A Compositional Logic for Polymorphic Higher-Order Functions

Kohei Honda
Department of Computer Science
Queen Mary, London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Department of Computing
Imperial College London
yoshida@doc.ic.ac.uk

Abstract

This paper introduces a compositional program logic for higher-order polymorphic functions and standard data types. The logic enables us to reason about observable properties of polymorphic programs starting from those of their constituents. Just as types attached to programs offer information on their composability so as to guarantee basic safety of composite programs, formulae of the proposed logic attached to programs offer information on their composability so as to guarantee fine-grained behavioural properties of polymorphic programs. The central feature of the logic is a systematic usage of names and operations on them, whose origin is in the logics for typed π-calculi. The paper introduces the program logic and its proof rules and illustrates their usage by non-trivial reasoning examples, taking a prototypical call-by-value functional language with impredicative polymorphism and recursive types as a target language.

Categories and Subject Descriptors: F.3.1 [Specifying and Verifying and Reasoning about Programs]:Assertions, Logics of programs, Specification techniques

General Terms: Language, Theory, Verification

1. Introduction

Types and Assertions. In modern functional programming languages such as ML and Haskell [27, 17, 9], as well as in recent object-oriented languages [23, 29], types are positioned as a key element of abstraction in programming. Types in these languages offer a basic compositional specification, in the sense that a composite program can be assigned a type if, and only if, its constituent programs can be assigned types and, moreover, the latter are mutually consistent with each other following a certain rule (for instance, if we wish to apply N to M, we require M has type $\alpha \Rightarrow \beta$ and N has type α for some α and β). Once a type is assignable to a program, we know, for example, it never runs into a constructor error, and that it would evaluate, if ever, to a value of that type, statically guaranteeing the fundamental safety property of programs.

PPDP'04, August 24–26, 2004, Verona, Italy.

Useful as they are, types have basic limitation as specifications, in that they cannot describe (hence cannot guarantee) fine-grained behavioural properties of programs. A type can indicate, for example, a given program is a function from natural numbers to natural numbers: but there is no way to say it is a function which, for example, transforms an even number to an odd number, or, for that matter, any number to its factorial. A significant merit of treating only a basic safety property is that a verification of type conformance of a given program is in many cases computationally feasible. But if we wish to certify more general behavioural properties, types alone cannot serve the purpose.

The present work introduces a simple assertional method for compositional specification of behavioural properties of higher-order programs and data types. Assertions are associated with two typed interface points of a typed program, namely its environment and itself, strictly following type information. Just as types attached to programs offer information on their composability so as to guarantee basic safety of composite programs, formulae attached to programs offer, in the present framework, information on their composability so as to guarantee fine-grained behavioural properties of polymorphic programs. The logical discipline is cleanly stratified on the top of a type discipline, extending specifiable properties to encompass essentially arbitrary observable properties of programs. One of the central elements of the presented program logic is a systematic usage of names and operations on them. This feature comes from a theory of logics for typed π-calculi developed in [21], which is essentially a typed variant of Hennessy-Milner logic [18], and which offers a general basis for a family of program logics through embeddings of the latter in the former. The relationship between the program logics and the process logics is detailed in [22, 21].

Let us briefly illustrate the essence of the proposed assertional method using simple examples.

Basic Ideas. Consider a simple program which computes a doubling function, $N \stackrel{\text{def}}{=} \lambda x.x+x$, with type $\mathbb{N} \Rightarrow \mathbb{N}$ where $\mathbb{N}$ is the type for natural numbers. The type of N represents an abstract specification of this program, saying that if we apply a value of $\mathbb{N}$, then it returns (if ever) a value of type $\mathbb{N}$; but if we apply a value of other types, it guarantees nothing. Thus a function can guarantee a type of the resulting value only relying on types of its arguments.

We extend the same compositional reasoning to more general behavioural properties, using Hoare-style assertions [19, 16]. Take the same N above; we observe that, if we apply 5 to N, then it returns 10; more generally if we apply any natural number to N, it always returns even. And in more detail, it returns the double of an arbitrary argument whenever the latter is well-typed, that is as far as it is a natural number. Thus this function can guarantee a certain behaviour by relying on the property of the argument. We wish to represent these behavioural properties using logical formulae. To do this, we do not mention N itself in a formula, but rather describe its properties by *naming* it as, say, f (any fresh name will do). Thus we can write

$$f \bullet 5 = 10 \tag{1}$$

as a property of N (named as f). Similarly we can write

$$\forall x^{\mathbb{N}}.Even(f \bullet x) \tag{2}$$

where $Even(n)$ is the predicate saying n is even (for example we can set $Even(x) \stackrel{\text{def}}{=} \exists n.(x = 2 \times n)$). The operator $\bullet$ is left associative and non-commutative, which is best understood as the application in applicative structures. We can further refine the formula to say f computes the doubling function. Formulae may be combined using arbitrary standard logical connectives and quantifiers, just as in Hoare Logic.

Using these formulae (which we let range over $A, B, \ldots$), the judgement of the logic has the following shape.

$$A \vdash M :_f B$$

which can be read as:

> If M, named as f, can rely on A as the behaviour of an environment, then the function combined with the environment can guarantee B.

The name f is called *anchor*, which can be any fresh name not occurring in M. An anchor is used for representing the point of operation, hence of specification, of M. Intuitively the above judgement corresponds to a Hoare triple $\{A\}M\{B\}$ where A is about the free variables of M and B is about f which is the name given to M. This idea – naming functions and programs – naturally comes from encodings of functions into the π-calculus [26, 28] where function M is mapped into an agent $[\![M]\!]_f$ equipped with unique name, f to be referred and shared by other agents. Reflecting its process-theoretic origin, the specification method regards programs as interactive entities [26] whose specifications are given via interrogating its behaviour one by one. As an example, we can write down the specification for N as:

$$\mathsf{T} \vdash N :_f \forall x^{\mathbb{N}}.Even(f \bullet x) \tag{3}$$

which says that the program N named as f, under the trivial assumption T on its free variables, satisfies $\forall x^{\mathbb{N}}.Even(f \bullet x)$ (the triviality of the assumption is because no free variables occur in M: the truth T in general indicates the weakest specification allowing any well-typed behaviour).

Let us present further examples of assertions, starting from a simple assertion for the identity function.

$$\mathsf{T} \vdash \lambda x^{\mathbb{N}}.x \ :_u \ \forall y^{\mathbb{N}}.\ u \bullet y = y. \tag{4}$$

The judgement says that the given program, when applied to any argument of type $\mathbb{N}$, will return that same argument as a result.

When this function is applied to the argument, $u \bullet y$ is peeled-off and replaced by a new anchor name m.

$$\mathsf{T} \vdash (\lambda x^{\mathbb{N}}.x)2 \ :_m \ m = 2 \tag{5}$$

Let us take a brief look at how we can compositionally derive (5). We first need the predicate for the argument.

$$\vdash 2 :_z z = 2 \tag{6}$$

which simply says that "2" as a program satisfies, when named as z, the predicate $z = 2$. We then infer:

$$\frac{(4),\ \ (6),\ \ (\forall y^{\mathbb{N}}.\ (u \bullet y = y) \wedge z = 2) \supset u \bullet z = 2}{\mathsf{T} \vdash (\lambda x^{\mathbb{N}}.x)2 \ :_m \ m = 2}$$

In the above derivation (which follows the proof rule for application, formally discussed in Section 4 later), the predicate in the conclusion, "$m = 2$", is obtained by substituting "m" for "$u \bullet z$" in the conclusion part of the entailment in the antecedent. Note this entailment is a simple instance of the standard axiom in the predicate logic with equality, $(A(x,x) \wedge x = y) \supset A(x,y)$ (cf. [25, §2.8]). While validity of such entailment is in general incalculable, it *is* calculable using simple axioms in this case as well as in many practical examples. The nature of the above derivation may become clearer by contrasting it with the corresponding typing derivation.

$$\frac{\vdash \lambda x^{\mathbb{N}}.x : \mathbb{N} \Rightarrow \mathbb{N} \qquad \vdash 2 : \mathbb{N}}{\vdash (\lambda x^{\mathbb{N}}.x)2 : \mathbb{N}}$$

Here, instead of calculating validity of entailment, we simply match the first $\mathbb{N}$ of the function's type $\mathbb{N} \Rightarrow \mathbb{N}$ with the argument's type $\mathbb{N}$, obtaining $\mathbb{N}$. In correspondence with simpler specification, we have only to use simpler, and essentially mechanical, inference.

Next we consider an example which uses a non-trivial assumption on a higher-order variable.

$$\forall x^{\mathbb{N}}.Even(f \bullet x) \vdash f3 + 1 :_u Odd(u) \tag{7}$$

where $Odd(n)$ says n is odd. Using the same environment, we can further derive:

$$\forall x^{\mathbb{N}}.Even(f \bullet x) \vdash f3 + f(f5) + 1 :_u Odd(u) \tag{8}$$

By using name f, the term which contains multiple f can share the single specification in the environment. The derivation of (7) and (8) starts from the following instance of the axiom for a variable.

$$\forall x^{\mathbb{N}}.Even(f \bullet x) \vdash f :_m \forall x^{\mathbb{N}}.Even(m \bullet x) \tag{9}$$

The specification says that if we assume $\forall x^{\mathbb{N}}.Even(f \bullet x)$ for the entity which a variable f denotes, then f as a program named as m satisfies precisely the same property (substituting m for f). Starting from (9), we can easily reach (7) and (8), using the rule for application as well as a similar rule for first-order operations (all of which are later presented in Section 4).

Let $L \stackrel{\text{def}}{=} f3 + f(f5) + 1$ and recall $N \stackrel{\text{def}}{=} \lambda x.x + x$ before. We now consider the following specification.

$$\mathsf{T} \vdash \texttt{let}\ f = N\ \texttt{in}\ L :_u Odd(u) \tag{10}$$

The "let" construct has the standard decomposition into abstraction and application, but has a special status in the present context in that its derivation clearly describes how we can "plug" a specification of a part into a specification of a whole (just as in the cut rule in the sequent calculus). The derivation of (10) follows.

$$\frac{(3),\quad (8),\quad \forall x^{\mathbb{N}}.Even(f \bullet x) \supset \forall x^{\mathbb{N}}.Even(f \bullet x)}{\mathsf{T} \vdash \texttt{let}\ f = N\ \texttt{in}\ L :_u Odd(u)} \tag{11}$$

Note the third condition in the antecedent is a trivially valid tautology. In this tautology, the first $\forall x^{\mathbb{N}}.Even(f \bullet x)$ is the conclusion of (3) while the second one is the assumption of (8): thus the property which is guaranteed by N is simply plugged into the assumption for L. This derivation may be understood as being analogous to a composition rule of Hoare Logic, where we infer $\{A\}P_1;P_2\{B\}$ from $\{A\}P_1\{C_1\}$ and $\{C_2\}P_2\{B\}$ such that $C_1 \supset C_2$. (11) may also be contrasted with the type derivation for the same program.

$$\frac{\vdash N : \mathbb{N} \Rightarrow \mathbb{N} \qquad f : \mathbb{N} \Rightarrow \mathbb{N} \vdash L : \mathbb{N}}{\vdash \texttt{let}\ f = N\ \texttt{in}\ L : \mathbb{N}} \tag{12}$$

which has a shape isomorphic to (11). Further examples of assertions and derivations are found in later sections.

As we already mentioned, the extensive use of names in these specifications and their derivations comes from the underlying π-calculus logic. Cumbersome as they may look, their use allows us to reason about programs and data structures of diverse kinds on a uniform basis, of which the present paper will focus on purely functional ones. Among others we shall treat function types, products, sums, recursion (in terms and in types), and second-order polymorphism (both universals and existentials). Here we only show a simplest example, taking a specification of the universal identity, $\Lambda X.\lambda x^X.x : \forall X.X \Rightarrow X$, leaving other examples to Sections 3 and 4.

$$\forall X.\forall x^X.(u \bullet [X] \bullet x = x). \tag{13}$$

The assertion says that the universal identity named as u, when it is applied to any type α and any well-typed argument y of type α, will return that argument itself as a result.

Summary of Contributions. Some of the possible contributions of the present paper would be:

- Introduction of an assertional method for specifying properties of polymorphic higher-order programs and data types, taking a prototypical call-by-value higher-order polymorphic language as an example.
- Compositional proof rules for the proposed assertional method, ensuring total correctness. As far as we know, this is the first such system for higher-order polymorphic languages.
- Discussions on how the presented method may be used for reasoning about behaviours of programs in higher-order languages and data types, with extensive examples.

It may be worth stressing that the assertional method discussed in the present paper cleanly extends to call-by-name evaluation, imperative procedures as well as data structures with destructive update. Such extensions are detailed in [21].

Related Work. The intersection type disciplines [11, 5, 13, 38] offer one of the most general ways to specify behaviours of programs among various type systems for functions (they are also applied to non-functional calculi recently, cf. [12, 24]). Apart from the fact that they have mainly been studied for untyped calculi, there are two main differences from the presented framework. First, specifications in intersection types are based on conjunction and entailment of a specific kind, whereas the logical language in the present framework allows the full use of standard logical connectives. Second, the properties expressible in intersection types are certain closed sets in the corresponding CPOs, whereas formulae in the presented logic essentially encompass arbitrary observable properties of programs. This second point also applies to Abramsky's domain logic [4], which has a close connection with intersection type disciplines, even though the domain logic is richer in type structures.

Equational logics for the λ-calculi have been studied since the classical work by Curry and Church. LCF [14] augments the standard equational theory of the λ-calculus with Scott's fixed point induction. The program logics for higher-order functions derived from process logics differ in that an assertion in the former describes behavioural properties of programs rather than equates them, allowing specifications with arbitrary degrees of precision, as well as smoothly extending to non-functional behaviour.

The reasoning methods for polymorphic λ-calculi have been studied focusing on the principles of parametricity, either using equational logics [34, 2] or using logical relations [1, 39, 32, 33]. The presented method differs in that it offers behavioural specifications for interface of a program, rather than directly equating/relating programs. It should however be noted that, for calculating validity of entailment, the present method does need to make resort to semantic arguments for polymorphic behaviours. This suggests fruitful interplay between the present logical method, on the one hand, and the reasoning principles as developed in, and extending, [34, 2, 39, 32, 33, 1], on the other.

Reynolds [35] develops a specification logic for Algol, extending Hoare Logic with treatment of higher-order procedures. In addition to basic differences in the constructions of formulae and judgement, the use of the assumption on free variables in the present logic leads to compositional proof rules, unlike in [35].

Reynolds, O'Hearn, Bornat and their colleagues [36, 37, 30, 8, 10] study extensions of Hoare's Logic with an aim to offer an effective reasoning method for low-level operations, including pointers, memory allocation/deallocation, and garbage collection. A central idea of their approach is to explicitly assume a dynamically allocated region of memory cells in the universe of discourse, and reason about them based on a conjunction which at the same time implies disjointness of such regions. The present framework is cleanly extensible to impure higher-order procedures as well as shared data structures with destructive update [21]. It would be an interesting subject of study to integrate the methods for low-level reasoning as cultivated through their studies with the present framework.

Finally, as we already noted, the origin of the present work is in the logic for typed π-calculi. [22, 21, 20] discuss in detail the theory of process logics for sequentially typed processes, together with their relationship to the program logics.

Structure of the Paper. Section 2 reviews the syntax and operational semantics of the polymorphic PCF used in the paper; Section 3 introduces the assertions and its semantics; Section 4 defines the proof rules; Section 5 shows non-trivial reasoning examples; Section 6 concludes the paper with further topics. The extensive discussions on the general framework of the logics, including the presentation of its imperative and other extensions, the soundness proofs, detailed comparisons with related work, as well as connections to the corresponding process logics, are found in [20, 21, 22].

2. Call-by-Value Polymorphic PCF

2.1 Syntax and Typing

As a target language, we choose the call-by-value PCF with unit, sums and products, extended with universal and existential second-order polymorphism as well as recursive types [31, 15, 40]. Apart from the treatment of recursive types (which relies on implicit type conformance for simplicity of associated proof rules), all constructs are standard. Our presentation essentially follows Pierce's recent textbook [31] (many program examples are also from his book).

We first list the grammar of programs, assuming an infinite set of term variables (sometimes called *names*) as well as an infinite set of type variables ($X, Y, \ldots$). Below let i range over $\{1,2\}$.

$$\begin{array}{rcl}
M & ::= & () \mid \mathbf{n} \mid \mathbf{b} \mid x \mid \lambda x^{\alpha}.M \mid MN \mid \mu x^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M \\
 & \mid & \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \mid \mathbf{op}(\vec{M}) \mid \langle M_1, M_2\rangle \\
 & \mid & \pi_i(M) \mid \mathtt{in}_i(M) \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).N_i\}_{i\in\{1,2\}} \\
 & \mid & \Lambda X.M \mid M\beta \mid \mathtt{pack}\ \langle \beta, M\rangle\ \mathtt{as}\ \exists X.\alpha \\
 & \mid & \mathtt{unpack}\ M\ \mathtt{as}\ \langle X, x\rangle\ \mathtt{in}\ N \\
V & ::= & () \mid \mathbf{n} \mid \mathbf{b} \mid x \mid \lambda x^{\alpha}.M \mid \mu^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M \\
 & \mid & \langle V_1, V_2\rangle \mid \mathtt{in}_i(V) \mid \Lambda X.V \mid \mathtt{pack}\ \langle \beta, V\rangle\ \mathtt{as}\ \exists X.\alpha
\end{array}$$

$M, N, \ldots$ denote general terms (sometimes called programs), while

$V, W, \ldots$ denote values. The grammar uses types $(\alpha, \beta, \ldots)$ which are given later. Binding etc. are standard. $\mathsf{fv}(M)/\mathsf{ftv}(M)$ denotes a set of free variables/type variables. Bound variables are annotated by types, though in examples we often omit them. $\mathbf{n}$ stands for non-negative numerals (1,2,...) and $\mathbf{b}$ for booleans ($\mathbf{t}$, $\mathbf{f}$). () is unit. We often use $\mathbf{c}$ for constants. $\mathbf{op}(M_0, ..., M_{n-1})$ denotes an n-ary arithmetic and boolean operation. Among others we consider the unary operations $\mathtt{succ}(M)$ (the successor) and $\neg M$ (the negation) and binary operations $M+N$, $M-N$, $M \times N$, $M=N$ (equality of two numbers), $M \wedge N$ and $M \vee N$. We use the standard weak call-by-value one-step reduction [15, 40], written $M \longrightarrow M'$. We only list the main rules for universals and existentials.

$$(\Lambda X.M)\beta \longrightarrow M[\beta/X]$$
$$\mathtt{unpack}\,(\mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists X.\alpha)\,\mathtt{as}\,\langle X, x\rangle\,\mathtt{in}\,N \longrightarrow N[\beta/X][V/x]$$

Types, ranged over by $\alpha, \beta, \ldots$, are generated from:

$$\alpha \ ::= \ \mathtt{Unit} \mid \mathbb{B} \mid \mathbb{N} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \mid X \mid \forall X.\alpha \mid \exists X.\alpha \mid \mu X.\alpha$$

Again binding is standard. $\mathtt{Unit}$, $\mathbb{B}$ and $\mathbb{N}$ are called *atomic types*. The typing judgements have the form $\Gamma \vdash M : \alpha$, where Γ is a *base*, which is a finite map from variables to types. Well-typed terms are derived from the rules in Figure 1 (only part of constants and operators are treated, which are easily extended to other cases). All rules are standard: in the rules for polymorphism, the constructors (type abstraction and pack) and the destructors (type concretion and unpack) compensate with each other. In the rule for recursive type (based on the equi-recursive approach [31]), $\approx$ denotes the tree equivalence of two types, which in particular includes the standard (un)folding, $\mu X.\alpha \approx \alpha[\mu X.\alpha/X]$. We often write $M^{\Gamma;\alpha}$ to denote a typable term M such that we have $\Gamma \vdash M : \alpha$. We shall call the language PolyPCFv.

$$[Var]\ \frac{-}{\Gamma, x:\alpha \vdash x:\alpha} \quad [Unit]\ \frac{-}{\Gamma \vdash () : \mathtt{Unit}}$$

$$[Num]\ \frac{-}{\Gamma \vdash \mathbf{n} : \mathbb{N}} \quad [Bool]\ \frac{-}{\Gamma \vdash \mathbf{b} : \mathbb{B}}$$

$$[Succ]\ \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathtt{succ}(M) : \mathbb{N}} \quad [Eq]\ \frac{\Gamma \vdash M_{1,2} : \mathbb{N}}{\Gamma \vdash M_1 = M_2 : \mathbb{B}}$$

$$[Abs]\ \frac{\Gamma, x:\alpha \vdash M : \beta}{\Gamma, x:\alpha \vdash \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \quad [App]\ \frac{\Gamma \vdash M : \alpha \Rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

$$[Rec]\ \frac{\Gamma, x : \alpha \Rightarrow \beta \vdash \lambda y^{\alpha}.M : \alpha \Rightarrow \beta}{\Gamma \vdash \mu x^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M : \alpha \Rightarrow \beta}$$

$$[If]\ \frac{\Gamma \vdash M : \mathbb{B} \quad \Gamma \vdash N_{1,2} : \alpha}{\Gamma \vdash \mathtt{if}\,M\,\mathtt{then}\,N_1\,\mathtt{else}\,N_2 : \alpha}$$

$$[Inl]\ \frac{\Gamma \vdash M : \alpha_1}{\Gamma \vdash \mathtt{in}_1(M) : \alpha_1 + \alpha_2} \quad [Inr]\ \frac{\Gamma \vdash M : \alpha_2}{\Gamma \vdash \mathtt{in}_2(M) : \alpha_1 + \alpha_2}$$

$$[Case]\ \frac{\Gamma \vdash M : \alpha_1 + \alpha_2 \quad \Gamma, x_i : \alpha_i \vdash M_i : \beta}{\Gamma \vdash \mathtt{case}\,M\,\mathtt{of}\,\{\mathtt{in}_i(x_i).M_i\}_{i \in \{1,2\}} : \beta}$$

$$[Pair]\ \frac{\Gamma \vdash M_i : \alpha_i \ (i=1,2)}{\Gamma \vdash \langle M_1, M_2\rangle : \alpha_1 \times \alpha_2} \quad [Proj]\ \frac{\Gamma \vdash M : \alpha_1 \times \alpha_2}{\Gamma \vdash \pi_i(M) : \alpha_i \ (i=1,2)}$$

$$[T\text{-}Abs]\ \frac{\Gamma \vdash M : \alpha \quad X \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda X.M : \forall X.\alpha} \quad [T\text{-}App]\ \frac{\Gamma \vdash M : \forall X.\alpha}{\Gamma \vdash M\beta : \alpha[\beta/X]}$$

$$[T\text{-}Pack]\ \frac{\Gamma \vdash M : \alpha[\beta/X]}{\Gamma \vdash \mathtt{pack}\,\langle \beta, M\rangle\,\mathtt{as}\,\exists X.\alpha : \exists X.\alpha}$$

$$[T\text{-}Unpack]\ \frac{\Gamma \vdash M : \exists X.\alpha \quad \Gamma \cdot x : \alpha \vdash N : \beta \quad X \notin \mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\beta)}{\Gamma \vdash \mathtt{unpack}\,M\,\mathtt{as}\,\langle X, x\rangle\,\mathtt{in}\,N : \beta}$$

$$[T\text{-}Rec]\ \frac{\Gamma \vdash M : \alpha \quad \alpha \approx \beta}{\Gamma \vdash M : \beta}$$

Figure 1: Typing Rules for PolyPCFv

2.2 Programming Examples

In the following we explore the expressiveness of PolyPCFv using simple programs, which are later used as running examples. As a simple instance of the universal abstraction, we take the standard universal identity.

$$\vdash \Lambda X.\lambda x^{X}.x : \forall X.X \Rightarrow X. \tag{14}$$

To use this identity, we instantiate it with a type and feed a value of that type:

$$((\Lambda X.\lambda x^{X}.x)\mathbb{N})2 \longrightarrow (\lambda x^{\mathbb{N}}.x)2 \longrightarrow 2. \tag{15}$$

The second example is Church's Polymorphic Booleans.

$$\mathsf{tru} = \Lambda X.\lambda x^{X}.\lambda y^{X}.x \qquad \mathsf{fls} = \Lambda X.\lambda x^{X}.\lambda y^{X}.y \tag{16}$$

Each function takes two arguments and returns one of them. We assign a common type to tru and fls assuming two arguments have the same type, but this type may be arbitrary since tru and fls do not use their arguments except return one of them. Then we can write boolean operations like not by constructing a new boolean that uses an existing one to decide which of its arguments to return. Let $\mathsf{Bool} \stackrel{\text{def}}{=} \forall X.X \Rightarrow X \Rightarrow X$ below.

$$\vdash \lambda b : \mathsf{Bool}.\Lambda X.\lambda t^{X}.\lambda f^{X}.b\,X\,f\,t \ : \ \mathsf{Bool} \Rightarrow \mathsf{Bool} \tag{17}$$

The third example is a type for a list of elements of type α, using recursive types.

$$\mathtt{List}(\alpha) \stackrel{\text{def}}{=} \mu Y.(\mathtt{Unit} + (\alpha \times Y)). \tag{18}$$

The empty list is represented by $\mathtt{in}_1(())$, while the list which consists of a head M (of type α) and a tail L (of type $\mathtt{List}(\alpha)$) becomes $\mathtt{in}_2(\langle M, L\rangle)$, which is of type $\mathtt{List}(\alpha)$ by $\mathtt{Unit} + \alpha \times \mathtt{List}(\alpha) \approx \mathtt{List}(\alpha)$. A list can be infinite, e.g. $\mu x.\mathtt{in}_2(\langle 1, x\rangle)$ is an infinite list of 1 and has type $\mathtt{List}(\mathbb{N})$. In our subsequent discussions we often treat lists. For legibility we use the following notations.

$$[\varepsilon] \stackrel{\text{def}}{=} \mathtt{in}_1(()), \qquad [M :: L] \stackrel{\text{def}}{=} \mathtt{in}_2(\langle M, L\rangle).$$

As is well-known, it is often convenient to treat lists generically, i.e. without depending on its types. In such situations, we can use the polymorphic list type $\forall X.\mathtt{List}(X)$. A simple example is the generic cons function:

$$\Lambda X.\,\lambda x^{X}.\lambda l^{\mathtt{List}(X)}.[x :: l]. \tag{19}$$

A little more complex example is a program which calculates the length of a list of an arbitrary type.

$$\Lambda X.\mu f^{\mathtt{List}(X) \Rightarrow \mathbb{N}}.\lambda l^{\mathtt{List}(X)}.\ \mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow 0 \mid \mathsf{isCons}(x, l') \Rightarrow f(l') + 1 \tag{20}$$

Above we use "$\mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow M \mid \mathsf{isCons}(x, l') \Rightarrow N$" which stands for

$$\mathtt{case}\ l\,\mathtt{of}\,\{\mathtt{in}_1(z).M \mid \mathtt{in}_2(y).((\lambda x l'.N)\pi_1(y))\pi_2(y)\}\ (z, y \text{ fresh})$$

Yet another example which uses a polymorphic list is a program which filters out elements of a list that satisfy a given predicate.

$$\vdash \Lambda \mathrm{X}.\, \lambda f^{\mathrm{X}\Rightarrow\mathbb{B}}.\mathsf{filter}(f) \;:\; \forall \mathrm{X}.\, (\mathrm{X}\Rightarrow\mathbb{B})\Rightarrow \mathtt{List}(\mathrm{X})\Rightarrow \mathtt{List}(\mathrm{X}) \quad (21)$$

where $\mathsf{filter}(f)$ is the following expressions:

$$\begin{array}{l} \mu g.\lambda l.\mathtt{case}\ l\ \mathtt{of}\ \mathtt{isNil}\Rightarrow[\varepsilon] \\ \qquad\qquad \mid \mathtt{isCons}(x,y)\Rightarrow \mathtt{if}\ f(x)\ \mathtt{then}\ [x::g(y)]\ \mathtt{else}\ g(y). \end{array}$$

We end our brief exploration with a simple use of existential abstraction. Below we use records, which are in the present case simply a labelled version of products.

$$\begin{array}{ll} \mathtt{pack} & \langle \mathbb{N}, \{\mathtt{new}:\lambda().0,\ \mathtt{inc}:\lambda x.x+1,\ \mathtt{get}:\lambda x.x\}\rangle \\ \mathtt{as} & \exists \mathrm{X}.\{\mathtt{new}:\mathtt{Unit}\Rightarrow\mathrm{X},\ \mathtt{inc}:\mathrm{X}\Rightarrow\mathrm{X},\ \mathtt{get}:\mathrm{X}\Rightarrow\mathbb{N}\} \end{array} \quad (22)$$

where $\lambda().M$ is a thunk which formally stands for $\lambda x^{\mathtt{Unit}}.M$ with $x \notin \mathsf{fv}(M)$. The type represents an abstract data type for a counter, which has three methods: "`new`" exports an opaque integer, "`inc`" increments it, and "`get`" turns an opaque value to the corresponding non-opaque value.

3. Assertions for PolyPCFv

3.1 Syntax

In this section we introduce formulae and judgement for a program logic for PolyPCFv, illustrate them by examples, and present their semantics. As in the standard Hoare logic, a specification for a program can be either about total correctness (where non-trivial assertions always guarantee termination) or about partial correctness (where divergence allows arbitrary specifications). These two, together with a third one which subsumes both, are discussed in detail in [21]. In the present paper we focus on the logics for total correctness, which allows a most straightforward presentation, both in syntax and semantics. The presentation places a particular emphasis on the type-oriented nature of the logic.

Terms and Formulae. A judgement in the present logic uses a pair of first-order logical formulae to describe behavioural properties of typed functionals. Formulae are made up from equations on terms and their combinations by standard logical connectives. The grammar of a language for the total logic for PolyPCFv follows.

$$\begin{array}{lll} e & ::= & x^{\alpha} \mid () \mid \mathbf{n} \mid \mathbf{b} \mid \mathbf{op}(e_1,..,e_n) \mid \perp^{\alpha} \mid e_1 \bullet e_2 \\ & \mid & \langle e_1, e_2\rangle \mid \pi_i(e) \mid \mathtt{in}_i^{\alpha+\beta}(e) \mid e\bullet[\beta] \mid \langle \beta, e\rangle^{\exists \mathrm{X}.\alpha} \\ A & ::= & e_1 = e_2 \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \supset A_2 \\ & \mid & \forall x.A \mid \exists x.A \mid \forall \mathrm{X}.A \mid \exists \mathrm{X}.A \end{array}$$

The first set of expressions $(e, e', \ldots)$ are called *terms*, while the second ones $(A, B, C \ldots)$ *formulae*. Terms contain variables, which we often call *names*. They also include constants such as unit $()$, natural numbers $\mathbf{n}$ and booleans $\mathbf{b}$. $\mathbf{op}$ indicates a first-order operation (usually carried over from the target programming language). $\perp^{\alpha}$ denotes divergence with type α, which is seldom used in specifications but is needed for semantic completeness. We write $e \Downarrow$ for $e \neq \perp$. $e_1 \bullet e_2$ is an application of e_1 and e_2 where $\bullet$ is left associative and non-commutative. Terms also includes the standard arithmetic operators. The first three terms in the second line are paring, projection and injection, respectively. The last two forms of terms are a type application for polymorphic universal type and packing for existential type, respectively. Operationally $e \bullet [\beta]$ corresponds to $M\beta$, while $\langle \beta, e\rangle$ corresponds to a packed pair.

In formulae, logical connectives are used with their standard precedence/association: e.g. $A \wedge B \supset \forall x.C \vee D \supset E$ is parsed as $(A\wedge B) \supset (((\forall x.C) \vee D) \supset E)$. $A \equiv B$ denotes A and B are logically equivalent. As far as no confusion arises, we use these connectives both syntactically (i.e. as connectives in formulae in program logics) and semantically (i.e. for discussing validity of various kinds). We also use the truth T (definable as, for example, $1 = 1$) and the falsity F (which is $\neg\mathsf{T}$). T and F also denote boolean values. The quantifications induce binding, for which we assume the standard bound name convention. $\mathsf{fv}(A)/\mathsf{ftv}(A)$ denotes the set of free variables/type variables in A.

Below we define the well-typed expressions and formulae.

Definition 3.1 (well-typedness in terms)

- $()$, $\mathbf{n}$, $\mathbf{b}$, x^{α} and $\perp^{\alpha}$ are well-typed with types $\mathtt{Unit}$, $\mathbb{N}$, $\mathbb{B}$, α and α, respectively.
- $e_1 \bullet e_2$ is well-typed with type β if $e_{1,2}$ are well-typed with types $\alpha \Rightarrow \beta$ and α, respectively.
- $\neg e$ is well-typed with type $\mathbb{B}$, if e is well-typed with type $\mathbb{B}$. $e_1 + e_2$ is well-typed with type $\mathbb{N}$ if $e_{1,2}$ are well-typed with type $\mathbb{N}$. Similarly for $\mathtt{succ}$, $\times$, $-$, etc.
- $\langle e_1, e_2\rangle$ is well-typed with type $\alpha_1 \times \alpha_2$ if $e_{1,2}$ are well-typed with types $\alpha_{1,2}$. $\pi_1(e)$ is well-typed with type α if e is well-typed with type $\alpha \times \beta$, similarly for $\pi_2(e)$.
- $\mathtt{in}_1^{\alpha+\beta}(e)$ is well-typed with type $\alpha+\beta$ if e is well-typed with type α. Similarly for $\mathtt{in}_2^{\alpha+\beta}(e)$.
- $e \bullet [\beta]$ is well-typed with type $\alpha[\beta/\mathrm{X}]$ if e is well-typed with type $\forall \mathrm{X}.\alpha$. $\langle \beta, e\rangle^{\exists \mathrm{X}.\alpha}$ is well-typed with type $\exists \mathrm{X}.\alpha$ if e is well-typed with type $\alpha[\beta/\mathrm{X}]$.

A is *well-typed* if whenever a variable/name occurs twice they own the same type and each pair of equated terms have the same type. Hereafter we only consider well-typed terms and formulae and omit type annotations if no ambiguity arises (or if ambiguity does not matter).

Judgement. There are two forms of the judgement, one for semantic validity and another for syntactic derivability.

$$A \models M^{\Gamma;\alpha} :_u B, \qquad\qquad A \vdash M^{\Gamma;\alpha} :_u B$$

In both forms of sequents, we assume $\Gamma \vdash M : \alpha$ and $u \notin \mathsf{fn}(\Gamma)$. The *primary names in A* are those in $\mathsf{dom}(\Gamma)$, while the *primary name* of B is u. Other names occurring in A and B are *auxiliary*. Then free names in A (resp. in B) should be typed under $\Gamma \cdot \Theta$ (resp. $u : \alpha \cdot \Theta$) for a common Θ, which is a map from the auxiliary names to types. Each sequent can be read as:

> *for any closed values of types Γ satisfying A, the result of substituting them for variables in M leads to the behaviour satisfying B, under an arbitrary interpretation of auxiliary names in A and B.*

Later we shall formalise this idea, first by defining semantics for validating the first form of sequent and, secondly, by introducing proof rules for deriving the second form of sequent. The term *assertion* is sometimes used for formulae used in a judgement, as well as for a judgement itself.

3.2 Examples of Assertions

We first recall several examples from Introduction. First,

$$Even(y) = \exists x^{\mathbb{N}}.\, (y^{\mathbb{N}} = 2 \times x)$$

is the standard predicate which says y is even, while:

$$Double(u) = \forall n^{\mathbb{N}}.(u^{\mathbb{N}\Rightarrow\mathbb{N}} \bullet n = 2 \times n)$$

says that a function u always returns double of the argument, which is satisfied by, for example, $\lambda x.x+x$. The following entailment is valid with respect to semantics of formulae given later.

$$Double(u) \supset \forall y^{\mathbb{N}}.Even(u^{\mathbb{N}\Rightarrow\mathbb{N}} \bullet y).$$

Next we recall another example from Introduction:

$$ID(u) \stackrel{\text{def}}{=} \forall \mathrm{X}.\forall x^{\mathrm{X}}.u^{\forall \mathrm{X}.\mathrm{X}\Rightarrow\mathrm{X}} \bullet [\mathrm{X}] \bullet x = x\ .$$

This is an assertion for the universal identity $\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x$, which means that the given program, when it is applied to any type α and any well-typed argument y of type α, will return that argument as a result. In contrast, $ID^{\mathbb{N}}(u) \stackrel{\text{def}}{=} \forall x^{\mathbb{N}}.(u^{\mathbb{N}\Rightarrow\mathbb{N}} \bullet x = x)$ is an assertion satisfied by the monomorphic identity $\lambda x^{\mathbb{N}}.x$. These two assertions are related in the following way:

$$\exists m.(u = m \bullet [\mathbb{N}] \wedge ID(m)) \quad\equiv\quad ID^{\mathbb{N}}(u)$$

As a simple example of sums and products, let us define $A(m) \stackrel{\text{def}}{=} \exists y'z'.(m = \langle \mathtt{in}_1(y'), z'\rangle \wedge Even(y') \wedge Odd(z'))$. Then we have, for example, $A(\langle \mathtt{in}_1(y), z\rangle) \supset Even(y)$.

A list can be encoded into product and sums with recursive types. Thus we can reason about a list using $\mathtt{in}_1(())$ and $\mathtt{in}_2(\langle M, L\rangle)$ as encodings of the nil $[\varepsilon]$ and the cons $[M :: L]$, respectively. However having terms for the list as such in assertions, written $[\varepsilon]$ (for $\mathtt{in}_1(())$) and $[e :: e']$ (for $\mathtt{in}_2(\langle e, e'\rangle)$), is convenient, especially when we reason about complex programs which process lists. As an example, using these notations, we can define a derived predicate for the well-foundedness (i.e. finiteness) of a list as follows.

$$\begin{array}{rcl} Wf(l) & \equiv & \exists n^{\mathbb{N}}.Len(l,n). \\ Len(l,0) & \equiv & l = [\varepsilon]. \\ Len(l,n+1) & \equiv & \exists x, l'.\, l = [x :: l'] \wedge x \neq \perp \wedge Len(l', n). \end{array}$$

Next, we recall the program $\mathtt{cntr}$ in (22) in Section 2.

$$\begin{array}{ll} \mathtt{pack} & \langle \mathbb{N}, \{\mathtt{new} : \lambda().0,\ \ \mathtt{inc} : \lambda x.x+1,\ \ \mathtt{get} : \lambda x.x\}\rangle \\ \mathtt{as} & \exists \mathrm{X}.\{\mathtt{new} : \mathtt{Unit} \Rightarrow \mathrm{X},\ \ \mathtt{inc} : \mathrm{X} \Rightarrow \mathrm{X},\ \ \mathtt{get} : \mathrm{X} \Rightarrow \mathbb{N}\} \end{array}$$

The following is a specification for this program before packing, named as w (we use a labelled projection $\mathtt{l}(e)$).

$$\exists y^{\mathrm{X}}, z^{\mathrm{X}}.(y = \mathtt{new}(w) \bullet () \ \wedge\ z = \mathtt{inc}(w) \bullet y \ \wedge\ \mathtt{get}(w) \bullet z \geq 1)$$

A more readable specification may use a notation that combines an application and a projection, writing $e.\mathtt{l}(e')$ for $(e.\mathtt{l}) \bullet (e')$.

$$\exists y^{\mathrm{X}}, z^{\mathrm{X}}.(w.\mathtt{new}() = y \ \wedge\ w.\mathtt{inc}(y) = z \ \wedge\ w.\mathtt{get}(z) \geq 1)$$

Now we consider the data hiding. Let:

$$\begin{array}{rcl} C(u, \mathrm{X}) & \stackrel{\text{def}}{=} & \exists w^{\alpha}, y^{\mathrm{X}}, z^{\mathrm{X}}.(u = \langle \mathrm{X}, w\rangle \wedge D(w, y, z)) \\ D(w, y, z) & \stackrel{\text{def}}{=} & w.\mathtt{new}() = y \ \wedge\ w.\mathtt{inc}(y) = z \ \wedge\ w.\mathtt{get}(z) \geq 1 \end{array}$$

We write $C(\langle \mathrm{X}, x\rangle, \mathrm{X})$ for $C(u, \mathrm{X})[\langle \mathrm{X}, x\rangle / u]$. Then:

- $\exists \mathrm{X}.C(u, \mathrm{X})$ hides a concrete type of an ADT, giving a specification of the packed program above with anchor u.
- In contrast, $C(\langle \mathrm{X}, x\rangle, \mathrm{X})$ opens an ADT, under which, for example, $\lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}()))$ with anchor u satisfies $u \bullet () \geq 1$.

Using these predicates in intermediate steps, we shall later show the following unpacked program, with anchor u, satisfies $u \bullet () \geq 1$.

$$\mathtt{unpack\ cntr\ as}\ \langle \mathrm{X}, x\rangle\ \mathtt{in}\ \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}()))$$

3.3 Semantics of Assertions

Below we define semantics of assertions taking a quickest path, using congruence classes of PolyPCFv (for further discussions on models, including those based on typed processes, see [21]). We use the standard typed congruence. Suppose $\Gamma \vdash M_{1,2} : \alpha$. Then $\Gamma \vdash M_1 \cong_{\lambda} M_2 : \alpha$ iff, for each closing $C[M_i]$ of type $\mathbb{N}$, we have $C[M_1] \Downarrow$ iff $C[M_2] \Downarrow$ where $M \Downarrow$ means $\exists N.M \longrightarrow^* N \not\longrightarrow$.

Model. Let $\kappa, \kappa', \ldots$ range over the $\cong_{\lambda}$-congruence classes of typed closed terms, which we call *behaviours*. We write κ^{α} if κ is of type α. Then for each type α, there is a unique congruent class of the diverging terms, which we write $\perp^{\alpha}$ or simply $\perp$. If $\kappa \neq \perp$, we say κ is a *total behaviour*. A *model* ξ is a well-typed finite map from names to total behaviours. Given $\kappa_1^{\alpha\Rightarrow\beta}$ and κ_2^{α}, $\kappa_1 \bullet \kappa_2$ is given as the congruence class including MN for $M \in \kappa_1$ and $N \in \kappa_2$. Similarly, given $\kappa^{\forall \mathrm{X}.\alpha}$ and β, $\kappa \bullet [\beta]$ is given as the congruence class including $M\beta$ for $M \in \kappa$. Then given $\kappa^{\alpha[\beta/\mathrm{X}]}$ and β, $\langle \beta, \kappa\rangle$ is given as the congruence class including $\mathtt{pack}\,\langle \mathbb{N}, M\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha$ for $M \in \kappa$. Arithmetic operations etc. are similarly defined.

Satisfaction. Fix auxiliary names and type variables in A. An *interpretation* I maps auxiliary names to both total and non-total behaviours, as well as type variables to closed types (the former should be consistent with the latter). Given a model ξ and an interpretation I, the map $[\![e]\!]_{I\cdot\xi}$ is defined by induction on e as follows:

- $[\![\mathbf{c}]\!]_{I\cdot\xi} = \mathbf{c}$, $[\![\perp]\!]_{I\cdot\xi} = \perp$ and $[\![x]\!]_{I\cdot\xi} = (I \cup \xi)(x)$.
- $[\![e \bullet e']\!]_{I\cdot\xi} = [\![e]\!]_{I\cdot\xi} \bullet [\![e']\!]_{I\cdot\xi}$
- $[\![e + e']\!]_{I\cdot\xi} = [\![e]\!]_{I\cdot\xi} + [\![e']\!]_{I\cdot\xi}$, similarly for others.
- $[\![\langle e, e'\rangle]\!]_{I\cdot\xi} = \langle [\![e]\!]_{I\cdot\xi}, [\![e']\!]_{I\cdot\xi}\rangle$, $[\![\pi_i(e)]\!]_{I\cdot\xi} = \pi_i([\![e]\!]_{I\cdot\xi})$ and $[\![\mathtt{in}_i(e)]\!]_{I\cdot\xi} = \mathtt{in}_i([\![e]\!]_{I\cdot\xi})$.
- $[\![e \bullet [\beta]]\!]_{I\cdot\xi} = [\![e]\!]_{I\cdot\xi} \bullet [\beta]$ and $[\![\langle \beta, e\rangle]\!]_{I\cdot\xi} = \langle \beta, [\![e]\!]_{I\cdot\xi}\rangle$

The satisfaction of A by a model ξ under an interpretation I, written $\xi \models^I A$, is given by induction on the structure of A. We start from:

$$\xi \models^I e_1 = e_2 \quad\equiv\quad [\![e_1]\!]_{I\cdot\xi} = [\![e_2]\!]_{I\cdot\xi}$$

Logical connectives are given the standard (classical) interpretation (below connectives on the right-hand side are about validity):

$$\begin{array}{rcl} \xi \models^I A_1 \wedge A_2 & \equiv & (\xi \models^I A_1) \quad\wedge\quad (\xi \models^I A_2) \\ \xi \models^I \neg A & \equiv & \neg\, (\xi \models^I A) \\ \xi \models^I \forall x^T.A & \equiv & \forall \mathsf{c} \in T.\ \xi \models^{I\cdot x:\mathsf{c}} A, \\ \xi \models^I \forall x^{\alpha}.A & \equiv & \forall \kappa \in [\![\alpha]\!].\ \xi \models^{I\cdot x:\kappa} A, \\ \xi \models^I \forall \mathrm{X}.A & \equiv & \forall \alpha.\ \xi \models^{I\cdot x:\alpha} A \end{array}$$

where, in the second last line, $[\![\alpha]\!]$ denotes the set of all behaviours of type α. The rest is de Morgan duality. We then write $\kappa \models^I_u A$ (read: *κ with anchor u satisfies A under I*) when $u : \kappa \models^I A$ ($u : \kappa$ is a mode which maps u to κ). Finally with M closed, $M \models^I_u A$ stands for $\exists \kappa.\ (M \in \kappa \wedge \kappa \models^I_u A)$. We also write $\vec{x} : \vec{V} \models^I A$ for $\vec{x} : \vec{\kappa} \models^I A$ with $V_i \in \kappa_i$. We can now define:

Definition 3.2 (semantics of assertions) $A \models M^{\vec{x}:\vec{\beta};\alpha} :_u B$ *iff, under each well-typed I and for each $\vec{V}^{\vec{\beta}}$ such that $\vec{x} : \vec{V} \models^I A$, we have $M[\vec{V}/\vec{x}] \models^I_u B$.*

Note that, by definition, if we have $A \models M :_u B$ for A such that $A \not\equiv \mathsf{F}$, it always holds $M \Downarrow$. Also note the validity of A is far from being effectively calculable since the logic properly extends the standard number theory (thus, for example, true formulae in the present logic are not recursively enumerable). This does not, however, preclude the construction and use of compositional proof rules for the logic, where the calculation of validity is often reduced to mechanical inferences. The compositional proof system for valid assertions is not only important for verifications: it offers a fundamental insight on the nature of each construct of the language. This is the theme of the next section.

4. Proof Rules for PolyPCFv

This section introduces proofs rules for PolyPCFv including derived rules. The shape of each rule naturally follows that of the corresponding typing rule (except structural rules which only manipulate formulae), making the reasoning principle compositional. We recall, from Section 3.1, that the main sequent for provability has the form:

$$A \vdash M^{\Gamma;\alpha} :_u B$$

where $\Gamma \vdash M : \alpha$ and names in A and B should be well-typed (as specified in Section 3.1). The well-typedness demands, among others, u never occurs in A, and free variables in Γ never occur in B. We often call A and B the *rely formula* and the *guarantee formula*, respectively. Variables which occur in Γ are the *primary names in* A, u is the *primary name in* B while names which occur in A and B but not in $\mathsf{dom}(\Gamma) \cup \{u\}$ are *auxiliary names*. These notions play a key role in the consistency of the proof rules.

The proof rules are listed in Figure 2. *In each rule, we assume all assertions are well-typed and a primary name and auxiliary names are never mixed in sequents in the antecedent.* Symbols $i, j, \ldots$ exclusively range over auxiliary names. In the following we illustrate each proof rule one by one.

First-Order Rules. We start from the basic rules: [*Var*] says that, if something can be said about what x denotes in the environment, then the same thing can be said about x as a term, named as u. Similarly for [*Const*] and [*Succ*]. For arithmetical/boolean operations, we have, for example:

$$[Add]\ \frac{A \vdash M :_v B_1 \quad A \vdash N :_w B_2 \quad (B_1 \wedge B_2) \supset B[v+w/u]}{A \vdash M+N :_u B}$$

Similarly for the equality operator:

$$[Eq]\ \frac{A \vdash M^{\mathbb{N}} :_{v_1} B_1 \quad A \vdash N^{\mathbb{N}} :_{v_2} B_2 \quad (B_1 \wedge B_2) \supset B[v_1{=}v_2/b]}{A \vdash M{=}N :_b B}$$

¿From these rules one can guess the general form of the proof rule for an n-ary operator (of which constants and unary/binary operators are special cases), given as $[Op]$ in Figure 2. The rule assumes op also appears as a term constructor in the logical language.

Next we move to the proof rule for the conditional, which says that, under A, if a boolean term M (named as b) satisfies A', and B holds for (1) N_1 under the assumption A' holds and (2) N_2 under the assumption A' does not hold, then, again under A, surely B holds for $\mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2$.

$$[If]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 :_u B}$$

Note the rule keeps clean symmetry, as in the corresponding rule in Hoare logic. Observe also $b \notin \mathsf{fv}(A)$ by the well-formedness of $A \vdash M :_b A'$. Another observation is that $A \vdash M :_b A'$ indicates (as far as A is non-trivial) M terminates. Thus, in a closed term, the conditional branch can surely be evaluated, reaching one of N_i, which in turn is guaranteed to terminate and satisfies B.

$$[Var]\ \frac{-}{A[x/u] \vdash x :_u A} \qquad [Const]\ \frac{-}{A[\mathbf{c}/u] \vdash \mathbf{c} :_u A}$$

$$[Succ]\ \frac{A \vdash M :_v B[v{+}1/u]}{A \vdash \mathtt{succ}(M) :_u B}$$

$$[Op]\ \frac{A \vdash M_i :_{m_i} B_i \ \ (1 \leq i \leq n) \quad (\wedge_i B_i) \supset B[\mathsf{op}(m_1,..,m_n)/u]}{A \vdash \mathsf{op}(M_1,..,M_n) :_u B}$$

$$[If]\ \frac{A \vdash M :_b A' \quad A \wedge A'[\mathsf{T}/b] \vdash N_1 :_u B \quad A \wedge A'[\mathsf{F}/b] \vdash N_2 :_u B}{A \vdash \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 :_u B}$$

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

$$[App]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \wedge B_2 \supset B[m \bullet x/u] \wedge m \bullet x \Downarrow}{A \vdash MN :_u B}$$

$$[Rec]\ \frac{A \vdash \lambda y.M :_u B(0) \qquad A \wedge B(i)[x/u] \vdash \lambda y.M :_u B(i+1)}{A \vdash \mu x.\lambda y.M :_u \forall i.B(i)}$$

$$[Consequence]\ \frac{A \supset A_0 \quad A_0 \vdash M :_u B_0 \quad B_0 \supset B}{A \vdash M :_u B}$$

$$[Pair]\ \frac{A \vdash M_i :_{m_i} B_i \quad B_1 \wedge B_2 \supset B[\langle m_1, m_2\rangle/u]}{A \vdash \langle M_1, M_2\rangle :_u B}$$

$$[Proj_1]\ \frac{A \vdash M :_m B[\pi_1(m)/u]}{A \vdash \pi_1(M) :_u B} \qquad [In_1]\ \frac{A \vdash M :_m B[\mathtt{in}_1(m)/u]}{A \vdash \mathtt{in}_1(M) :_u B}$$

$$[Case]\ \frac{A \vdash M :_m E \quad A^{\text{-}\vec{x}} \wedge E[\mathtt{in}_i(x_i)/m] \vdash N_i :_u B}{A \vdash \mathtt{case}\, M\, \mathtt{of}\, \{\mathtt{in}_i(x_i).N_i\} :_u B}$$

$$[T\text{-}Abs]\ \frac{A^{\text{-}\mathrm{X}} \vdash M :_m B}{A \vdash \Lambda \mathrm{X}.M :_u \forall \mathrm{X}.B[u \bullet [\mathrm{X}]/m]}$$

$$[T\text{-}App]\ \frac{A \vdash M :_m \forall \mathrm{X}.B[m \bullet [\mathrm{X}]/u]}{A \vdash M\beta :_u B[\beta/\mathrm{X}]}$$

$$[T\text{-}Pack]\ \frac{A \vdash M :_m B[\langle \mathrm{X}, m\rangle/u][\beta/\mathrm{X}]}{A \vdash \mathtt{pack}\, \langle \beta, M\rangle\, \mathtt{as}\, \exists \mathrm{X}.\alpha :_u \exists \mathrm{X}.B}$$

$$[T\text{-}Unpack]\ \frac{A \vdash M :_m \exists \mathrm{X}.E \quad A^{\text{-}x\mathrm{X}} \wedge E[\langle \mathrm{X}, x\rangle/m] \vdash N :_u B}{A \vdash \mathtt{unpack}\, M\, \mathtt{as}\, \langle \mathrm{X}, x\rangle\, \mathtt{in}\, N :_u B}$$

$$[T\text{-}Rec]\ \frac{A' \vdash M :_m B' \quad A' \approx A \quad B' \approx B}{A \vdash M :_m B}$$

Figure 2: Proof Rules for PolyPCFv

Proof Rules for Higher-Order Functions. We now move to three of the key rules for PolyPCFv-logic, abstraction, application and recursion. Below recall A^x indicates the only *primary* name in A is x, $\vec{i}$ means a (possibly empty) vector of names, while $A^{\text{-}x\vec{i}}$ says no names from $x\vec{i}$ occur in A.

$$[Abs]\ \frac{A^{\text{-}x\vec{i}} \wedge A_1^x \vdash M :_m A_2 \quad \forall x\vec{i}.(A_1 \supset A_2[u \bullet x/m]) \supset B}{A \vdash \lambda x.M :_u B}$$

The rule says that, whenever M named as m satisfies A_2 relying on A (which is not about x) and A_1 (which is about x), then $\lambda x.M$ named u has the behaviour such that, whenever u is given x satisfying A_1, it returns the result $(u \bullet x)$ which satisfies A_2 for m. This rule can be decomposed into:

$$[\mathit{Abs\text{-}sub}]\,\frac{A \vdash M :_m B,\ i \text{ fresh}}{A[i/x] \vdash \lambda x.M :_u B[m \bullet i/u]}$$

$$[\supset]\,\frac{A \wedge B \vdash M^{\Gamma;\alpha} :_u C,\ \mathsf{fv}(B) \cap \mathsf{fv}(\Gamma) = \emptyset}{A \vdash M^{\Gamma;\alpha} :_u B \supset C}$$

However we prefer [*Abs*] because of their convenience in reasoning. The provability does not differ in the presence of [$\supset$] and [*Aux*-$\forall$] (which is given later). Next we look at the rule for application.

$$[\mathit{App}]\ \frac{A \vdash M :_m B_1 \quad A \vdash N :_x B_2 \quad B_1 \wedge B_2 \supset B[m \bullet x/u] \wedge m \bullet x \Downarrow}{A \vdash MN :_u B}$$

The rule says that, if M, named as m, satisfies B_1, and N, named as x, satisfies B_2, and B_1 and B_2 say that m applies to x converges (recall $m \bullet x \Downarrow$ stands for $m \bullet x \neq \bot$, cf.§ 3.1), then MN named as u satisfies B whenever B is a consequence of B_1 and B_2 where, in the latter, the returned value u is replaced by $m \bullet x$.

Related to the above two proof rules, it is instructive to examine the rule for the `let` construct.

$$[\mathit{Let}]\ \frac{A \vdash M :_x A' \quad A \wedge A' \vdash N :_u B}{A \vdash \mathtt{let}\, x = M \,\mathtt{in}\, N :_u B}.$$

The rule can be derived from [*Abs*] and [*App*] via the standard translation, $\mathtt{let}\, x = M \,\mathtt{in}\, N \stackrel{\text{def}}{=} (\lambda x.N)M$. Note x cannot occur in A since x is an anchor.

Proof Rules for Recursion. Below n is a fresh variable of $\mathbb{N}$-type.

$$[\mathit{Rec}]\ \frac{A^{\text{-}x} \vdash \lambda y.M :_u B(0) \quad A \wedge B(n)[x/u] \vdash \lambda y.M :_u B(n+1)}{A \vdash \mu x.\lambda y.M :_u \forall n.B(n)}$$

Under A, the term satisfies, without assuming anything about x, already $B(0)$ (where $B(\,\cdot\,)$ is a formula with a hole). The term also satisfies, if we assume $B(n)$ for x in addition to A, $B(n+1)$. So we expect it satisfies $B(n)$ for each n. It is also notable that the `letrec` construct, with the standard operational semantics [15], has a clean proof rule. Below let V have an arrow type.

$$[\mathit{Letrec}]\ \frac{\begin{array}{c} A \vdash V[y/x] :_x E(0) \\ A \wedge E(n)[y/x] \vdash V[y/x] :_x E(n{+}1) \\ A \wedge \forall n.E(n) \vdash M :_u B \end{array}}{A \vdash \mathtt{letrec}\, x = V \,\mathtt{in}\, M :_u B}$$

Structural Rules. We list the consequence rule and the two associated rules for auxiliary names (there are other natural structural rules which we omit).

$$[\mathit{Conseq}]\ \frac{A \supset A' \quad A' \vdash M :_u B' \quad B' \supset B}{A \vdash M :_u B}$$

$$[\mathit{Aux}\text{-}\forall]\ \frac{A^{\text{-}i} \vdash M :_u B}{A \vdash M :_u \forall i.B} \qquad [\mathit{Aux}\text{-}\exists]\ \frac{A \vdash M :_u B^{\text{-}i}}{\exists i.A \vdash M :_u B}$$

To put the consequence rule to real use (as well as all other rules which use validity of formulae), we may as well use inference rules for the underlying semantic structure, in addition to the standard rules for predicate calculus with equality and number theory. Three basic rules are given as follows:

$$\begin{array}{ll} \textbf{(ext)} & \forall y.\, e_1 \bullet y = e_2 \bullet y \quad \supset \quad e_1 = e_2 \\ \textbf{(}\bot\textbf{-}\bullet\textbf{)} & \bot \bullet e = \bot \quad e \bullet \bot = \bot \\ \textbf{(}\bot\textbf{-op)} & \mathsf{op}(e_1, .., \bot, .., e_n) = \bot \end{array}$$

All are easily justifiable from the underlying model given in § 3.3 (for further account of such axioms see [21, § 10]).

Sum and Products. The rules for sum and products in Figure 2 should be understood as [*Op*] and [*If*]. Note how the introduction of constructors and destructors in programs is directly reasoned using the corresponding constructs (terms) in assertions.

Products and sums are naturally extended to their labelled versions, records and variants. For example, records add the term of the form $\{l_i : e_i\}$ and $e.l$, interpreted as (labelled) products and their projection. The proof rules become:

$$[\mathit{Record}]\,\frac{A \vdash M_i :_{m_i} B_i \quad \wedge B_i \supset B[\{l_i : m_i\}/u]}{A \vdash \{l_i : M_i\} :_u B}$$

$$[\mathit{Sel}_l]\ \frac{A \vdash M :_m B[m.l/u]}{A \vdash M.l :_u B}$$

Polymorphism and Recursive Types. Finally we explain the rules for the second order polymorphism. [*T-Abs*] and [*T-App*] in Figure 2 are similarly explained as [*Abs*] and [*App*], respectively. Here we focus on the packing and unpacking rules for $\exists$.

$$[\mathit{T\text{-}Pack}]\ \frac{A \vdash M :_m B[\langle \mathrm{X}, m\rangle/u][\beta/\mathrm{X}]}{A \vdash \mathtt{pack}\, \langle \beta, M\rangle \,\mathtt{as}\, \exists \mathrm{X}.\alpha :_u \exists \mathrm{X}.B}$$

The rule says that, if $M^{\alpha[\beta/\mathrm{X}]}$, named as m, satisfies $B[\langle \beta, m^{\alpha[\beta/\mathrm{X}]}\rangle/u]$, then $\mathtt{pack}\,\langle \beta, M\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha$ named u satisfies $\exists \mathrm{X}.B$ (we remind that $\langle \beta, e^{\alpha[\beta/\mathrm{X}]}\rangle$ means a packed expression whose type is $\exists \mathrm{X}.\alpha$). Note that we create a packing M of β by replacing a pack of its anchor name m and X by fresh anchor name u (note the essentially same technique is used in the rule for paring, [*Pair*]).

[*T-Unpack*] is defined symmetrically as follows.

$$[\mathit{T\text{-}Unpack}]\ \frac{A \vdash M :_m \exists \mathrm{X}.E \quad A^{\text{-}x\mathrm{X}} \wedge E[\langle \mathrm{X}, x\rangle/m] \vdash N :_u B}{A \vdash \mathtt{unpack}\, M \,\mathtt{as}\, \langle \mathrm{X}, x\rangle \,\mathtt{in}\, N :_u B}$$

The rule says that, if M named as m satisfies $\exists \mathrm{X}.E$ and N named as n satisfies B with m packed as $\langle \mathrm{X}, x\rangle$, then the resulting term satisfies B. Unpacking is simply handled by an instantiation of a pack of x and X. Intuitively here hiding is ensured by linking one name in a pack by another name in another pack. In the rule for recursive types, [*T-Rec*], $A \approx A'$ denotes, as before, two formulae are identical except for substituting occurrences of $\approx$-related types. We can also use the extensionality of operators for $e \bullet e'$, $e \bullet [\alpha]$, and $\langle e, e'\rangle$ and the entailment $\forall \mathrm{X}.A \supset A[\alpha/\mathrm{X}]$ and $A[\alpha/\mathrm{X}] \supset \exists \mathrm{X}.A$.

Soundness. We conclude this section with the key property of these proof rules, their soundness with respect to the semantics of assertions. This can be established via embedding into the π-calculus, following three steps:

- We first translate PolyPCFv-terms into the second-order polymorphic π-calculus preserving types. Then we prove *equational full abstraction* up to the contextual coungruences, using the method in [6, 7].
- Secondly, we translate PolyPCFv-assertions into those of the second-order polymorphic π-calculus. Then we prove the *logical full abstraction* of the translation (i.e. a PolyPCFv-formula is valid iff the translation is valid in the corresponding process logic. [21].
- Finally, since the process logic satisfies the soundness [20], we can prove, for each $\vec{V}^{\vec{\beta}}$ such that $\vec{x} : \vec{V} \models^{I} A$, we have $M[\vec{V}/\vec{x}] \models^{I}_{u} B$ via the translation of terms and formulae [21].

Since the logic for the π-calculus is with much fewer constructs, this gives a transparent proof. The details are found in [21, 20].

Theorem 4.1 (soundness) *$A \vdash M :_u B$ implies $A \models M :_u B$.*

5. Reasoning Examples

This section illustrates the use of proof rules introduced in the previous section by a few non-trivial reasoning examples. We also introduce a couple of derived rules which can directly reason high-level data structures.

Universal Identity. We first infer the property of the *universal* identity. This is a simplest example to demonstrate how we can use an algebra of names and type variable and compose specifications. For legibility, we often reuse the anchor name. Recall (14):

$$\vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x : \forall \mathrm{X}.\mathrm{X} \Rightarrow \mathrm{X}.$$

for which we infer:

1.	$\mathsf{T} \wedge x = l \vdash x :_m m = l$	(Var)
2.	$\mathsf{T} \vdash \lambda x^{\mathrm{X}}.x :_u \forall xl.(x = l \supset u^{\mathrm{X} \Rightarrow \mathrm{X}} \bullet x = l)$	(Abs)
3.	$\mathsf{T} \vdash \lambda x.x :_u \forall x^{\mathrm{X}}.(u^{\mathrm{X} \Rightarrow \mathrm{X}} \bullet x = x)$	(Conseq)
4.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall \mathrm{X}.\forall x^{\mathrm{X}}.((u^{\forall \mathrm{X}.\mathrm{X} \Rightarrow \mathrm{X}} \bullet [\mathrm{X}]) \bullet x = x)$	(T-Abs)

The resulting assertion says that $\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x$ is a polymorphic function named as u which takes two arguments, arbitrary type X and value with type X, and just returns the value as it is. Recall (15). ¿From the above inference we can infer:

5.	$\mathsf{T} \vdash (\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x)\mathbb{N} :_u \forall x^{\mathbb{N}}.(u^{\mathbb{N} \Rightarrow \mathbb{N}} \bullet x = x)$	(T-App)
6.	$\mathsf{T} \vdash ((\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x)\mathbb{N})3 :_u u = 3$	(App, Conseq)

Next we infer an assertion with a non-trivial assumption. We start from Line 4 in the preceding inference, and infer as follows.

5′.	$\mathsf{T} \vdash (\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x)\mathbb{N} :_u \forall x^{\mathbb{N}}.(u^{\mathbb{N} \Rightarrow \mathbb{N}} \bullet x = x)$	(T-App)
6′.	$Even(y) \vdash y + 1 :_m Odd(m)$	(Var,Op,Conseq)
7′.	$Even(y) \vdash ((\Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x)\mathbb{N})(y+1) :_w Odd(w)$	(App,Conseq)

Line 7′ is derived from the following inference.

$$\begin{array}{lll} (\forall x.u \bullet x = x \ \wedge \ Odd(m)) & \supset & (u \bullet m = m \ \wedge \ Odd(m)) \\ & \equiv & (w = m \ \wedge \ Odd(m))[u \bullet m/w] \end{array}$$

Since $(w = m \ \wedge \ Odd(m)) \supset Odd(w)$, this satisfies the side condition of (App), $(B_1 \wedge B_2) \supset B[u \bullet m/w]$, hence we reach Line 7′.

Convention 5.1 *In the following inferences we allow free variables of a program to occur in the guarantee formula in a sequent, with the same semantics as Definition 3.2 except "$M[\vec{V}/\vec{x}] \models^{I}_{u} B$" in the definition is replaced by "$M[\vec{V}/\vec{x}] \models^{I \cdot \vec{x}:\vec{V}}_{u} B$". The proof rules stay precisely the same.*

While the strict usage of names as we have been obeying so far does not lose generality, the new convention above is useful for making the reasoning shorter, especially in complex examples.

Church's Polymorphic Booleans. Next recall Church Polymorphic Booleans whose type is $\mathsf{Bool} = \forall \mathrm{X}.\mathrm{X} \Rightarrow \mathrm{X} \Rightarrow \mathrm{X}$ and the not function.

$$\mathsf{tru} \ = \ \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.\lambda y^{\mathrm{X}}.x \qquad \mathsf{fls} \ = \ \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.\lambda y^{\mathrm{X}}.y$$
$$\vdash \lambda b : \mathsf{Bool}.\Lambda \mathrm{X}.\lambda t^{\mathrm{X}}.\lambda f^{\mathrm{X}}.b\,\mathrm{X}\,f\,t \ : \ \mathsf{Bool} \Rightarrow \mathsf{Bool}$$

We first infer the property of tru as follows.

1.	$\mathsf{T} \vdash x^{\mathrm{X}} :_u u = x$	(Var)
2.	$\mathsf{T} \vdash \lambda y^{\mathrm{X}}.x^{\mathrm{X}} :_u \forall y^{\mathrm{X}}.u^{\mathrm{X} \Rightarrow \mathrm{X}} \bullet y = x$	(Abs)
3.	$\mathsf{T} \vdash \lambda x^{\mathrm{X}}.\lambda y^{\mathrm{X}}.x^{\mathrm{X}} :_u \forall x^{\mathrm{X}}.\forall y^{\mathrm{X}}.u^{\mathrm{X} \Rightarrow \mathrm{X} \Rightarrow \mathrm{X}} \bullet x \bullet y = x$	(Abs)
4.	$\mathsf{T} \vdash \mathsf{tru} :_u \forall \mathrm{X}.\forall x^{\mathrm{X}}.\forall y^{\mathrm{X}}.u^{\mathsf{Bool}} \bullet [\mathrm{X}] \bullet x \bullet y = x$	(T-Abs)

The specification exactly represents the polymorphic truth. Similarly we can specify fls by swapping x and y. Let us denote boolean specification as:

$$\begin{array}{lll} BTru(u) & = & \forall \mathrm{X}.\forall x^{\mathrm{X}}.\forall y^{\mathrm{X}}.u^{\mathsf{Bool}} \bullet [\mathrm{X}] \bullet x \bullet y = x \\ BFls(u) & = & \forall \mathrm{X}.\forall x^{\mathrm{X}}.\forall y^{\mathrm{X}}.u^{\mathsf{Bool}} \bullet [\mathrm{X}] \bullet x \bullet y = y \end{array}$$

Now we infer the property of not. We define:

$$BNot(u) = \forall b.((BTru(b) \supset BFls(u \bullet b)) \wedge (BFls(b) \supset BTru(u \bullet b)))$$

which means if it gets the truth, then it returns the false and if it gets the false, then it returns the truth.

1.	$BTru(b) \vdash b :_m BTru(m)$	(Var)
2.	$BTru(b) \vdash b\,\mathrm{X} :_m \forall y^{\mathrm{X}}.\forall x^{\mathrm{X}}.m \bullet y \bullet x = y$	(T-App)
3.	$BTru(b) \vdash b\,\mathrm{X}\,y :_m \forall x^{\mathrm{X}}.m \bullet x = y$	(App)
4.	$BTru(b) \vdash b\,\mathrm{X}\,yx :_m m = y$	(App)
5.	$BTru(b) \vdash \lambda y.b\,\mathrm{X}\,yx :_m \forall y.m \bullet y = y$	(Abs)
6.	$BTru(b) \vdash \lambda x.\lambda y.b\,\mathrm{X}\,yx :_m \forall x.\forall y.m \bullet x \bullet y = y$	(Abs)
7.	$BTru(b) \vdash \lambda \mathrm{X}.\lambda x.\lambda y.b\,\mathrm{X}\,yx :_m BFls(m)$	(T-Abs)
8.	$\mathsf{T} \vdash \mathsf{not} :_u \forall b.(BTru(b) \supset BFls(u \bullet b))$	(Abs)
9.	$\mathsf{T} \vdash \mathsf{not} :_u BTru(b) \supset BFls(u \bullet b)$	(Conseq)

Symmetrically we can infer:

$$\mathsf{T} \vdash \mathsf{not} :_u BFls(b) \supset BTru(u \bullet b)$$

Now we apply the following derived proof rule to 9 and the above.

$$[And] \quad \frac{A \vdash M :_u B_i \quad (i = 1, 2)}{A \vdash M :_u B_1 \ \wedge \ B_2}$$

Noting b is an auxiliary variable, an application of $[Aux\text{-}\forall]$ gives us the following desired proof.

$$\mathsf{T} \vdash \mathsf{not} :_u BNot(u)$$

Polymorphic Recursive Function and Lists. Recall the program $\mathsf{filter}(f)$ which filters a list using a function $f : \mathrm{X} \Rightarrow \mathbb{B}$, given in (21). The following program takes off all occurrences of 0 from a list of natural numbers using $\mathsf{filter}(f)$.

$$\mathsf{elimZero} \stackrel{\text{def}}{=} ((\Lambda \mathrm{X}.\lambda f^{\mathrm{X} \Rightarrow \mathbb{B}}.\mathsf{filter}(f))\mathbb{N})(\lambda x^{\mathbb{N}}.x \neq 0).$$

We can easily see this program does not terminate if an argument is an infinite list. Thus, to prove total correctness, we should specify that a given list is well-founded and finite, and that each element terminates. We define this and other related notions as derived predicates.

$$\begin{array}{rcl} Wf(l) & \equiv & \exists n^{\mathbb{N}}.Len(l, n). \\ Len(l, 0) & \equiv & l = [\varepsilon]. \\ Len(l, n+1) & \equiv & \exists x, l'.\, l = [x :: l'] \wedge x \neq \bot \wedge Len(l', n). \end{array}$$

One of the natural specifications for $\mathsf{elimZero}$ is given as:

$$ElimZero(u) \stackrel{\text{def}}{=} \forall l^{\mathtt{List}(\mathbb{N})}.\ (Wf(l) \supset A(u, l) \wedge B(u, l)) \qquad (23)$$

where we set:

$$A(u,l) \stackrel{\text{def}}{=} l=[\varepsilon] \supset u \bullet l = [\varepsilon].$$
$$B(u,l) \stackrel{\text{def}}{=} l=[x::y] \wedge x \neq \bot \supset (x \neq 0 \supset u \bullet l = [x :: u \bullet y]) \wedge (x=0 \supset u \bullet l = u \bullet y).$$

The predicate $ElimZero(u)$ (on u of type $\mathtt{List}(\mathbb{N}) \Rightarrow \mathtt{List}(\mathbb{N})$) says that, assuming an integer list is well-founded, applying the list to u would result in another integer list which is the same as the original one except that each 0 in the list is eliminated one by one. Our aim is to prove:

$$\mathsf{T} \vdash \mathsf{elimZero} :_u ElimZero(u). \qquad (24)$$

To derive (24), the key step is to infer the following generic assertion for the program $\mathsf{filter}(f)$.

$$\mathsf{T} \vdash \mathsf{filter}(f) :_u \forall l.\ Filter(u,l,f) \qquad (25)$$

where $Filter(u,l,f)$ says $u \bullet l$ is the result of filtering l by f:

$$Filter(u,l,f) \stackrel{\text{def}}{=} Wf(l) \supset A(u,l) \wedge C(u,l,f) \qquad (26)$$

where $C(u,l,f)$ is defined as follows:

$$l=[x::y] \wedge x \neq \bot \supset (f \bullet x = \mathsf{T} \supset u \bullet l = [x :: u \bullet y]) \wedge (f \bullet x = \mathsf{F} \supset u \bullet l = u \bullet y).$$

Once we have (25), we can reach (24) as follows.

1.	$\mathsf{T} \vdash \mathsf{filter}(f) :_u \forall l.\ Filter(u,l,f)$	(Assumption)
2.	$\mathsf{T} \vdash \lambda f.\mathsf{filter}(f) :_u \forall f,l.\ Filter(u \bullet f,l,f)$	(Abs)
3.	$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f) :_u \forall \mathrm{X}.\forall f,l.\ Filter(u \bullet [\mathrm{X}] \bullet f,l,f)$	(T-Abs)
4.	$\mathsf{T} \vdash (\Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f))\mathbb{N} :_u \forall f,l.\ Filter(u \bullet f,l,f)$	(T-App)
5.	$\mathsf{T} \vdash \lambda x.(x \neq 0) :_f NonZero(f)$	(Var, Num, Eq, Abs)
6.	$\mathsf{T} \vdash ((\Lambda \mathrm{X}.\lambda f.\mathsf{filter}(f))\mathbb{N})(\lambda x.x \neq 0) :_u ElimZero(u)$	(4, 5, App)

In Line 5, $NonZero(f)$ stands for:

$$\forall x.((x=0 \supset f \bullet x = \mathsf{F}) \wedge (x = n \geq 1 \supset f \bullet x = \mathsf{T}).$$

Then in Line 6, we calculate the assertions as:

$$\forall l.\ Filter(u \bullet f,l,f) \wedge NonZero(f) \supset ElimZero(u \bullet f).$$

We now derive (25). The derivation is, as usual, divided into the base case and the induction. For reasoning, we first note the following inductive definition of the filter predicate.

$$Filter(u,l,f) \equiv \forall n^{\mathbb{N}}.(Len(l,n) \supset Fil(l,f,u \bullet l,n))$$
$$Fil(l,f,l',0) \stackrel{\text{def}}{=} l=l'$$
$$Fil(l,f,l',n{+}1) \stackrel{\text{def}}{=} (l=[\varepsilon] \supset l'=[\varepsilon]) \wedge (l=[x::y] \wedge x \neq \bot \supset (f \bullet x = \mathsf{T} \supset \exists y'.(l'=[x::y'] \wedge Fil(y,f,y',n))) \wedge (f \bullet x = \mathsf{F} \supset Fil(y,f,l',n))$$

The equivalence with (26) is immediate by induction on the length of a list. We also let:

$$M \stackrel{\text{def}}{=} \mathtt{case}\ l\ \mathtt{of}\ \mathsf{isNil} \Rightarrow [\varepsilon] \mid \mathsf{isCons}(x,y) \Rightarrow N$$
$$N \stackrel{\text{def}}{=} \mathtt{if}\ f(x)\ \mathtt{then}\ [x::g(y)]\ \mathtt{else}\ g(y)$$

Note $\mathsf{filter}(f) \stackrel{\text{def}}{=} \mu g.\lambda l.M$. We also use the following abbreviations for brevity.

$$E(n)(u) \equiv \forall l.(Len(l,n) \supset Fil(l,f,u \bullet l,n))$$

$$[Nil]\ \frac{-}{A[[\varepsilon]/u] \vdash [\varepsilon] :_u A}$$

$$[Cons]\ \frac{A \vdash M :_m B_1 \quad A \vdash L :_l B_2 \quad B_1 \wedge B_2 \supset B[[m::l]/u]}{A \vdash [M :: L] :_u B}$$

$$[List]\ \frac{A \vdash M :_m E \qquad A \wedge E[[\varepsilon]/m] \vdash N_1 :_u B \quad A \wedge E[[x::l]/m] \vdash N_2 :_u B}{A \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathsf{isNil} \Rightarrow N_1 \mid \mathsf{isCons}(x,l) \Rightarrow N_2 :_u B}$$

Figure 3: Proof Rules for Lists and Case

The base case follows.

1.	$l=[\varepsilon] \vdash l :_{l'} l' = [\varepsilon]$	(Var)
2.	$[\varepsilon]=[\varepsilon] \vdash [\varepsilon] :_m m=[\varepsilon]$	(Nil)
3.	$\mathsf{F} \vdash \mathtt{if}\ f(x)\ \mathtt{then}\ [x::g(y)]\ \mathtt{else}\ g(y) :_m m=[\varepsilon]$	(falsity)
4.	$l=[\varepsilon] \vdash M :_m Fil(l,f,m,0)$	(1,2,3,List)
5.	$\mathsf{T} \vdash \lambda l.M :_m E(0)(m)$	(Abs)

In Lines 2 and 4, the derived proof rules for lists in Figure 3 are used where we assume $[\varepsilon]$ and $[e::e']$ are included among the terms in the logic (which formally stand for $\mathtt{in}_1(())$ and $\mathtt{in}_2(\langle e,e' \rangle)$, respectively).

We move to the induction step. Below we use the following abbreviations:

$$C(lxyn) \stackrel{\text{def}}{=} l=[x::y] \wedge Len(y,n).$$
$$G(fxl'yn) \stackrel{\text{def}}{=} (f \bullet x = \mathsf{T} \supset \exists y'.(l'=[x::y'] \wedge Fil(y,f,y',n))) \wedge (f \bullet x = \mathsf{F} \supset Fil(y,f,l',n))$$
$$A(lxyng\mathsf{T}) \stackrel{\text{def}}{=} E(n)(g) \wedge C(lxyn) \wedge f \bullet x = \mathsf{T}$$
$$A(lxyng\mathsf{F}) \stackrel{\text{def}}{=} E(n)(g) \wedge C(lxyn) \wedge f \bullet x = \mathsf{F}$$

Note $G(fxl'ny)$ is the part of $Fil(l,f,l',n{+}1)$ when the list is nonempty.

1. $C(lxyn) \vdash l :_{l'} l' = [x::y] \wedge Len(y,n)$	(Var)
2. $\mathsf{F} \vdash [\varepsilon] :_m E(n+1)(m)$	(falsity)
3. $\mathsf{T} \vdash f(x) :_b f \bullet x = b$	(Var, Var, App)
4. $A(lxyng\mathsf{T}) \vdash g(y) :_m f \bullet x = \mathsf{T} \wedge Fil(y,f,m,n)$	(Var, Var, App)
5. $A(lxyng\mathsf{T}) \vdash [x::g(y)] :_m f \bullet x \wedge \exists y'.(m=[x::y'] \wedge Fil(y,f,y',n))$	(Cons)
6. $A(lxyng\mathsf{T}) \vdash [x::g(y)] :_m G(fxmyn)$	(Conseq)
7. $A(lxyng\mathsf{F}) \vdash g(y) :_m f \bullet x = \mathsf{F} \wedge Fil(l,f,m,n)$	(Var, Var, App)
8. $A(lxyng\mathsf{F}) \vdash g(y) :_m G(fxmyn)$	(Conseq)
9. $E(n)(g) \wedge C(lxyn) \vdash N :_m G(fxmyn)$	(3,6,8,If)
10. $E(n)(g) \wedge C(lxyn) \vdash M :_m G(fxmyn)$	(1,2,9,List)
11. $E(n)(g) \vdash \lambda l.M :_m \forall lxy.\ (C(lxyn) \supset G(fxmyn))$	(10, Abs)
12. $E(n)(g) \vdash \lambda l.M :_m E(n+1)(m)$	(Conseq)

In the final line, we observe: $\forall lxy.\ (C(lxyn) \supset G(fxmyn)) \supset \forall l.\ (Len(l,n{+}1) \supset Fil(l,f,m \bullet l,n{+}1)) \stackrel{\text{def}}{=} E(n+1)(m)$. We can

$$[Object]\ \frac{A^{\cdot\vec{y}_i\vec{j}_i} \wedge E_i^{\vec{y}_i} \vdash M_i :_{m_i} B_i \quad (\forall i\in I) \qquad \wedge_i \forall \vec{y}_i \vec{j}_i.(E_i \wedge B_i[u.l_i(\vec{y}_i)/m_i]) \supset B}{A \vdash \{l_i(\vec{y}_i) : M_i\}_{i\in I} :_u B}$$

$$[Inv]\ \frac{A \vdash M :_m B \quad A \vdash N_i :_{y_i} B_i \qquad \wedge_i B_i \supset B[u.l(\vec{y}_i)/m]}{A \vdash M.l(\vec{N}) :_u B}$$

Figure 4: Proof Rules for Objects and Invocations

now combine the two conclusions, reaching (25).

1.	$\mathsf{T} \vdash \lambda l.M :_m E(0)(m)$	
2.	$E(n)(g) \vdash \lambda l.M :_m E(n+1)(m)$	
3.	$\mathsf{T} \vdash \mu g.\lambda l.M :_m \forall n.E(n)(m)$	(Rec)

We have now arrived at the assertion (25).

Abstract Data Type. The final example is a short inference for packing and unpacking, using the counter ADT in (22). We consider programs written in a more convenient, object-like notation. cntr is defined by $\mathtt{pack}\,\langle\mathbb{N},\ M\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha$ and M is given as:

$$\begin{array}{rcl} M & \stackrel{\text{def}}{=} & \langle\mathbb{N}, \{\mathtt{new}() : 0,\ \ \mathtt{inc}(x) : x+1,\ \ \mathtt{get}(x) : x\}\rangle \\ \alpha & \stackrel{\text{def}}{=} & \{\mathtt{new} : \varepsilon \Rightarrow \mathrm{X},\ \ \mathtt{inc} : \mathrm{X} \Rightarrow \mathrm{X},\ \ \mathtt{get} : \mathrm{X} \Rightarrow \mathbb{N}\}. \end{array}$$

(note the empty vector ε has replaced what was () of Unit-type before). The target program is given as follows, writing $e.\mathtt{l_i}(e')$ for $(e.\mathtt{l_i}) \bullet (e')$:

$$N \stackrel{\text{def}}{=} \mathtt{unpack\ cntr\ as}\ \langle\mathrm{X}, x\rangle\ \mathtt{in}\ \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}()))$$

Observe $\vdash N : \mathtt{Unit} \Rightarrow \mathbb{N}$. The assertion we wish to prove is:

$$\mathsf{T} \vdash N :_u u \bullet () \geq 1. \tag{27}$$

We can reason for the above program using the rules for record and field selection in Section 4, Page . However the reasoning becomes much simpler if we use the proof rules which directly correspond to the above notation.[1] The proof rules for objects and object invocations are given in Figure 4. In (Object), we assume methods are indexed by $i \in I$, where each method in an object would take a distinct vector of typed arguments. Note the rule is a simple variant of the abstraction rule (and, dually, (Inv) the application rule).

For the derivation of (27), we use the following abbreviations.

$$\begin{array}{rcl} C(u,\mathrm{X}) & \stackrel{\text{def}}{=} & \exists w^{\alpha}, y^{\mathrm{X}}, z^{\mathrm{X}}.(u = \langle\mathrm{X}, w\rangle \wedge D(w,y,z)) \\ D(w,y,z) & \stackrel{\text{def}}{=} & w.\mathtt{new}() = y\ \wedge\ w.\mathtt{inc}(y) = z\ \wedge\ w.\mathtt{get}(z) \geq 1 \end{array}$$

Also let $C(\langle\mathrm{X},x\rangle,\mathrm{X}) \stackrel{\text{def}}{=} C(u,\mathrm{X})[\langle\mathrm{X},x\rangle/u]$. The derivation follows.

[1]In general, using the proof rules which precisely correspond to the granularity of a given programming language is essential for tractable reasoning. See [21, §8] for the corresponding derivation for the same assertion using the rules for record and field selection.

1.	$\mathsf{T} \vdash M :_m\ m.\mathtt{new}() = 0\ \wedge\ m.\mathtt{inc}(0) = 1\ \wedge\ m.\mathtt{get}(1) = 1$	(Num, Var, Add, Var, Object)
2.	$\mathsf{T} \vdash M :_m \exists y^{\mathbb{N}}.z^{\mathbb{N}}.(\langle\mathbb{N},m\rangle = \langle\mathbb{N},m\rangle\ \wedge\ D(m,y,z))$	(Conseq)
3.	$\mathsf{T} \vdash \mathtt{pack}\,\langle\mathbb{N},M\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha :_m\ \exists\mathrm{X}.C(m,\mathrm{X})$	(T-Pack)
4.	$C(\langle\mathrm{X},x\rangle,\mathrm{X}) \vdash x.\mathtt{new}() :_m\ x.\mathtt{new}() = m$	(Inv)
5.	$C(\langle\mathrm{X},x\rangle,\mathrm{X}) \vdash x.\mathtt{inc}(x.\mathtt{new}()) :_m$ $\exists y.\,(x.\mathtt{new}() = y \wedge x.\mathtt{inc}(y) = m)$	(Var, 4, Inv)
6.	$C(\langle\mathrm{X},x\rangle,\mathrm{X}) \vdash x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}())) :_m\ m \geq 1$	(Var, 5, Inv)
7.	$C(\langle\mathrm{X},x\rangle,\mathrm{X}) \vdash \lambda().x.\mathtt{get}(x.\mathtt{inc}(x.\mathtt{new}())) :_u\ u \bullet () \geq 1$	(Abs)
8.	$\mathsf{T} \vdash N :_u\ u \bullet () \geq 1$	(3,7,T-Unpack)

¿From Line 2 to Line 3, we apply the following logical equivalence as the condition for [*T-Pack*].

$$C(m,\mathrm{X})[\langle\mathrm{X},x\rangle/u][\mathbb{N}/\mathrm{X}] \equiv \exists y^{\mathbb{N}}.z^{\mathbb{N}}.(\langle\mathbb{N},m\rangle = \langle\mathbb{N},m\rangle\ \wedge\ D(m,y,z))$$

6. Conclusion

In this paper we proposed an assertional method for specifying and reasoning about polymorphic higher-order functions and data types, together with associated compositional proof rules, and demonstrated their usage using non-trivial reasoning examples for polymorphic programs. A compositional program logic has a fundamental status in many fields of software engineering, ranging from the traditional specification and verification, to model checking, to program testing, to static and dynamic analyses of programs, and to certification of mobile code. While being extensively studied for imperative programs, a clean treatment of higher-order computation in the compositional program logics, either for functional programming languages or procedural/object-oriented ones, may not have been known so far. We believe the presented framework is one of the promising directions towards this goal.

Although not discussed in this paper for the space sake, the presented method is extensible to diverse forms of typed behaviours, including behaviours with global and local state, call-by-name evaluation, user-defined data types and stateful objects, as well as to different notions of correctness including partial correctness. Further, in the total logics, valid logical judgements for a given program precisely characterise its semantics up to the canonical congruence. These results are discussed in [21]. The use of typed processes as the underlying semantic domain also enables a uniform treatment of proof rules and calculation of validity.

The studies on compositional logics for higher-order computation along the line of the present study have however just begun, and many significant challenges remain before we can reach a comprehensive theory of logical articulation of complex software behaviours, on the one hand, and a general and effective engineering framework, on the other. Some of the notable challenges are: development of compositional logics for a fully fledged functional programming language, such as ML [27] and Haskell [17], as well as those for object-oriented languages (this would involve treatment of input/output, exceptions and concurrency); development of practical notations for assertions as well as for proofs; non-trivial applications of the developed logics, including their use in integrated development environments, certified mobile code, and formally founded version control; and the applications of the derived program logics beyond standard verification/specification methodologies. More theoretically, a significant challenge is to obtain an in-depth understanding of the models of the presented logic and its ramifications, which may also lead to tractable methods for calculating validity in the logic besides that of number-theoretic facts.

Acknowledgement. The authors thank anonymous referees for their useful suggestions. Kohei Honda is partially supported by EPSRC grant GR/S55545. Nobuko Yoshida is partially supported by EPSRC grants GR/R33465 and GR/S55538.

References

[1] Abadi, M., $\top\top$-closed relations and admissibility, MSCS 10(3) (June 2000), 313-320.

[2] Abadi, M., Cardelli, L., Curien, P.-L., Formal Parametric Polymorphism, *TCS* 121, 1-2, (Dec), 9-58. Elsevier, 1993.

[3] Abadi, M. and Leino, R. A logic for object-oriented programs. Technical Report SRC-161, Compaq SRC, 1998.

[4] Abramsky, S., Domain Theory in Logical Form, Annuals of Pure and Applied Logic, 51:1-77, 1991.

[5] Barbanera, F., Dezani-Ciancaglini, M. and de'Liguoro, U., Intersection and Union Types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.

[6] Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, *TLCA01*, LNCS 2044, 29–45, Springer, 2001.

[7] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, *FoSSaCs'03*, LNCS 2620, 103–119, Springer, 2003.

[8] Bornat, R., Proving pointer programs in Hoare Logic. *Mathematics and Program Construction*, 2000.

[9] Caml Home Page, http://caml.inria.fr/.

[10] Calcagno, C., O'Hearn, P. and Bornat, R. Program logic and equivalence in the presence of garbage collection. *TCS*, 298(3):557-581, 2003.

[11] Coppo, M. and Dezani-Ciancaglini, M. An extension of basic functionality theory of lambda-calculus. *Notre Dame J. Formal Log.* 21:685–693, 1980.

[12] Damiani,F., Dezani-Ciancaglini, M. and Giannini, P. A filter model for mobile processes. *MSCS*, 9(1):63-101, 1999.

[13] Dunfield, J. and Pfenning, F., Type Assignment for Intersections and Unions in Call-by-Value, *FoSSaCs'03*, LNCS 2620, 250-266, Springer, 2003.

[14] Gordon, M., Milner, A. and Wadsworth, C., Edinburgh LCF, LNCS 78, Springer, 1979.

[15] Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[16] Floyd, W. Assigning meaning to programs. *Prc. Symp. in Applied Mathematics*. 19:19-32, 1967.

[17] The Haskell home page, http://haskell.org.

[18] Hennessy, M. and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32:1, pp.137-161, 1985.

[19] Hoare, C.A.R. An axiomatic basis of computer programming. *CACM*, 12:576-580, 1969.

[20] Honda, K., *Sequential Process Logics: Soundness Proofs*. Typescript, 50pp. November 2003. Available at: www.dcs.qmul.ac.uk/~kohei/logics.

[21] Honda, K., *Process Logic and Duality: Part (1) Sequential Processes*. Typescript, 234pp, March 2004. Available at: www.dcs.qmul.ac.uk/~kohei/logics.

[22] Honda, K. *From Process Logic to Program Logic*. A short version of Sections 1–3 and 6 in [21]. *ICFP'04*, ACM, 2004.

[23] Java home page. Sun Microsystems Inc., http://www.javasoft.com/, 1995.

[24] U. de' Liguoro, Characterizing convergent terms in object calculi via intersection types, *TLCA'01*, LNCS 2044. Springer, 2001.

[25] Mendelson, E., Introduction to Mathematical Logic (third edition). Wadsworth Inc., 1987.

[26] Milner, R., Functions as Processes, *MSCS.* 2(2):119–141, 1992,

[27] Milner, R., Tofte, M. and Harper, R.W, The Definition of Standard ML, MIT Press, 1990.

[28] Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, *Info. & Comp.* 100(1):1–77, 1992.

[29] Microsoft Corporation, .NET Framework Developer's Guide, `http://msdn.microsoft.com`, 2003.

[30] O'Hearn, P., Yang, H. and Reynolds, J., Separation and Information Hiding, POPL'04, 268–280, 2004.

[31] Pierce, B.C., *Types and Programming Languages*, MIT Press, 2002.

[32] Pitts, A.M., Existential Types: Logical Relations and Operational Equivalence, Proceedings ICALP'98, LNCS 1443, 309–326, Springer, 1998.

[33] Pitts, A.M., Parametric Polymorphism and Operational Equivalence, Mathematical Structures in Computer Science, 2000, 10:321–359.

[34] Plotkin, G. and Abadi, M., A Logic for Parametric Polymorphism, *LICS'98*, 42–53, IEEE Press, 1998.

[35] Reynolds, J. Idealized Algol and its specification logic. *Tools and Notions for Program Construction*, 121–161, CUP.

[36] Reynolds, J. Intuitionistic Reasoning about Shared Mutable Data Structure, *Millennial Perspectives in Computer Science*, 2000, Palgrave.

[37] Reynolds, J., Separation logic: a logic for shared mutable data structures. Invited Paper, *LICS* 2002.

[38] van Bakel, S., Intersection Type Assignment Systems. *TCS*, 151(2):385-435, 1995.

[39] Wadler, P. *Theorems for Free*. Functional Programming and Computer Architecture, Adison Wesley, 1989.

[40] Winskel, G. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[41] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. A full version in Journal of *Info. & Comp.*, 191 (2004) 145–202, Elsevier.