Program Logics for Sequential Higher-Order Control

Martin Berger

Department of Informatics, University of Sussex

Abstract. We introduce a Hoare logic for call-by-value higher-order functional languages with control operators such as `callcc`. The key idea is to build the assertion language and proof rules around an explicit logical representation of jumps and their dual 'places-to-jump-to'. This enables the assertion language to capture precisely the intensional and extensional effects of jumping by internalising rely/guarantee reasoning, leading to simple proof rules for higher-order functions with `callcc`. We show that the logic can reason easily about non-trivial uses of `callcc`. The logic matches exactly with the operational semantics of the target language (observational completeness), is relatively complete in Cook's sense and allows efficient generation of characteristic formulae.

1 Introduction

Non-trivial control manipulation is an important part of advanced programming and shows up in many variants such as jumps, exceptions and continuations. Research on axiomatic accounts of control manipulation starts with [10], where a simple, imperative first-order low-level language with `goto` is investigated. Recently, this research tradition was revived by a sequence of works on similar languages [2–4, 7, 24, 29, 32, 34]. None of those investigates the interplay between advanced control constructs and higher-order features. The present paper fills this gap and proposes a logic for ML-like call-by-value functional languages with advanced control operators (`callcc`, `throw`). The key difficulty in axiomatising higher-order control constructs for functional languages (henceforth "higher-order control") is that program logics are traditionally based on the idea of abstracting behaviour in terms of input/output relations. This is a powerful abstraction for simple languages but does not cater well for jumping, a rather more intensional form of behaviour. Consider the well-known program $\texttt{argfc} \stackrel{def}{=} \texttt{callcc}\ \lambda k.(\texttt{throw}\ k\ \lambda x.(\texttt{throw}\ k\ \lambda y.x))$ [12]. This function normalises to a λ-abstraction, but, as [28] investigates, distinguishes programs by application that are indistinguishable in the absence of continuations: $(\lambda x.(x\ 1);(x\ 2))\ \texttt{argfc} = 1$ and $(\lambda x.\lambda y.(x\ 1);(y\ 2))\ \texttt{argfc}\ \texttt{argfc} = 2$ with $M;N$ being the sequential composition of M and N, binding more tightly than λ-abstraction. The reason is that continuations carry information about contexts that may be returned (jumped) to later. Thus, values in languages with higher-order control are no longer simple entities, precluding logics based on input/output relations. Two ways of dealing with the intensional nature of control manipulation suggest themselves:

– Using continuation-passing style (CPS) transforms [33] to translate away control manipulating operators and then reason about transformed programs in logics like [16] for functional languages.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 194–211, 2010.

– Using a direct syntactic representation of intensional features.

We choose the second option for pragmatic reasons: It is difficult to reconstruct a program's specification from the specification of its CPS transform. This is because CPS transforms increase the size of programs, even where higher-order control is not used. This increases reasoning complexity considerably. In contrast, in our approach programs or program parts that do not feature higher-order control can be reasoned about in simpler logics for conventional functional languages. The more heavyweight logical apparatus for higher-order control is required only where control is manipulated explicitly, leading to more concise proofs and specifications.

Key Elements of the Present Approach. This work makes three key proposals for a logical treatment of higher-order control.

– *Names* as an explicit representation of places to jump to, or being jumped to.
– *Jumps* $\overline{x}\langle\tilde{e}\rangle A$ as an explicit logical operator which says that a program jumps to x carrying a vector $\tilde{e}$ of values, and after jumping, A holds. Jumps are complementary to the evaluation formulae $x\langle\tilde{e}\rangle A$, studied in [5, 16, 18, 36], which means a jump to x carrying values $\tilde{e}$ leads to a program state where A holds.
– *Rely/guarantee formulae* $\{A\}B$ and *tensor* $A \circ B$. $\{A\}B$ says that if the environment is as specified by A, then the program together with the environment will act as constrained by B. Likewise, $A \circ B$ says a program has a part that is as described by A, and a part that is as given by B. Rely/guarantee formulae generalise implication, and tensor generalise conjunction because in e.g. $A \wedge (B \supset C)$ a free variable must have the same type in A as in B and C. With rely/guarantee formulae, we weaken this requirement: e.g. in $\overline{x}\langle 2u\rangle \wedge \{x\langle vw\rangle\overline{w}\langle v+1\rangle\}\overline{u}\langle 3\rangle$ the variable x is used to output in the left conjunct and for input in the right conjunct, with the input occurring in the rely part of the rely/guarantee formula. The left conjunct says that the program jumps to x carrying 2 and u (an intensional specification at x). The right conjunct says that if the environment offers a function to be invoked at x that computes the successor of its first argument and returns the result at the second, then the jump to u carrying 3 will happen, a more extensional specification in that the program together with the assumed environment behaves as a function. Similarly, $\overline{x}\langle 3\rangle \circ x\langle 3\rangle A$ uses x with different polarities, specifying a program that contains a jump to x carrying 3 to a target for this jump.

Informal Explanation. Operationally, a program, for example the constant 5, can be seen as a value-passing jump carrying 5 to some distinguished name, called *default port*, left implicit in the language, but made explicit in implementations, usually as a return address on the stack. It can be left implicit in the absence of higher-order control because there are no alternatives for returning: every function, if it returns at all, does so at the default port. Higher-order control breaks this simplicity: for example `throw` k 5 will jump to k and not to the default port. Our logic deals with multiple return points by *naming* the default port in judgements, giving rise to the following shape of judgements (for total and partial correctness):

$$M :_{\overline{u}} A$$

It asserts that the program M satisfies the formula A, assuming that M's default port is named u (we do not need preconditions because they can be simulated using rely/guarantee formulae, see below). Using explicit jumps we can specify:

$$5 :_{\overline{u}} \overline{u}\langle 5\rangle \qquad\qquad \texttt{throw}\, k\, 5 :_{\overline{u}} \overline{k}\langle 5\rangle.$$

The left-hand side says that the program 5 terminates and jumps to the default port u carrying 5 as a value. The assertion on the right expresses that `throw` k 5 also terminates with a jump carrying 5, but now the jump is to k, which is not the default port.

Evaluation formulae are used to specify the behaviour of functions. When a function like $\lambda x.x+1$ is invoked, the result is returned at the default port of the invocation. As functions can be invoked more than once and in different contexts (in the present context, invoking a function f is the same as jumping to f, and we use both phrases interchangeably), different default ports are needed for different invocations. In implementations, a dynamically determined place on the stack is used for this purpose. In addition, when a λ-abstraction like $\lambda x.x+1$ is evaluated, the λ-abstraction itself, i.e. $\lambda x.x+1$, is returned at its default port. To express such behaviour we use the following specification (writing $\overline{u}(a)A$ for $\exists a.\overline{u}\langle a\rangle A$, and $a(xm)A$ for $\forall xm.a\langle xm\rangle A$).

$$\lambda x.x+1 :_{\overline{u}} \overline{u}(a)a(xm)\overline{m}\langle x+1\rangle$$

This judgement states that the abstraction returns a name a at the default port. This name can be jumped *to* (i.e. invoked) with two arguments, a number x and a name m, the default port for invocations of a. If invoked, the successor of x will be returned at m.

The role of rely/guarantee formulae is to generalise and internalise preconditions. Consider the application g 3. If jumps to g with two arguments, a number x and a return port u, yield a jump $\overline{u}\langle x+x\rangle$ then the evaluation of g 3 with default port u should induce a jump $\overline{u}\langle 6\rangle$. In a program logic with preconditions, we would expect to be able to derive $\{g(xm)\overline{m}\langle x+x\rangle\}\ g\ 3\ :_{\overline{u}} \{\overline{u}\langle 6\rangle\}$. With rely/guarantee formulae we can express this by defining

$$\{A\}\ M\ :_{\overline{m}} \{B\} \qquad \stackrel{def}{=} \qquad M :_{\overline{m}} \{A\}B.$$

The advantage of internalising preconditions with rely/guarantee formulae are threefold. (1) Key structural relationships between jumps and evaluation formulae are easily expressible as axioms like: $\overline{x}\langle\tilde{e}\rangle \supset \{x\langle\tilde{e}\rangle A\}A$. It states that e.g. a jump $\overline{g}\langle 3u\rangle$ makes A true whenever the environment guarantees that jumps *to* g with arguments 3 and u will validate A. (2) We gain more flexibility in localising and manipulating assumptions, leading to more succinct and compositional reasoning. To see why, consider a complicated formula $C[\overline{x}\langle 2\rangle]$ containing a jump to x. Using the axiom just given, and setting $A \stackrel{def}{=} x\langle 2\rangle\overline{u}\langle 3\rangle$, we know that $\overline{x}\langle 2\rangle \supset \{A\}\overline{u}\langle 3\rangle$, hence $C[\overline{x}\langle 2\rangle]$ implies $C[\{A\}\overline{u}\langle 3\rangle]$. Such reasoning is cumbersome if all assumptions have to be concentrated in the precondition. Moreover, local hypotheses can be collected, i.e. we can usually infer from $C[\{A\}\overline{u}\langle 3\rangle]$ to $\{A\}C[\overline{u}\langle 3\rangle]$, hence conventional reasoning based on rigid pre-/postconditions remains valid unmodified without additional cost (all relevant rules and axioms are derivable). The added fluidity in manipulating assumptions is vital for reasoning about involved forms of mutually recursive jumping. (3) Finally, the most important virtue of internalising preconditions is sheer expressive power. With rely/guarantee formulae, it easy

to use different assumptions in a single formula: consider $A \stackrel{def}{=} g(xm)\overline{m}\langle x+x\rangle$ and $B \stackrel{def}{=} g(xm)\overline{m}\langle x\cdot x\rangle$. We can now specify $g\ 3 :_{\overline{u}} (\{A\}\overline{u}\langle 6\rangle) \wedge \{B\}\overline{u}\langle 9\rangle$. This expressiveness enables convenient reasoning about complicated behavioural properties of programs that would be difficult to carry out otherwise.

Contributions. The present work provides the first general assertion method with compositional proof rules for higher-order functions with functional control (`callcc` and similar operators) and recursion under the full type hierarchy. The work identifies as key ingredients in this approach: (1) An explicit representation of jumps in formulae, which can specify intensional aspects of control operators in a uniform manner. (2) Rely/guarantee formulae and an associated tensor to facilitate local specification of extensional as well as intensional aspects of higher-order control, and to enable complicated forms of reasoning not otherwise possible. (3) Proof rules and axioms that capture the semantics of PCF^+precisely, as demonstrated by strong completeness results. Missing proofs can be found in the full version of this paper.

2 PCF with Jumps

Now we define our programming language. We extend PCF with `callcc` and `throw`, and call the resulting language PCF^+. Arguments are evaluated using *call-by-value* (CBV). Later we briefly consider μPCF, a variant of CBV PCF with different control operators. The relationship between both is explained in [21]. *Types*, *terms* and *values* are given by the grammar below. Sums, products and recursive types for PCF^+ are straightforward and are discussed in the full version of this paper.

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid \alpha \to \beta \mid (\alpha)^? \qquad V ::= x \mid \mathtt{c} \mid \lambda x^\alpha.M \mid \mathtt{rec}\, f^\alpha.\lambda x^\beta.M$$
$$M ::= V \mid MN \mid \mathsf{op}(\tilde{M}) \mid \mathtt{if}\, M\, \mathtt{then}\, N\, \mathtt{else}\, N' \mid \mathtt{callcc} \mid \mathtt{throw}$$

Here $(\alpha)^?$ corresponds to $(\alpha\ \mathsf{cont})$ in SML and is the type of continuations with final answer type α, $\mathtt{c}$ ranges over constants like $0,1,2,\ldots$, op over functions like addition. We write e.g. $ab3$ for the vector $\langle a,b,3\rangle$, $\tilde{M}$ for the vector $\langle M_0,\ldots,M_{n-1}\rangle$, etc; $x,f,\ldots$ range over *variables*. Names are variables that can be used for jumping. The notions of *free variables* $\mathsf{fv}(M)$ and *bound variables* $\mathsf{bv}(M)$ of M are defined as usual. Typing judgements $\Gamma \vdash M : \alpha$ are standard, with Γ being a finite, partial map from variables to the types α. *From now on we assume all occurring programs to be well-typed.* The semantics of PCF^+ is straightforward, cf. [28].

3 The Logic

This section defines the syntax and semantics of the logic. Since variables in programs are typed, and the logic speaks about such variables, our logic is typed, too. Types are those of PCF^+, with two generalisations. (1) We add a type $(\tilde{\alpha})^!$ which is the type *being-jumped-to* with arguments typed by the vector $\tilde{\alpha}$. (2) We no longer need function spaces, because e.g. $\alpha \stackrel{def}{=} \mathbb{N} \to \mathbb{B}$ can now be decomposed into $\alpha^\circ \stackrel{def}{=} (\mathbb{N}(\mathbb{B})^?)^!$. Type α° holds of names that can be *jumped to* with two arguments, first a number and then another name, which might be used for subsequent jumps carrying a boolean. This is

the behaviour of functions $\mathbb{N} \to \mathbb{B}$ under call-by-value evaluation. If we denote by $\overline{\alpha}$ the result of changing all occurring ? in α into ! and vice versa, and if we denote by α° the result of translating PCF^+ types as just described, then:

$$(\alpha \to \beta)^\circ = (\overline{\alpha^\circ}(\beta^\circ)^?)^!.$$

Our types are given by the grammar:

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid (\alpha)^? \mid (\alpha\beta)^? \mid (\alpha)^! \mid (\alpha\beta)^!$$

Types play essentially the same role in our logic as they do in programming languages, namely to prevent terms that do not make sense, like $x = 5 + \mathsf{t}$ or $\overline{x}\langle 3\rangle \circ x\langle\rangle A$. Since our use of types is straightforward, the reader can mostly ignore types in the remainder of the text, as long as he or she bears in mind that *all occurring formulae and judgements must be well-typed*. (Further information about this typing system is given in [15].)

Expressions, Formulae, Assertions. Expressions are standard ($e ::= x \mid \mathsf{c} \mid \mathsf{op}(\tilde{e})$) and formulae for our logic are generated by the following grammar.

$$A ::= e = e' \mid A \wedge B \mid \neg A \mid \forall x^\alpha.A \mid x\langle\tilde{e}\rangle A \mid \overline{x}\langle\tilde{e}\rangle A \mid \{A\}B \mid A \circ B$$

Variables, constants and functions are those of §2. Standard logical operators such as T implication and existential quantification are defined as usual. We often omit type annotations. Logical operators have the usual rules of precedence, e.g. $\forall x.A \wedge B$ should be read as $\forall x.(A \wedge B)$, $A \circ B \wedge C$ as $(A \circ B) \wedge C$, and $\{A\}B \wedge C$ is short for $(\{A\}B) \wedge C$. We write $\mathsf{fv}(A)$ for A's free variables, and A^{-x} indicates that $x \notin \mathsf{fv}(A)$. Names are also variables. Typing environments, $\Gamma, \Delta, \ldots$ are defined as finite maps from names to types. Typing judgements for expressions $\Delta \vdash e : \alpha$ and formulae $\Delta \vdash A$ are defined as usual, e.g. x must be of type $\mathbb{N}$ in $x + 3 = 2$. The new operators are typed as follows.

- $\Gamma \vdash \overline{x}\langle\tilde{e}\rangle A$ if $\Gamma \vdash x : (\tilde{\alpha})^?$, $\Gamma \vdash e_i : \beta_i$, $\beta_i \in \{\alpha_i, \overline{\alpha}_i\}$ and $\Gamma \vdash A$.
- $\Gamma \vdash x\langle\tilde{e}\rangle A$ if $\Gamma \vdash x : (\tilde{\alpha})^!$, $\Gamma \vdash e_i : \beta_i$, $\beta_i \in \{\alpha_i, \overline{\alpha}_i\}$ and $\Gamma \vdash A$.
- For rely/guarantee formulae $\Gamma \vdash \{A\}B$ we say x is *compensated* in A if the type of x in A is dual to that in Γ. For example $\overline{x}\langle 2y\rangle \supset \{x\langle 2y\rangle\overline{y}\langle 3\rangle\}B$ is typable under $\Delta \stackrel{def}{=} x : (\mathbb{N}(\mathbb{B})^!)^?, y : (\mathbb{B})^?$.

We write e.g. $\overline{x}\langle y\cdot\rangle A$ to stand for any $\overline{x}\langle yz\rangle A$ such that z is fresh and does not occur in A, and likewise for evaluation formulae. We write $\overline{x}\langle\tilde{e}(y)\rangle A$ for $\exists y.\overline{x}\langle\tilde{e}y\rangle A$, assuming y not to occur in $\tilde{e}$. *Judgements*, also called *assertions*, are of the form $M :_{\overline{m}} A$. Judgements must be well-typed, i.e. M and A must be well-typed and the variables common to A and M must be given consistent types, e.g. $g\ 3 :_{\overline{u}} \{g\langle 4u\rangle\mathsf{T}\}2 = 3$ is well-typed, but $g\ 3 :_{\overline{u}} \{g\langle z\rangle\mathsf{T}\}2 = 3$ is not.

Examples of Assertions. We continue with simple examples of assertions.

- Let $A \stackrel{def}{=} g(xk)(\mathsf{even}(x) \supset \overline{k}(a)\mathsf{even}(a))$. This first example specifies a place g to jump to. If a jump to g happens carrying an even number x as first argument and k, the default port, then that invocation at g will return at its default port, carrying another even number. A does not specifying anything if x is odd.

- Next consider the following formulae. $A \stackrel{def}{=} x(kr)(\overline{k}\langle 7\rangle \vee \overline{r}\langle 8\rangle)$ and $B \stackrel{def}{=} \{A\}\overline{u}(m)\,(m = 7 \vee m = 8)$ A specifies a place x to jump to with two arguments, k and r (the default port), both of which are used for jumping: either jumping to k carrying 7, or jumping to the default port carrying 8. B specifies a jump to u carrying 7 or 8, provided the environment provides a place to jump to at x, as just described by A.
- Now consider the formula $A \stackrel{def}{=} x(ab)\overline{a}\langle bb\rangle$. It says that if we jump to x carrying two arguments, a and b, both being used for jumping, then the invocation at x replies with a jump to a, carrying b twice. Figure 2 shows that $\overline{u}(x)A$ specifies the behaviour of `callcc`, assuming u as default port.
- Finally, consider the following formula. $A \stackrel{def}{=} \overline{n}(b)b(xy)\overline{n}(c)c(zr)\overline{r}\langle x\rangle$. The formula A specifies a jump to n, carrying a function b that can be jumped to with two arguments, x and y. Of those, y is subsequently ignored. If b is invoked, it jumps to n again, carrying another function c, which also takes two arguments, z and r. Of these z is also ignored, but r is jumped to immediately, carrying x. It can be shown that A specifies `argfc`.

Models and the Satisfaction Relation. This section sketches key facts about the semantics of our logic and states soundness and completeness results. We use a typed π-calculus to construct our semantics. This choice simplifies models and reasoning about models for the following reasons.

- Models and the satisfaction relation need to be built only once and then cater for many different languages with functional control like PCF and μPCF. Thus soundness of axioms needs to be proven only once. Proving soundness and completeness is also simpler with π-calculus based models because powerful reasoning tools are available, e.g. labelled transitions, that languages with higher-order sequential control currently lack.
- Using processes, the semantics is simple, intuitive and understandable, as well as capturing behaviour of higher-order control precisely. The typed processes that interpret PCF^+ or μPCF-programs are up to bisimilarity exactly the morphisms (strategies) in the control categories that give fully abstract models to PCF^+ or μPCF[13, 21]. Hence the present choice of model gives a direct link with game-based analysis of control.

Processes. The grammar below defines processes with expressions e as above, cf. [17] for details.

$$P ::= 0 \mid \overline{x}\langle\tilde{e}\rangle \mid !x(\tilde{v}).P \mid (\nu x)P \mid P|Q \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$$

We can use this calculus to give fully abstract encodings of PCF^+ and μPCF [17, 21]. Translation is straightforward and we show some key cases.

$$[\![\lambda x.M]\!]_u \stackrel{def}{=} \overline{u}(a)!a(xm).[\![M]\!]_m \qquad [\![\texttt{throw}]\!]_u \stackrel{def}{=} \overline{u}(a)!a(xm)\overline{m}(b)!b(y\cdot)\overline{x}\langle y\rangle$$
$$[\![MN]\!]_u \stackrel{def}{=} (\nu m)([\![M]\!]_m|!m(a).(\nu n)([\![N]\!]_n|!n(b).\overline{a}\langle bu\rangle)) \qquad [\![\texttt{callcc}]\!]_u \stackrel{def}{=} \overline{u}(a)!a(xm).\overline{x}\langle mm\rangle$$

This translation generalises a well-known CPS transform [33]. All cases of the translation are syntactically essentially identical with the corresponding logical rules. This simplifies soundness and completeness proofs and was a vital rule-discovery heuristic.

The Model and Satisfaction Relations. *Models of type* Γ are of the form (P,ξ) where P is a process and ξ maps values names and variables to their denotation. We write $\models M :_{\overline{m}} A$ if for all appropriately typed-models (P,ξ) with m fresh in ξ we have

$$([\![M]\!]_m \xi | P, \xi) \models A$$

This satisfaction relation works for total and partial correctness, since termination can be stated explicitly through jumps in total correctness judgements. On formulae, the satisfaction relation is standard except in the following four cases, simplified to streamline the presentation (here $\cong$ is the contextual congruence on typed processes).

- $(P,\xi) \models \overline{x}\langle y\rangle$ if $P \cong Q|\overline{a}\langle b\rangle$, $\xi(x) = a, \xi(y) = b$.
- $(P,\xi) \models x\langle y\rangle A$ if $P \cong Q|!a(v).R$ with $\xi(x) = a$ and $(P|\overline{a}\langle \xi(y)\rangle, \xi) \models A$.
- $(P,\xi) \models \{A\}B$ if for all Q of appropriate type $(Q,\xi) \models A$ implies $(P|Q,\xi) \models B$.
- $(P,\xi) \models A \circ B$ if we can find Q,R such that $P \cong Q|R$, $(Q,\xi) \models A$ and $(R,\xi) \models B$.

The construction shows that rely/guarantee formulae correspond to (hypothetical) parallel composition [20].

4 Axioms and Rules

This section introduces all rules and some key axioms of the logic. We start with the latter and concentrate on axioms for jumps, tensor and rely/guarantee formulae. Some axioms correspond closely to similar axioms for implication and conjunction. *All axioms and rules are included in the logic exactly when they are typable.*

Axioms for Dynamics. We start with the two axioms that embody the computational dynamics of jumping. The first expresses the tight relationship between jumping and being-jumped-to (evaluation formulae):

$$\overline{u}\langle \tilde{e}\rangle A \circ u\langle \tilde{e}\rangle B \qquad \supset \qquad A \circ B \tag{CUT}$$

[CUT] says that if a system is ready to make a jump to u, say it satisfies $\overline{u}\langle \tilde{e}\rangle A$, and if the system also contains the target for jumps to u, i.e. it satisfies $u\langle \tilde{e}\rangle B$, then that jump will happen, and $A \circ B$ will also be true of the system.

The next axiom says that a jump $\overline{x}\langle \tilde{e}\rangle A$ which guarantees A implies the weaker statement that if the environment can be jumped to at x with arguments $\tilde{e}$, then B holds, provided the environment can rely on A in its environment.

$$\overline{x}\langle \tilde{e}\rangle A \quad \supset \quad \{x\langle \tilde{e}\rangle \{A\}B\}B \tag{XCHANGE}$$

Further Axioms for Tensor and Rely/Guarantee Formulae. Now we present some axioms for the tensor that show its close relationship with conjunction. In parallel, we also exhibit axioms for rely/guarantee formulae that relate them with implication. As

before, we assume that both sides of an entailment or equivalence are typed under the same typing environment. This assumption is vital for soundness, as we illustrate below.

$$\begin{array}{ccc} A \circ B \equiv A \wedge B & A \circ B \supset A & A \supset \{B\}A \\ A \circ (B \circ C) \equiv (A \circ B) \circ A & (\forall x.A) \circ B^{\bullet x} \equiv \forall x.(A \circ B) & \{A\}\{B\}C \equiv \{A \circ B\}C \\ A \circ B \equiv B \circ A & \{A\}B \equiv A \supset B & B \circ \{B\}A \supset A \end{array}$$

Our explanation of these axioms starts on the left. The first axiom says that if $A \wedge B$ are typable then tensor is just conjunction. This does not imply that $\overline{x}\langle 3\rangle \circ x\langle 3\rangle A$ is equivalent to $\overline{x}\langle 3\rangle \wedge x\langle 3\rangle A$, since $\overline{x}\langle 3\rangle \wedge x\langle 3\rangle A$ is not typable. However $(x = 3 \circ y = 1) \equiv (x = 3 \wedge y = 1)$ is valid. The next two axioms below state associativity and commutativity of tensor. The top axiom in the middle shows that tensor is not like parallel composition, because the tensor can "forget" their component formulae. The axiom below shows that tensor associates as expected with quantification. The bottom axiom in the middle shows that rely/guarantee formulae reduce to implication if all free variables have the same type in A as in B, i.e. $(\{x\langle\tilde{e}\rangle\overline{a}\}\overline{x}\langle e\rangle) \equiv ((x\langle\tilde{e}\rangle\overline{a}) \supset \overline{x}\langle e\rangle)$ is not a valid instance of the axiom, but $(\{x\langle\tilde{e}\rangle\overline{a}\}x\langle\tilde{e}\rangle\overline{b}) \equiv ((x\langle\tilde{e}\rangle\overline{a}) \supset x\langle\tilde{e}\rangle\overline{b})$ is. The top right axiom shows that it is possible to weaken with a rely formula. The middle axiom on the right shows how to merge two assumptions in rely/guarantee formulae. The bottom right axiom can be seen as a typed form of Modus Ponens, and we call it [MP]. The expected forms of weakening also hold, i.e. if $A \supset A'$ then $A \circ B$ implies $A' \circ B$, $\{A'\}B$ implies $\{A\}B$ and $\{B\}A$ implies $\{B\}A'$.

Further Axioms for Jumps and Evaluation Formulae. Before moving on to rules, we present some axioms for jumps and evaluation formulae.

$$\begin{array}{cc} x\langle\tilde{e}\rangle(A \wedge y\langle\tilde{g}\rangle B) \equiv y\langle\tilde{g}\rangle(B \wedge x\langle\tilde{e}\rangle A) & x\langle\tilde{e}\rangle\mathsf{T} \equiv \mathsf{T} \\ A \circ (\overline{x}\langle\tilde{e}\rangle B) \equiv \overline{x}\langle\tilde{e}\rangle(A \circ B) & \overline{x}\langle\tilde{e}\rangle \wedge \overline{y}\langle\tilde{g}\rangle \supset (x = y \wedge \tilde{e} = \tilde{g}) \end{array}$$

The top left axiom states that free variables like x and y that can be jumped to, are 'always there', i.e. they cannot come into existence only after some function has been invoked. The top right axiom says that places to jump to cannot 'refuse' arguments: in other words, the statement $x\langle e\rangle\mathsf{T}$ carries no information. This axiom is called [NOINFO]. The bottom left axiom says that if a program contains a part that jumps at x then the program as a whole can also jump at x, provided that the program does not contain a component that offers an input at x (not offering an input at x is implicit in typability of the axiom). Finally, the last axiom expresses that our language is sequential: at most one jump can happen at any time.

Rules for PCF$^+$. The total correctness rules for PCF$^+$ are given in Figure 1. Rules are subject to straightforward well-formedness conditions. *From now on we assume all rules to be well-typed.* We explain the rules in some detail. As [VAR, CONST, ABS] have already been sketched in the introduction, we start with the rule for application. The purpose of [APP], the rule for function application, is to ensure the coordination of functions and their invocations by jumps. One issue is the generation and management of default ports: the present approach requires that a (terminating) function application *may* return its result at the application's default port, assuming the evaluations of the

$$\frac{M :_{\overline{m}} A}{\lambda x.M :_{\overline{u}} \overline{u}(a)a(xm)A}\ \textsc{Abs} \qquad \frac{\lambda x.M :_{\overline{u}} \overline{u}(a)A}{\mathtt{rec}\ g.\lambda x.M :_{\overline{u}} \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ A)}\ \textsc{Rec} \qquad \frac{-}{\mathtt{c} :_{\overline{u}} \overline{u}\langle \mathsf{c}\rangle}\ \textsc{Const}$$

$$\frac{M :_{\overline{m}} A \quad N :_{\overline{n}} B}{MN :_{\overline{u}} \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{a}\langle bu\rangle))}\ \textsc{App} \qquad \frac{-}{\mathtt{callcc} :_{\overline{u}} \overline{u}(a)a(xm)\overline{x}\langle mm\rangle}\ \textsc{Ccc} \qquad \frac{-}{x :_{\overline{u}} \overline{u}\langle x\rangle}\ \textsc{Var}$$

$$\frac{-}{\mathtt{throw} :_{\overline{u}} \overline{u}(a)a(xm)\overline{m}(b)b(y\cdot)\overline{x}\langle y\rangle}\ \textsc{Throw} \qquad \frac{M :_{\overline{m}} A \quad N :_{\overline{n}} B}{M+N :_{\overline{u}} \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{u}\langle a+b\rangle))}\ \textsc{Add}$$

$$\frac{M :_{\overline{m}} A \quad N :_{\overline{u}} B \quad N' :_{\overline{u}} C}{\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' :_{\overline{u}} \exists m.(A \circ m(a)((a = \mathtt{t} \supset B) \wedge (a = \mathtt{f} \supset C)))}\ \textsc{If} \qquad \frac{M :_{\overline{u}} A \quad A \supset B}{M :_{\overline{u}} B}\ \textsc{Cons}$$

Fig. 1. Total Correctness rules for PCF^+. The *forwarder* is given by $\mathsf{fw}_{xy} \stackrel{def}{=} x(\tilde{v})\overline{y}\langle\tilde{v}\rangle$.

function itself, and that of the argument return their respective results at (distinct) default ports themselves. [App] achieves this by explicitly representing the sequence of jumps that are integral parts of evaluating a function application. First the jump to the default port of the function is received by an evaluation formula at m. It receives an argument a. Then the evaluation of the argument is triggered, and its result, should it return at the fresh default port n, is received by a second evaluation formula at n. Finally, should both, the function and its argument return at their respective default ports, a jump to a carrying b and the application's default port u is executed. By typing we know that the jump to a must find an evaluation formula expecting two arguments.

Why do we have to represent the internals of application evaluation in the logic explicitly, rather then have them implicit as in the simpler logics for PCF [16]? After all, even in PCF, these jumps take place, albeit behind the scenes. The answer is that because of continuations, functions can return more than once, i.e. can jump to their default port more than once. The function `argfc` from the introduction is an example of such behaviour. The axiomatisation of PCF in [16] hides default ports, because programs cannot return anywhere but at default ports. It might not be possible to give a logical account of returning to a port more than once without explicit representation of default ports.

Representing jumps and default ports in a single formula, as we do in [App], has ramifications for typing: when names (like m,n above) are used in a formula for both, jumping, and for being-jumped-to we need to mediate, in a controlled way, the rigidity of typing, that enforces all names to be used under the same typing. Our rules use tensor for this purpose. All rules can be stated without tensors using just rely/guarantee formulae, but, it seems, not without a making the inference system more complicated.

Using [App], setting $A \stackrel{def}{=} \exists m.((\overline{m}(a)a(xu)\overline{u}\langle x+1\rangle) \circ m(a)\exists n.(\overline{n}\langle 7\rangle \circ n(b)\overline{a}\langle bu\rangle))$ and assuming that $\lambda x.x+1 :_{\overline{m}} \overline{m}(a)a(xr)\overline{r}\langle x+1\rangle$, we infer:

$$\frac{1 \quad \lambda x.x+1 :_{\overline{m}} \overline{m}(a)a(xr)\overline{r}\langle x+1\rangle}{2 \quad 7 :_{\overline{n}} \overline{n}\langle 7\rangle \textsc{Const}}$$

$$3 \quad (\lambda x.x+1)7 :_{\overline{u}} A \textsc{App}, 1, 2$$

The expected judgement $(\lambda x.x+1)7 :_{\overline{u}} \overline{u}\langle 8\rangle$, is by [CONS] and the following implication :

$$\begin{array}{lll} A & \supset & \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\exists n.(\overline{n}\langle 7\rangle\circ n(b)\overline{a}\langle bu\rangle)) \supset \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\exists n.\overline{a}\langle 7u\rangle) \\ & \supset & \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\overline{a}\langle 7u\rangle) \qquad\qquad\qquad\qquad\quad \supset \overline{u}\langle 8\rangle \end{array}$$

This implication follows from [CUT] and simple logical manipulations.

As second example we consider the application $g\ x$, with an assumption on the behaviour of g. The intent is to illuminate the use of rely/guarantee formulae and the [XCHANGE] axiom. Let $A \stackrel{def}{=} \mathsf{even}(x) \wedge g(xk)(\mathsf{even}(x) \supset \overline{k}(a)\mathsf{even}(a))$. We want to show that

$$\{A\}\ gx\ :_{\overline{u}} \{\overline{u}(a)\mathsf{even}(a)\}, \tag{1}$$

recalling that $\{B\}\ M\ :_{\overline{m}} \{C\}$ is short for $M :_{\overline{m}}\ \{A\}B$. First we reason as follows.

$$\dfrac{\dfrac{\dfrac{}{1\quad g :_{\overline{m}}\ \overline{m}\langle g\rangle}\textsc{Var}}{2\quad x :_{\overline{n}}\ \overline{n}\langle x\rangle}\textsc{Var}}{3\quad gx :_{\overline{u}}\ \exists m.(\overline{m}\langle f\rangle\circ m(a)\exists n.(\overline{n}\langle x\rangle\circ n(b)\overline{a}\langle bu\rangle))}\textsc{App},1,2$$

$$4\quad \{A\}\ gx\ :_{\overline{u}} \{\overline{u}(a)\mathsf{even}(a)\}.\textsc{Cons},3$$

The interesting step is the last, where we reason as follows.

$$\begin{array}{lll} \exists m.(\overline{m}\langle g\rangle\circ m(a)\exists n.(\overline{n}\langle x\rangle\circ n(b)\overline{a}\langle bu\rangle)) & \supset & \exists m.(\overline{m}\langle g\rangle\circ m(a)\exists n.(\overline{a}\langle xu\rangle)) \supset \\ \exists m.(\overline{m}\langle g\rangle\circ m(a)\overline{a}\langle xu\rangle) & \supset & \exists m.\overline{g}\langle xu\rangle \qquad\qquad\qquad\quad \supset \overline{g}\langle xu\rangle \end{array}$$

The first and third inferences use [CUT], the two others remove unused quantifiers. Theorem 1 shows that $\overline{g}\langle xu\rangle$ is an optimal specification for our program in the sense that anything that can be said at all about the program gx with anchor u can be derived from $\overline{g}\langle xu\rangle$. We continue by deriving (1), using $B \stackrel{def}{=} \mathsf{even}(x) \supset \overline{u}(a)\mathsf{even}(a)$.

$$\begin{array}{lll} \overline{g}\langle xu\rangle & \supset & \{g\langle xu\rangle(\mathsf{even}(x)\wedge B)\}(\mathsf{even}(x)\wedge B) \supset \{g(xu)(\mathsf{even}(x)\wedge B)\}(\mathsf{even}(x)\wedge B) \\ & \supset & \{A\}\overline{u}(a)\mathsf{even}(a) \end{array}$$

The first implication is by [XCHANGE], the others are straightforward strengthening of the precondition, and simple first-order logic manipulations. Now (1) follows by the consequence rule.

The derivation above has a clear 2-phase structure: first a general assertion about the behaviour of the application is derived without assumptions on free variables. Then such assumptions are added using [XCHANGE] and the consequence rule. It is noteworthy that the first phase is mechanical by induction on the syntax of the program, while the second phase takes place without reference to the program. It is possible to use a more traditional style of reasoning, where applications of languages rules and [CONS] are mixed, but this tends to make inferences longer.

Like the rule for application, [REC] is an adaption of the corresponding rule in [16], but forwarding all jumps to the recursion variable g directly to the recursive function at a. This forwarding corresponds to "copy-cat strategies" in game-semantics [1, 19], here realising the feedback loop of jumps to f into a that enables recursion by using tensor. [REC] implies a more convenient rule, given as follows.

$$\frac{\lambda x.M :_{\overline{m}} \overline{m}(a)\forall j \lneq i.\{A[g/a][j/i]\}A}{\texttt{rec}\ g.\lambda x.M :_{\overline{m}} \overline{m}(a)\forall i.A}\ \textsc{Rec}'$$

As first example of using [REC] we consider a simple function $\omega \stackrel{def}{=} \texttt{rec}\ g.\lambda x.gx$ that diverges upon invocation. Since our rules and axioms are for total correctness, we should not be able to specify anything about ω, except that it terminates and returns at its default port when evaluated as an abstraction, i.e. we show: $\omega :_{\overline{u}} \overline{u}(a)a(xu)\mathsf{T}$. Mechanically we infer the following judgement

$$\omega :_{\overline{u}} \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle)$$

We use axiomatic reasoning to obtain $\omega :_{\overline{u}} \overline{u}(a)a(xu)\mathsf{T}$ by [CONS].

$$\begin{array}{lll}
\overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle) & \supset \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\{g\langle xk\rangle\mathsf{T}\}\mathsf{T}) & \supset \\
\overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ \{g\langle xk\rangle\mathsf{T}\}a(xk)\mathsf{T}) & \supset \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ \{\mathsf{fw}_{ga}\}a(xk)\mathsf{T}) & \supset \\
\overline{u}(a)\exists g.a(xk)\mathsf{T} & \supset \overline{u}(a)a(xk)\mathsf{T} &
\end{array}$$

The first line uses [XCHANGE], the next pushes the assumption of the rely/guarantee formula to the left of the evaluation formula. Then we simply replace that assumption by fw_{ga}. We can do this, because that strengthens the assumption, i.e. weakens the rely/guarantee formula. Then we apply [MP]. The last line removes the superfluous quantifier. We note that there is a simpler derivation of the same fact, relying on the implications:

$$\overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle) \quad \supset \quad \overline{u}(a)\mathsf{T} \quad \supset \quad \overline{u}(a)a(xk)\mathsf{T}.$$

The first of those is just weakening of the tensor, while the second is an instance of [NOINFO].

[CCC] says that `callcc` is a constant, always terminating, and returning at the default port, carrying a function, denoted a, as value. This function takes two arguments, x, the name of another function, and m, the default port for the invocation of a. By typing we know that m must be a function invoked with an argument of continuation type $(\alpha)^?$. Whenever a is invoked, it jumps to x, carrying its default port m as first and second argument. In other words, if the invocation at x terminates at its default port, it does so at a's default port. Moreover, x can also jump to m explicitly. Note that m is duplicated [CCC], i.e. used non-linearly. This non-linearity is the reason for the expressive power of functional control.

We consider another example of reasoning about `callcc`: $M \stackrel{def}{=} \texttt{callcc}\ \lambda k.7$. Mechanically, we derive

$$M :_{\overline{u}} \underbrace{\exists m.(\overline{m}(a)a(xr)\overline{x}\langle rr\rangle \circ m(a)\exists n.(\overline{n}(b)b(ks)\overline{s}\langle 7\rangle \circ n(b)\overline{a}\langle bu\rangle))}_{A}$$

Then we use axiomatic reasoning to reach the expected judgement $M :_{\overline{u}} \overline{u}\langle 7\rangle$.

$$\begin{array}{lll} A \supset \exists a.(a(xr)\overline{x}\langle rr\rangle \circ \exists b.(b(ks)\overline{s}\langle 7\rangle \circ \overline{a}\langle bu\rangle)) & \supset \exists ab.(a(xr)\overline{x}\langle rr\rangle \circ \overline{a}\langle bu\rangle \circ b(ks)\overline{s}\langle 7\rangle) & \\ \quad \supset \exists ab.(\overline{b}\langle uu\rangle \circ b(ks)\overline{s}\langle 7\rangle) & \supset \exists ab.\overline{u}\langle 7\rangle & \supset \overline{u}\langle 7\rangle \end{array}$$

[THROW] says that `throw` is a function returning at its default port a function a which takes x as its first argument (by typing a continuation $(\alpha)^?$), and returns at its default port m a second function b, which in turn takes two argument, the first of which is y (of type α). The second argument, the default port of y is ignored, since x will be *jumped* carrying y as argument.

We continue with reasoning about simple programs with `throw`. We show that:

$$\texttt{throw}\ k\ 3 :_{\overline{u}} \overline{k}\langle 3\rangle \qquad\qquad \omega(\texttt{throw}\ k\ 3) :_{\overline{u}} \overline{k}\langle 3\rangle.$$

We begin with the assertion on the left. The assertion for this program will be quite sizable because [APP] must be applied twice. The following abbreviation is useful to shorten specifications arising from [APP].

$$A \mid_{mnu} B \qquad \stackrel{def}{=} \qquad \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{a}\langle bu\rangle)).$$

Here we assume that u,n do not occur in M and u,m are not in N. We let $\mid_{mnu}$ bind less tightly than all the other operators of the logic. This abbreviation is interesting because of the following derived rule, which is immediate from the rules.

$$\overline{m}(a)a(bu)A \mid_{mnu} \overline{n}(b)B \qquad \supset \qquad \exists ab.(A \wedge B). \tag{2}$$

From [THROW], $k :_{\overline{b}} \overline{b}\langle k\rangle$ and $3 :_{\overline{n}} \overline{n}\langle 3\rangle$ we get:

$$\texttt{throw}\ k\ 3 :_{\overline{u}} (\overline{g}(a)a(xm)\overline{m}(b)b(y\cdot)\overline{x}\langle y\rangle) \mid_{gbm} \overline{b}\langle k\rangle \mid_{mnu} \overline{n}\langle 3\rangle$$

which simplifies to $\texttt{throw}\ k\ 3 :_{\overline{u}} \overline{k}\langle 3\rangle$ by applying (2) twice. Now we deal with $\omega(\texttt{throw}\ k\ 3)$. As before: $\omega(\texttt{throw}\ k\ 3) :_{\overline{u}} A$ with $A \stackrel{def}{=} \overline{m}(a)a(bu)\mathsf{T} \mid_{mnu} \overline{k}\langle 3\rangle$, but we cannot apply (2) since $\texttt{throw}\ k\ 3$ does not return at the default port. Instead we reason from the axioms.

$$\exists n.(\overline{k}\langle 3\rangle \circ n(b)\overline{a}\langle bu\rangle) \ \supset\ \exists n.\overline{k}\langle 3\rangle(\mathsf{T} \circ n(b)\overline{a}\langle bu\rangle) \supset \overline{k}\langle 3\rangle\exists n.(\mathsf{T} \circ n(b)\overline{a}\langle bu\rangle) \ \supset\ \overline{k}\langle 3\rangle$$

Here the first line is an application of [CUT], the second switches quantification with a jump, and the third line is by [NOINFO], in addition to straightforward logical manipulations. Thus we can use [CUT] once more and infer:

$$\begin{array}{lll} \overline{m}(a)a(bu)\mathsf{T} \mid_{mnu} \overline{k}\langle 3\rangle & \supset \exists m.((\overline{m}(a).a(bu)\mathsf{T}) \circ m(a)\overline{k}\langle 3\rangle) & \supset \exists mb.((a(bu)\mathsf{T}) \circ \overline{k}\langle 3\rangle) \\ & \supset \overline{k}\langle 3\rangle\exists mb.((a(bu)\mathsf{T}) \circ \mathsf{T}) & \supset \overline{k}\langle 3\rangle \end{array}$$

[IF] simply adds a recipient for the default port at M, the condition of the conditional, where a boolean b is received. Depending on b, the specification of one of the branches is enabled. [ADD] is similar to [APP] and the [CONS], the rule of consequence, is standard in program logics.

A Comment on the Shape of Rules. Program logics are usually presented "bottom-up", meaning that postconditions in the conclusion of rules are just a meta-variable standing for arbitrary (well-typed) formulae. This facilitates reasoning starting from a desired postcondition of the program under specification, and then trying to find an appropriate premise. We have chosen the "top-down" presentation because it gives simpler and more intuitive rules, and shortens inferences substantially. A "bottom-up"presentation of proof rules is possible, and may be useful in some cases. The status of the "bottom-up" rules (e.g. completeness) is yet to be established.

Completeness. A goal of axiomatic semantics is to be in harmony with the corresponding operational semantics. That means that two programs should be contextually indistinguishable if and only if they satisfy the same formulae. This property is called *observational completeness*. We establish observational completeness as a consequence of *descriptive completeness*.

Definition 1. *By $\sqsubseteq$ we mean the standard typed contextual precongurence for PCF^+, i.e. $M \sqsubseteq N$ if for $C[M] \Downarrow$ implies $C[N] \Downarrow$ for all closing contexts $C[\cdot]$, where $\Downarrow$ means termination.*

Theorem 1. *(Descriptive Completeness for Total Correctness) Our logic is* descriptively complete*: for all closed M, N (typable under the same typing), A and m, we have: $\vdash M :_{\overline{m}} A$ implies that (1) $\models M :_{\overline{m}} A$ and (2) whenever $\models N :_{\overline{m}} A$ then $M \sqsubseteq N$.*

The proof of this theorem, and the derivation of observational completeness (as well as relative completeness in the sense of Cook) from descriptive completeness follows [14].

The $\lambda\mu$-Calculus. From the rules and axioms for PCF^+, it is easy to derive a logic for $\mu\mathsf{PCF}$, an extension of the $\lambda\mu$-calculus, a Curry-Howard correspondence for classical logic, with a recursion operator. The logic enjoys similar completeness properties.

$$\frac{M :_m A \quad N :_u B}{M+N :_u \exists mn.(A \wedge B \wedge u = a+b)}\ \textsc{sAdd} \qquad \frac{M :_m A}{\lambda x.M :_u u\langle x\rangle_m A}\ \textsc{sAbs} \qquad \frac{M :_{\overline{m}} A}{\lambda x.M :_u u\langle xm\rangle A}\ \textsc{sAbs'}$$

$$\frac{\lambda x.M :_u A}{\mathtt{rec}\ g.\lambda x.M :_u \exists g.(\mathsf{fw}_{ga} \circ A)}\ \textsc{sRec} \qquad \frac{M :_m A \quad N :_n B}{MN :_{\overline{u}} \exists mn.((A \wedge B) \circ \overline{m}\langle nu\rangle)}\ \textsc{sApp} \qquad \frac{M :_m m\langle n\rangle_u A \quad N :_n B}{MN :_u \exists mn.(A \wedge B)}\ \textsc{sApp'}$$

$$\frac{-}{\mathtt{callcc} :_u u(xm)\overline{x}\langle mm\rangle}\ \textsc{sCcc} \qquad \frac{-}{\mathtt{throw} :_u u\langle x\rangle_m m(y\cdot)\overline{x}\langle y\rangle}\ \textsc{sThrow} \qquad \frac{-}{x :_u \overline{u} = x}\ \textsc{sVar}$$

$$\frac{-}{\mathtt{c} :_u u = \mathtt{c}}\ \textsc{sConst} \qquad \frac{M :_m A \quad N :_u B \quad N' :_u C}{\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' :_u \exists m.(A \circ (m = \mathtt{t} \supset B) \wedge (a = \mathtt{f} \supset C))}\ \textsc{sIf}$$

$$\frac{M :_u A \quad A \supset B}{M :_u B}\ \textsc{sCons} \qquad \frac{M :_{\overline{n}} \exists \tilde{a}.(m\langle e\rangle_u A \circ \overline{m}\langle en\rangle B)}{M :_u \exists \tilde{a}.(A \circ B)}\ \textsc{sCut}$$

Fig. 2. Some derived rules that are useful for reasoning about PCF^+ programs that return at their default port

5 Simplifying Reasoning

PCF-terms are a subset of PCF$^+$-terms. Reasoning about PCF-terms using the logic for PCF$^+$ is moderately more laborious than using a logic tailor-made for PCF like [16]. This is because intermediary jumps in function application are represented explicitly in the former, but not the latter. Reasoning in §4 about simple programs like $(\lambda x.x+1)7$ and `throw` k 3 suggest that intermediate jumps can be eliminated mechanically in applications where a function and its argument return at the default port. We formalise this intuition and obtain simplified derivable logical rules and axioms, that can be used to reason about a large class of programs, including PCF$^+$ programs that do use functional control. We start by defining two syntactic shorthands that apply only to judgements and evaluation formulae that return at their default ports (u fresh in both):

$$M :_m A \stackrel{def}{=} M :_{\overline{u}} \overline{u}(m)A \qquad x\langle\tilde{e}\rangle_m A \stackrel{def}{=} \forall u.x\langle\tilde{e}u\rangle\overline{u}(m)A$$

We write $x(\tilde{y})_m A$ for $\forall\tilde{y}.x\langle\tilde{y}\rangle_m A$. Using this syntax, $\lambda x.x+1$ has the following specification, as we shall show below. $\lambda x.x+1 :_u u\langle x\rangle_m m = x+1$. In order to derive specifications like this more efficiently than by expansion of abbreviations, we introduce *derivable* rules and axioms that work directly with this new syntax. Figure 2 lists some rules. Axioms can be simplified in the same way.

Termination at default ports is not the only place where higher-level rules are useful. Examples in §4 indicate that reasoning about non-default jumps also often follows more high-level patterns. To support this intuition, we add more shorthands.

$$M \nearrow A \stackrel{def}{=} M :_{\overline{u}} A \wedge \overline{m}\langle\cdot\rangle \wedge m \neq u \qquad a \bullet e \nearrow \{A\} \stackrel{def}{=} a\langle eu\rangle(A \wedge \overline{m}\langle\cdot\rangle \wedge m \neq u)$$

In both u must be fresh. Rules using these additional rules can be found in Figure 3.

Theorem 2. *All rules in Figures 2 and 3, and all associated axioms are derivable.*

We continue with some further examples of using the derived rules and axioms. We start by deriving $3 + \texttt{throw}\ k\ 7 :_{\overline{u}} \overline{k}\langle 3\rangle$ once more.

1	$k :_{\overline{n}} \overline{n}\langle k\rangle$	VAR
2	$7 :_h h = 7$	SVAR
3	$\texttt{throw}\ k\ 7 \nearrow \overline{k}\langle y\rangle$	JTHROW"
4	$3 :_m m = 3$	SCONST
5	$3 + \texttt{throw}\ k\ 7 \nearrow \overline{k}\langle y\rangle$	JADD'

Now we consider an example that show that the simplified rules are also useful when reasoning about programs with free variables. Consider

$$\texttt{callcc}\ x :_m \{A\}(m = 7 \vee m = 8) \tag{3}$$

where $A \stackrel{def}{=} x(kr)(\overline{k}\langle 7\rangle \vee \overline{r}\langle 8\rangle)$. Mechanically, using the simplified rules, we infer

1	$\texttt{callcc} :_a a(bc)\overline{b}\langle cc\rangle$	SVAR
2	$x :_b b = x$	SCCC
3	$\texttt{callcc}\, x :_{\overline{u}} \exists ab.(a(bc)\overline{b}\langle cc\rangle \wedge b = x) \circ \overline{a}\langle bu\rangle$	SAPP, 1, 2
4	$\texttt{callcc}\, x :_{\overline{u}} \overline{x}\langle uu\rangle$	CONS, 3
5	$\texttt{callcc}\, x :_m \{A\}(m = 7 \vee m = 8)$	CONS, 4

Line 4 is by a straightforward application of [CUT] and some straightforward logical manipulations. To get Line 5, we reason as follows.

$$\overline{x}\langle uu\rangle \supset \{x\langle uu\rangle(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle)\}(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle) \supset \{A\}(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle) \supset \overline{u}(m)\{A\}(\overline{m}\langle 7\rangle \vee \overline{m}\langle 8\rangle)$$

The first of these implications uses [XCHANGE], while the second strengthens the pre-condition of the rely/guarantee formula.

Example (3) shows how easily we can reason about programs that have free variables which are assumed to act like throwing a continuation. Just as easily one can assume that a variable acts like `callcc` and prove $x\ \lambda k.\texttt{throw}\ k\ 7 :_m \{A\}m = 7$, where $A \stackrel{def}{=} x(ab)\overline{a}\langle bb\rangle$.

$$\frac{M \nearrow A}{\lambda x.M :_u u \bullet x \nearrow \{A\}}\ \text{JABS} \qquad \frac{M :_m A \quad N \nearrow B}{M+N \nearrow B}\ \text{JADD'} \qquad \frac{M \nearrow A}{M+N \nearrow A}\ \text{JADD} \qquad \frac{M :_m A \quad N \nearrow B}{MN \nearrow B}\ \text{JAPP'}$$

$$\frac{M \nearrow A}{\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \nearrow A}\ \text{JADD} \qquad \frac{M \nearrow A}{\texttt{callcc}\ M \nearrow A}\ \text{JCCC} \qquad \frac{M \nearrow A}{MN \nearrow A}\ \text{JAPP} \qquad \frac{M \nearrow A}{\texttt{throw}\ M\ N \nearrow A}\ \text{JTHROW}$$

$$\frac{M :_m A \quad N \nearrow B}{\texttt{throw}\ M\ N \nearrow B}\ \text{JTHROW'} \qquad \frac{M :_{\overline{m}} \overline{m}\langle k\rangle \quad N :_n A}{\texttt{throw}\ M\ N \nearrow \overline{k}(n)A}\ \text{JTHROW''}$$

Fig. 3. Some derived rules, helpful for reasoning about PCF$^+$ programs that jump

Relating the Logics for PCF and PCF$^+$. The derivable rules and axioms just discussed pose the question of the systematic relationship between the present logic and that for PCF [14, 16]. We give an answer by providing a simple translation of formulae and judgements from the logic for PCF to that for PCF$^+$, and then showing that the inclusion on programs preserves derivability. The idea behind the translation is straightforward: just add fresh default ports.

We continue with a summary of the logic for PCF in [14, 16]. Types and formulae are given by the following grammar, with expressions being unchanged.

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid \alpha \to \beta \qquad A ::= e = e' \mid A \wedge B \mid \neg A \mid \forall x^{\alpha}.A \mid x\langle e\rangle_y A$$

Judgements are of the form $\{A\}\ M\ :_m\ \{B\}$. Next is the translation of PCF-formulae into PCF^+-formulae.

$$\ulcorner e = e' \urcorner \stackrel{def}{=} e = e' \qquad \ulcorner A \wedge B \urcorner \stackrel{def}{=} \ulcorner A \urcorner \wedge \ulcorner B \urcorner \qquad \ulcorner \neg A \urcorner \stackrel{def}{=} \neg \ulcorner A \urcorner$$
$$\ulcorner \forall x^{\alpha}.A \urcorner \stackrel{def}{=} \forall x^{\alpha^\circ}.\ulcorner A \urcorner \qquad \ulcorner x\langle e \rangle_y\, A \urcorner \stackrel{def}{=} \forall u.x\langle eu \rangle \overline{u}(y) \ulcorner A \urcorner \quad u \text{ fresh}$$

Please note that the translation changes α to α° in the translation of quantifiers (α° was defined in §3). Judgements are translated as follows: $\ulcorner \{A\}\ M\ :_m\ \{B\} \urcorner \stackrel{def}{=} M :_{\overline{u}} \overline{u}(m)\{\ulcorner A \urcorner\}\ulcorner B \urcorner$ (u fresh). This translation has the following properties.

Theorem 3. *1. The translation of judgements, when applied to rules, takes PCF-rules to derivable rules for PCF^+.*
2. $\vdash \{A\}\ M\ :_m\ \{B\}$ implies $\vdash \ulcorner \{A\}\ M\ :_m\ \{B\} \urcorner$, where derivability on the left is in the logic for PCF, on the right it's for PCF^+.

6 Conclusion

We have investigated program logics for a large class of stateless sequential control constructs. One construct not considered here are exceptions. Exceptions are a constrained form of jumping that is used to escape a context without the possibility of returning, a feature very useful for error handling. Exceptions are not included in the present logic because they are caught dynamically, which does not sit comfortably with our typing system. We believe that a simple extension of the logic presented here can easily account for exceptions. A second omission is that many programming languages with interesting control constructs also feature state. We believe that adding state to PCF^+or $\mu\mathsf{PCF}$ can be done easily with the help of content quantification [16].

Related Work. The present work builds upon a large body of preceding work on the semantics of control, including, but not limited to [11, 17, 21, 22, 25–28]. As mentioned, the investigation of logics for control manipulation was started by Clint and Hoare [10]. It has been revived by [2–4, 7, 24, 29, 32, 34] (the long version of the present paper will feature a more comprehensive discussion). None of these approaches investigates logics for fully-fledged higher-order control constructs like `callcc`.

The present work adds a new member to a family of logics for ML-like languages [5, 16, 18, 36], and integrates in a strong sense: e.g. all rules and axioms from [16] are, adapting the syntax, also valid for $\mathsf{PCF}^+$and $\mu\mathsf{PCF}$. We believe that all common CPS-transforms between PCF, PCF^+ and $\mu\mathsf{PCF}$ are logically fully abstract in the sense of [23]. This coherence between programming languages, their operational and axiomatic semantics, and compilations between each other paves the way for a comprehensive proof-compilation infrastructure for ML-like languages.

Rely/guarantee based reasoning was introduced in [20]. Internalising rely/guarantee reasoning into the program logic itself by way of rely/guarantee formulae was first proposed in [30, 31] and has been used in Ambient Logics [9] and in expressive typing systems [8]. The use of tensor is also found in [30, 31], and has been advocated by Winskel [35]. In all cases the context is concurrency, not sequential control.

A preliminary version of the present work was finished in 2007, and its key ideas, in particular rely/guarantee formulae and the tensor have since lead to a Hennessy-Milner

logic for typed π-calculus [6]. Neither proof-rules nor axioms for higher-order control are investigated in [6]. Clarifying the relationship between the present logic and that of [6] is an interesting research question.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. & Comp. 163, 409–470 (2000)
2. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resource verification. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 34–49. Springer, Heidelberg (2004)
3. Bannwart, F., Müller, P.: A program logic for bytecode. ENTCS 141(1), 255–273 (2005)
4. Benton, N.: A Typed, Compositional Logic for a Stack-Based Abstract Machine. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 364–380. Springer, Heidelberg (2005)
5. Berger, M., Honda, K., Yoshida, N.: A logical analysis of aliasing for higher-order imperative functions. In: Proc. ICFP, pp. 280–293 (2005); Full version to appear in JFP
6. Berger, M., Honda, K., Yoshida, N.: Completeness and logical full abstraction in modal logics for typed mobile processes. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 99–111. Springer, Heidelberg (2008)
7. Beringer, L., Hofmann, M.: A bytecode logic for JML and types. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 389–405. Springer, Heidelberg (2006)
8. Caires, L.: Spatial-behavioral types, distributed services, and resources. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 98–115. Springer, Heidelberg (2007)
9. Cardelli, L., Gordon, A.D.: Anytime, Anywhere. Modal Logics for Mobile Ambients. In: Proc. POPL, pp. 365–377 (2000)
10. Clint, M., Hoare, C.A.R.: Program Proving: Jumps and Functions. Acta Informatica 1, 214–224 (1972)
11. Duba, B.F., Harper, R., MacQueen, D.: Typing First-Class Continuations in ML. In: Proc. POPL, pp. 163–173 (1991)
12. Harper, R., Lillibridge, M.: Operational Interpretations of an Extension of F_ω with Control Operators. Journal of Functional Programming 6(3), 393–417 (1996)
13. Honda, K.: Processes and games. ENTCS 71 (2002)
14. Honda, K., Berger, M., Yoshida, N.: Descriptive and Relative Completeness of Logics for Higher-Order Functions. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 360–371. Springer, Heidelberg (2006)
15. Honda, K., Yoshida, N.: A uniform type structure for secure information flow. In: POPL 2002, pp. 81–92. ACM Press, New York (2002); Full version to appear in ACM TOPLAS
16. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: Proc. PPDP 2004, pp. 191–202. ACM Press, New York (2004)
17. Honda, K., Yoshida, N., Berger, M.: Control in the π-calculus. In: Proc. CW 2004, ACM Press, New York (2004)
18. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order functions. In: LICS 2005, pp. 270–279 (2005)
19. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF. Inf. & Comp. 163, 285–408 (2000)
20. Jones, C.B.: Specification and Design of (Parallel) Programs. In: IFIP Congress, pp. 321–332 (1983)
21. Laird, J.: A Semantic Analysis of Control. PhD thesis, Univ. of Edinburgh (1998)

22. Longley, J.: When is a functional program not a functional program? SIGPLAN Not. 34(9), 1–7 (1999)
23. Longley, J., Plotkin, G.: Logical Full Abstraction and PCF. In: Tbilisi Symposium on Logic, Language and Information. CSLI (1998)
24. Ni, Z., Shao, Z.: Certified Assembly Programming with Embedded Code Pointers. In: Proc. POPL (2006)
25. Ong, C.-H.L., Stewart, C.A.: A Curry-Howard foundation for functional computation with control. In: Proc. POPL, pp. 215–227 (1997)
26. Parigot, M.: $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
27. Plotkin, G.: Call-By-Name, Call-By-Value, and the λ-Calculus. TCS 1(2), 125–159 (1975)
28. Riecke, J.G., Thielecke, H.: Typed exceptions and continuations cannot macro-express each other. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 635–644. Springer, Heidelberg (1999)
29. Saabas, A., Uustalu, T.: A Compositional Natural Semantics and Hoare Logic for Low-Level Languages. In: Proc. Workshop Structural Operational Semantics, SOS (2006)
30. Stirling, C.: A complete compositional proof system for a subset of CCS. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 475–486. Springer, Heidelberg (1985)
31. Stirling, C.: Modal logics for communicating systems. TCS 49, 311–347 (1987)
32. Tan, G., Appel, A.W.: A Compositional Logic for Control Flow. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 80–94. Springer, Heidelberg (2005)
33. Thielecke, H.: Continuations, functions and jumps. Bulletin of EATCS, Logic Column 8 (1999)
34. Thielecke, H.: Frame rules from answer types for code pointers. In: Proc. POPL, pp. 309–319 (2006)
35. Winskel, G.: A complete proof system for SCCS with modal assertions. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 392–410. Springer, Heidelberg (1985)
36. Yoshida, N., Honda, K., Berger, M.: Logical reasoning for higher-order functions with local state. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 361–377. Springer, Heidelberg (2007)