exceptions because ML uses a universal type `exn` for exceptions, in effect allowing a carefully delimited area of untypedness into the language. The typing of ML exceptions is therefore much less informative than that of checked exceptions.

Note that `here` and `go` are still the same weakening and contraction hybrid as in the static setting. But here their significance is a completely different one because the $\to$-introduction is coupled with a sort of $\vee$-introduction. To see the difference, recall that in the static setting λ-abstracting over a `go` reifies the jump continuation and thereby, at the type level, gives rise to classical disjunction. This is not possible with the version of λ that gives `go` the dynamic semantics. Consider the following inference:

$$\frac{x : A \vdash_{\mathsf{d}} \texttt{go}\, x : B, A}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\texttt{go}\, x : A \to B \vee A, C}$$

The C-accepting continuation at the point of definition is not accessible to the `go` inside the λ_{d}. Instead, the `go` refers only to the A-accepting continuation that will be available at the point of call. Far from the excluded middle, the type of $\lambda_{\mathsf{d}} x.\mathtt{go}\, x$ is thus "A implies A or B; or anything".

In the same vein, as a further illustration how fundamentally different the dynamic `here` and `go` are from the static variety, we revisit the term that, in the static setting, gave rise to `call/cc` with its classical type:

$$\lambda f.\mathtt{here}\,(f\;(\lambda x.\mathtt{go}\,x)).$$

Now in the dynamic case, we can only derive the intuitionistic formula

$$((A \to B \vee A) \to A \vee A) \to A \vee C$$

as the type of this term.

Let Γ be the context $f : (A \to B \vee A) \to A \vee A$. Then we have:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{d}} f : (A \to B \vee A) \to A \vee A, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{d}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{d}} \mathtt{go}\, x : B, A}}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\mathtt{go}\, x : A \to B \vee A, A}}{\dfrac{\Gamma \vdash_{\mathsf{d}} (f\;(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, A}{\Gamma \vdash_{\mathsf{d}} \mathtt{here}\,(f\;(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, C}}}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} f.\mathtt{here}\,(f\;(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : ((A \to B \vee A) \to A \vee A) \to A \vee C, D}$$

5 Return continuation

Our last choice is passing the return continuation as the extra continuation to the body of a λ-abstraction. So the CPS transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{\mathsf{r}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\,M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\,M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

This transform grants λ_{r} the additional role of a continuation binder. The original operator for this purpose, `here`, is rendered redundant, since `here` M is now equivalent to $(\lambda_{\mathsf{r}} x.M)(\lambda_{\mathsf{r}} y.y)$ where x is not free in M. At first sight, binding continuations seems an unusual job for a λ; but it becomes less so if we think of `go` as the `return` statement of C or Java.

5.1 *Non-first class* `return`

Because the enclosing λ determines which continuation `go` jumps to with its argument, the `go`-operator has the same effect as a `return` statement. The

Fig. 3. Typing for go as a $\mathtt{return}$-operation

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{r}} x : A, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \mathsf{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{r}} N : A, C}{\Gamma \vdash_{\mathsf{r}} MN : B, C}$$

type of extra continuation assumed by go needs to agree with the return type of the nearest enclosing λ:

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C}$$

The whole type system for the calculus with λ_{r} is in Figure 3.

The agreement between go and the enclosing λ_{r} is comparable with the typing in C, where the expression featuring in a `return` statement must have the return type declared by the enclosing function. For instance, M needs to have type `int` in the definition:

$$\mathtt{int\ f()\{}\ldots\mathtt{return}\ M\mathtt{;}\ldots\mathtt{\}}$$

With λ_{r}, the special form go cannot be made into a first-class function. If we try to λ-abstract over $\mathsf{go}\, x$ by writing $\lambda_{\mathsf{r}} x.\mathsf{go}\, x$ then go will refer to that λ_{r}.

The failure of λ_{r} to give first-class returning can be seen logically as follows. In order for λ_{r} to be introduced, both types on the right have to be the same:

$$\frac{x : A \vdash_{\mathsf{r}} \mathsf{go}\, x : A, A}{\vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.\mathsf{go}\, x : A \to A, C}$$

Rather than the classical "not A or A" this asserts merely the intuitionistic "A implies A; or anything".

One has a similar situation in Gnu C, which has both the `return` statement and nested functions, without the ability to refer to the return address of another function. If we admit go as a first-class function, it becomes a much more powerful form of control, Landin's **JI**-operator.

5.2 The **JI**-operator

Keeping the meaning of λ_{r} as a continuation binder, we now consider a control operator **JI** that always refers to the statically enclosing λ_{r}, but which, unlike the special form go, is a first-class expression, so that we can pass the return continuation to some other function f by writing $f(\mathbf{JI})$. The CPS of this operator is this:

$$[\![\mathbf{JI}]\!] = \lambda ks.k(\lambda xrd.\boxed{s}x)$$

That is almost, but not quite, the same as if we tried to define **JI** as $\lambda_{\mathsf{r}} x.\mathsf{go}\, x$:

Fig. 4. Typing for **JI**

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{j}} x : A, C} \qquad \frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{j}} M : B, B}{\Gamma \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{j}} N : A, C}{\Gamma \vdash_{\mathsf{j}} MN : B, C}$$

$$\begin{aligned}
[\![\mathbf{JI}]\!] &= [\![\lambda_{\mathsf{r}} x.\mathtt{go}\, x]\!] \\
&= \lambda ks.k(\lambda xrd.\boxed{r}x)
\end{aligned}$$

We can, however, define **JI** in terms of `go` if we use the static λ_{s}, that is $\mathbf{JI} = \lambda_{\mathsf{s}} x.\mathtt{go}\, x$, as this does not inadvertently shadow the continuation s that we want **JI** to refer to.

The whole transform for the calculus with **JI** is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.qx \\
[\![\lambda_{\mathsf{r}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathbf{JI}]\!] &= \lambda ks.k(\lambda xrd.\boxed{s}x)
\end{aligned}$$

Recall that the role of `here` has been usurped by λ_{r}, and we replaced `go` by its first-class cousin **JI**.

In the transform for **JI**, the jump continuation is the current "dump" in the sense of the SECD-machine. The dump in the SECD-machine is a sort of call stack, which holds the return continuation for the procedure whose body is currently being evaluated. Making the dump into a first-class object was precisely how Landin invented first-class control, embodied by the **J**-operator.

The typing for the language with **JI** is given in Figure 4. In particular, the type of **JI** is the classical disjunction

$$\frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

As an example of the type system for the calculus with the **JI**-operator, we see that Reynolds's [9] definition of `call/cc` in terms of **JI** typechecks. (Strictly speaking, Reynolds used `escape`, the binding-form cousin of `call/cc`, but `call/cc` and `escape` are syntactic sugar for each other.) We infer the type of `call/cc` $\equiv \lambda_{\mathsf{r}} f.((\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI}))$ to be:

$$((A \to B) \to A) \to A)$$

To write the derivation, we abbreviate some contexts as follows:

$$\begin{aligned}
\Gamma_{fk} &\equiv f : (A \to B) \to A, k : (A \to B) \\
\Gamma_{f} &\equiv f : (A \to B) \to A
\end{aligned}$$

Then we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_{fk} \vdash_{\mathsf{j}} f : (A \to B) \to A, A} \qquad \dfrac{}{\Gamma_{fk} \vdash_{\mathsf{j}} k : (A \to B), A}}{\Gamma_{fk} \vdash_{\mathsf{j}} f\,k : A, A}}{\Gamma_f \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} k.fk : (A \to B) \to A, A} \qquad \dfrac{}{\Gamma_f \vdash_{\mathsf{j}} \mathbf{JI} : A \to B, A}}{\dfrac{\Gamma_f \vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} k.f\,k)(\mathbf{JI}) : A, A}{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.((\lambda_{\mathsf{r}} k.f\,k)(\mathbf{JI})) : ((A \to B) \to A) \to A), C}}$$

Because **JI** has such evident logical meaning as classical disjunction, we have considered it as basic. Landin [6] took another operator, called **J**, as primitive, while **JI** was derived as the special case of **J** applied to the identity combinator:

$$\mathbf{J}\,\mathbf{I} = \mathbf{J}\,(\lambda x.x)$$

This explains the name "**JI**", as "**J**" stands for "jump" and **I** for "identity". We were able to start with **JI**, since (as noted by Landin) the **J**-operator is syntactic sugar for **JI** by virtue of:

$$\mathbf{J} = (\lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx))\,(\mathbf{JI}).$$

To accommodate **J** in our typing, we use this definition in terms of **JI** to derive the following type for **J**:

$$\vdash_{\mathsf{j}} \mathbf{J} : (A \to B) \to (A \to C), B$$

Let Γ be the context $x : A, r : B \to C, f : A \to B$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{j}} r : B \to C, C} \qquad \dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{j}} f : A \to B, C} \qquad \dfrac{}{\Gamma \vdash_{\mathsf{j}} x : A, C}}{\Gamma \vdash_{\mathsf{j}} fx : B, C}}{\Gamma \vdash_{\mathsf{j}} r(fx) : C, C}}{r : B \to C, f : A \to B \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.r(fx) : A \to C, A \to C}}{r : B \to C \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (A \to B) \to (A \to C), (A \to B) \to (A \to C)}}{\dfrac{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (B \to C) \to (A \to B) \to (A \to C), B}{\vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx))\,(\mathbf{JI}) : (A \to B) \to (A \to C), B}}$$

This type reflects the behaviour of the **J**-operator in the SECD machine. When **J** is evaluated, it captures the B-accepting current dump continuation; it can then be applied to a function of type $A \to B$. This function is composed with the captured dump, yielding a non-returning function of type $A \to C$, for arbitrary C. By analogy with `call-with-current-continuation`, we may read the **J**-operator as "`compose-with-current-dump`" [13].

The logical significance, if any, of the extra function types in the general **J** seems unclear. There is a curious, though vague, resemblance to exception handlers in dynamic control, since they too are functions only to be applied on jumping. This feature of **J** may be historical, as it arose in a context where greater emphasis was given to attaching dumps to functions than to dumps as first-class continuations in their own right.

6 Type preservation

The typings agree with the transforms in that they are preserved in the usual way for CPS transforms: we have a "double-negation" transform for types, contexts and judgements. The only (slight) complication is in typing the dynamic continuation in those transforms that ignore it.

The function type of the form $A \to B \vee C$ for the dynamic semantics is translated as follows:

$$[\![A \to B \vee C]\!] = [\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to ([\![C]\!] \to \mathbf{Ans}) \to \mathbf{Ans}$$

Each call expects not only the B-accepting return continuation, but also the C-accepting continuation determined by the `here` that encloses the call.

Because we have not varied the transform of application, functions defined with λ_{s} and λ_{r} are also passed this dynamic continuation, even though they ignore it:

$$[\![\lambda_{\mathsf{s}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s})$$
$$[\![\lambda_{\mathsf{r}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r})$$

In both of these cases, the dynamic jump continuation d is fed to each function call, but never needed. Each function definition must expect this argument to be of certain type. Because different calls of the same function may have dynamically enclosing `here` operators with different types, the type ascribed to d should be polymorphic.

So the function type of the form $A \to B$ is transformed so as to accept this unwanted argument polymorphically:

$$[\![A \to B]\!] = \forall\beta.[\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to \beta \to \mathbf{Ans}$$

That is, a function of type $A \to B$ accepts an argument of type A, a B-accepting return continuation, and the continuation determined by the `here` dynamically enclosing the call.

For all the transforms we have preservation of the respective typing: if $\Gamma \vdash_? M : A, B$, then

$$[\![\Gamma]\!] \vdash [\![M]\!] : ([\![A]\!] \to \mathbf{Ans}) \to ([\![B]\!] \to \mathbf{Ans}) \to \mathbf{Ans}.$$

The proof is a straightforward induction over the derivation.

As a typical example, consider how the classical axiom of excluded middle

$$\vdash_{\mathsf{j}} \mathbf{JI} : A \to B, A$$

is translated to the λ-term $[\![\mathbf{JI}]\!] = \lambda ks.k(\lambda xrd.rx)$ with the type

$$((\forall\beta.[\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to \beta \to \mathbf{Ans}) \to \mathbf{Ans}) \to ([\![A]\!] \to \mathbf{Ans}) \to \mathbf{Ans}.$$

Fig. 5. Comparison of the type systems as logics

Static `here` and `go`, implies `call/cc`

$$\frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{s}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{s}} B, C}{\Gamma \vdash_{\mathsf{s}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} A \to B, C \quad \Gamma \vdash_{\mathsf{s}} A, C}{\Gamma \vdash_{\mathsf{s}} B, C}$$

Dynamic `here` and `go`, like checked exceptions

$$\frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{d}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} A, C}{\Gamma \vdash_{\mathsf{d}} B, C}$$

Non-first class `return`-operation

$$\frac{\Gamma \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{r}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} A \to B, C \quad \Gamma \vdash_{\mathsf{r}} A, C}{\Gamma \vdash_{\mathsf{r}} B, C}$$

Landin's **JI**-operator

$$\frac{}{\Gamma \vdash_{\mathsf{j}} B \to C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{j}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{j}} B, B}{\Gamma \vdash_{\mathsf{j}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} A \to B, C \quad \Gamma \vdash_{\mathsf{j}} A, C}{\Gamma \vdash_{\mathsf{j}} B, C}$$

7 Conclusions

As a summary of the four control constructs we have considered, we present their typings in Figure 5, omitting the terms for conciseness. As logical systems, these toy logics may seem a little eccentric, with two succedents that can only be manipulated in a slightly roundabout way. But they are sufficient for our purposes here, which is to illustrate the correspondence of first-class continuations with classical logic and weaker control operation with intuitionistic logic, and the central role of the arrow type in this dichotomy.

Recall the following fact from proof theory (see for example [15]). Suppose one starts from a presentation of intuitionistic logic with sequents of the form $\Gamma \vdash \Delta$. If a rule like the following is added that allows $\rightarrow$-introduction even if there are multiple succedents, the logic becomes classical.

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}$$

In continuation terms, the significance of this rule is that the function closure of type $A \rightarrow B$ may contain any of the continuations that appear in Δ; to use the jargon, these continuations become "reified". The fact that the logic becomes classical means that once we can have continuations in function closures, we gain first-class continuations and thereby the same power as `call/cc`. We have this form of rule for static `here` and `go`; though not for **JI**, since **JI** as the excluded middle is already blatantly classical by itself.

But the logic remains intuitionistic if the $\rightarrow$-introduction is restricted. The rule for this case typically admits only a single formula on the right:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B, \Delta}$$

Considered as a restriction on control operators, this rule prohibits λ-abstraction for terms that contain free continuation variables. There are clearly other possibilities how we can prevent assumptions from Δ to become hidden (in that they can be used in the derivation of $A \rightarrow B$ without showing up in this type itself). We could require these assumptions to remain explicit in the arrow type, by making Δ a singleton that either coincides with the B on the right of the arrow, or is added to it:

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \rightarrow B, C} \qquad \frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \rightarrow B \vee C, D}$$

These are the rules for $\rightarrow$-introduction in connection with the `return`-operation, and dynamic `here` and `go`, respectively. Neither of which gives rise to first-class continuations, corresponding to the fact that with these restrictions on $\rightarrow$-introduction the logics remain intuitionistic.

The distinction between static and dynamic control in logical terms appears to be new, as is the logical explanation of Landin's **JI**-operator.

7.1 Related work

Following Griffin [3], there has been a great deal of work on classical types for control operators, mainly on `call/cc` or minor variants thereof. A similar CPS transform for dynamic control (exceptions) has appeared in [5], albeit for a very different purpose. Felleisen describes the **J**-operator by way of CPS, but since his transform is not double-barrelled, **J** means something different in each λ [2]. Variants of the `here` and `go` operators are even older than the notion of continuation itself: the operations `valof` and `resultis` from CPL later appeared in Strachey and Wadsworth's report on continuations [11,12].

These operators led to the modern `return` in C. As we have shown here, they lead to much else besides if combined with different flavours of λ.

7.2 Further work

In this paper, control constructs were compared by CPS transforms and typing of the *source*. A different, but related approach compares them by typing in the *target* of the CPS [1]. On the source, we have the dichotomy between intuitionistic and classical typing, whereas on the target, the distinction is between linear and intuitionistic. We hope to relate these in further work.

References

[1] Berdine, J., P. W. O'Hearn, U. Reddy and H. Thielecke, *Linearly used continuations*, in: A. Sabry, editor, *Proceedings of the 3rd ACM SIGPLAN Workshop on Continuations*, 2001.

[2] Felleisen, M., *Reflections on Landin's J operator: a partly historical note.*, Computer Languages **12** (1987), pp. 197–207.

[3] Griffin, T. G., *A formulae-as-types notion of control*, in: *Proc. 17th ACM Symposium on Principles of Programming Languages*, San Francisco, CA USA, 1990, pp. 47–58.

[4] Kelsey, R., W. Clinger and J. Rees, editors, *Revised^5 report on the algorithmic language Scheme*, Higher-Order and Symbolic Computation **11** (1998), pp. 7–105.

[5] Kim, J., K. Yi and O. Danvy, *Assessing the overhead of ML exceptions by selective CPS transformation*, in: *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

[6] Landin, P. J., *A generalization of jumps and labels*, Report, UNIVAC Systems Programming Research (1965).

[7] Landin, P. J., *A generalization of jumps and labels*, Higher-Order and Symbolic Computation **11** (1998), reprint of [6].

[8] Plotkin, G., *Call-by-name, call-by-value, and the λ-calculus*, Theoretical Computer Science **1** (1975), pp. 125–159.

[9] Reynolds, J. C., *Definitional interpreters for higher-order programming languages*, in: *Proceedings of the 25th ACM National Conference* (1972), pp. 717–740.

[10] Riecke, J. G. and H. Thielecke, *Typed exceptions and continuations cannot macro-express each other*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS **1644** (1999), pp. 635–644.

[11] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps*, Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK (1974).

[12] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps*, Higher-Order and Symbolic Computation **13** (2000), pp. 135–152, reprint of [11].

[13] Thielecke, H., *An introduction to Landin's "A generalization of jumps and labels"*, Higher-Order and Symbolic Computation **11** (1998), pp. 117–124.

[14] Thielecke, H., *On exceptions versus continuations in the presence of state*, in: G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000,*, number 1782 in LNCS (2000), pp. 397–411.

[15] Troelstra, A. S. and H. Schwichtenberg, "Basic Proof Theory," Cambridge University Press, 1996.