# A Compositional Program Logic for Imperative Higher-Order Functions

Kohei Honda[1], Nobuko Yoshida[2], and Martin Berger[1]

[1] Department of Computer Science, Queen Mary, University of London
[2] Department of Computing, Imperial College London

**Abstract.** Imperative higher-order functions have been one of the main topics in the semantics of types and programming languages for decades. They are not only a cornerstone of functional programming languages, but also serves as a central analytical tool for procedural, object-oriented and even low-level languages. In spite of this prominent status, compositional program logics for imperative extensions of the $\lambda$-calculus have not been known so far. This paper proposes a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic and our preceding work on program logics for pure higher-order functions. A central feature of the logic is its systematic use of names and operations on them. This allows precise and general description of complex higher-order imperative behaviours involving, for example, stored higher-order procedures. Valid logical descriptions faithfully correspond to the observational congruence of the language. The proof rules of the logic can cleanly embed and justify the standard proof rules for total correctness of Hoare logic with recursive procedure. The use of the logic is demonstrated through non-trivial reasoning examples which have been hard to assert and infer in the existing program logics.

## 1 Introduction

Imperative higher-order functions have been one of the main topics in the study on semantics and types of programming languages for decades. They are a cornerstone of richly typed functional programming languages such as ML [29] and Haskell [3] and central to the analysis of procedural, object-oriented and even low-level languages [1, 12, 30, 33, 37]. In spite of their prominent status, compositional program logics for imperative extensions of the $\lambda$-calculus have not been known so far [6, 9].

This paper proposes a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic [17] and our preceding work on program logics for pure higher-order functions [19, 21]. The program logic can precisely specify behaviours of imperative higher-order procedures as logical assertions up to the observational congruence, as well as enabling us to formally derive these assertions using proof rules which follow the syntactic structure of given programs. As far as we know, this is the first time a compositional assertional method for specifying and proving imperative higher-order functions has been developed. This is made possible by the central feature of the logic, a systematic use of names and operations on them (whose origin lines in the logics for the $\pi$-calculus [16, 20, 28]), allowing general and flexible description of complex higher-order imperative behaviours. The logic uses Floyd-Hoare's fundamental observation [10, 17] on logical description of sequential programs, the significance of the initial and final states of imperative variables for describing state change induced by execution of programs.

A syntactically simple extension to the Floyd-Hoare tradition is that assertions in the present logic not only talk about first-order data stored in imperative variables, as in Hoare's logic and its extensions, but also about arbitrary higher-order procedural behaviours, which may be fed as arguments, denoted by functional variables and stored in imperative variables. Having programs' behaviour as part of the universe of discourse is essential for reasoning about practical programs since functionalities of a higher-order program often crucially depend on the combined behaviour of the programs it uses. As an example, consider `twice` $\stackrel{\text{def}}{=} \lambda f^{\alpha\Rightarrow\alpha}.\lambda x^{\alpha}.(f(fx))$, with $\alpha$ possibly a higher-order type. The behaviour of this program depends on the potentially complex interplay between $f$ and $x$, all the more so if they induce side effects. We may also recall that storing higher-order procedures in imperative variables and using them is a fundamental idiom in system-level programming, which is one of the reasons why imperative $\lambda$-calculi are used for analysing low-level programs [12, 30, 37].

Let us present two simple, but non-trivial programming examples. A clean and rigorous description of their behaviour by a logical means is set to be a technical challenge in our present inquiry. We use the notations from the standard textbooks [13, 33].

$$\texttt{closureFact} \stackrel{\text{def}}{=} \mu f^{\mathsf{Nat}\Rightarrow\mathsf{Unit}}.\lambda x^{\mathsf{Nat}}.\ \texttt{if}\ x=0\ \texttt{then}\ y:=\lambda().1 \quad \texttt{else}\ \ y:=\lambda().\,(f(x-1)\,;\,x\times(!y)()\,)$$

Above $()$ is the unique constant of type $\mathsf{Unit}$ and $\lambda().N$ denotes $\lambda z^{\mathsf{Unit}}.N$ with $z$ fresh. When invoked as e.g. `closureFact` 3, the program stores a procedure in the imperative variable $y$. If we further invoke this stored procedure as $(!y)()$, then `closureFact` is called again with the argument $3-1=2$, after which a program stored in $y$ *at that time* is invoked, so that the multiple of $x$ and the value returned by that program is calculated and is given as the final return value. The intension of the program is that this final value should be the factorial of 3. The observable behaviour of `closureFact` can be informally described as: *When the program is fed with a number n, then it stores in y a closure which, when invoked with* $()$*, will return the factorial of n.* Note that, inside the body of `closureFact`, a free variable $f$ and the content of an imperative variable $y$ are used non-trivially. In particular, the correctness of this program crucially depends on how $y$ is updated sequentially in an orderly manner.

As another example, let us consider the following program.

$$\texttt{scheduler} \stackrel{\text{def}}{=} \texttt{map}\ (\lambda y^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.\,(\pi_1(y)\pi_2(y)))$$

where `map` is the standard higher-order map function, i.e.

$$\texttt{map} \stackrel{\text{def}}{=} \lambda f^{X\Rightarrow Y}.\mu m^{\mathsf{List}(X)\Rightarrow\mathsf{List}(Y)}.\lambda l^{\mathsf{List}(X)}.\,\texttt{case}(l)\ \texttt{of}\ \mathsf{Nil}\Rightarrow\mathsf{Nil}\ []\ x{+}{+}y\Rightarrow(fx){+}{+}(my)$$

where $x{+}{+}y$ is the list whose head is $x$ and whose tail is $y$. The program `scheduler` receives a list of jobs, where each job is a pair of a function and its argument, and executes these functions with corresponding arguments sequentially. Assuming each function may have side effects, what would be the specification of the scheduler, parameterised by properties of stored programs, and how can we derive it from the program text? A possible informal description would be: *Given a list of jobs* $\langle f_1,x_1\rangle,\langle f_2,x_2\rangle,..,\langle f_{n-1},x_{n-1}\rangle$ *(*$n\geq 1$*), if applying* $f_1$ *to* $x_1$ *changes the state* $\sigma_1$ *to the state* $\sigma_2$*, applying* $f_2$ *to* $x_2$ *changes the state* $\sigma_2$ *to the state* $\sigma_3$*, and so on, and finally applying* $f_{n-1}$ *to* $x_{n-1}$ *changes* $\sigma_{n-1}$ *to* $\sigma_{n-1}$*, then feeding the scheduler with this job list starting from* $\sigma_1$ *will eventually reach the state* $\sigma_n$. Compositional reasoning about such a program should treat higher-order functions, recursion, closures and products to derive an intended assertion from a program text. The proposed logic offers a simple language to specify such complex behaviour with precision as assertions, combined with mechanical proof rules for deriving predicates compositionally. Our logic's semantic precision is attested to by the fact that valid assertions can characterise typed programs' behaviours up to the canonical observational congruence.

In the remainder, Section 2 illustrates central ideas of the logic informally. Section 3 introduces the target language, the logic and its semantics. Section 4 presents compositional proof rules for the logic. Section 5 concisely summarises the semantics of the logic. Section 6 explores the use of the logic through reasoning examples, including an embedding of Hoare logic for total correctness (offering a clean analysis of the standard proof rules for, among others, assignment, while loops and recursive procedures), as well as an illustration of how we can treat two programming examples presented above. Section 7 discusses related work and further topics. The full version [2] presents detailed definitions, more examples and all missing proofs.

## 2 Preview

This section illustrates the key ideas of assertions for imperative higher-order functions, starting from a brief review of the logic for pure functions presented in [19, 21].

**Pure Higher-Order Functions.** In the present approach to program logics, behaviour is located at, and reasoned via, names. Consider a simple program which computes a doubling function, $N \stackrel{\text{def}}{=} \lambda x^{\mathsf{Nat}}.x+x$ where $\mathsf{Nat}$ is the type of natural numbers. If we apply 5 to $N$, 10 is returned. More generally, the result of applying any natural number to $N$ is always even. To represent these behavioural properties using logical formulae, we do not mention $N$ itself in a formula, but rather describe its properties by *naming* it as, say, $f$. Thus we can write $f \bullet 5 = 10$ as a property of $N$, named as $f$. Similarly we can write

$$\forall x^{\mathsf{Nat}}.Even(f \bullet x) \tag{2.1}$$

where $Even(n)$ is the predicate saying $n$ is even (e.g. $Even(x) \stackrel{\text{def}}{=} \exists n.(x = 2 \times n)$). The operator $\bullet$ is left associative and non-commutative, and may be understood as an analogue of application in applicative structures. Formulae may be combined using all the standard logical connectives and quantifiers, just as in Hoare logic.

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

$$\{C\}M :_f \{C'\}$$

which can be read as: if $M$, *named as* $f$, can *rely* on $C$ as the behaviour of an environment, then the program combined with the environment can *guarantee* $C'$. The name $f$ is called *anchor*. It can be any fresh name not occurring in $M$ and $C$. An anchor is used to represent $M$'s point of operation, hence of specification. As an example, the specification for $N$ is:

$$\{\mathsf{T}\}N :_f \{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} \tag{2.2}$$

which says that the program $N$ named as $f$, under the trivial assumption $\mathsf{T}$ on its free variables, satisfies $\forall x^{\mathsf{Nat}}.Even(f \bullet x)$. By having names in assertions, we can compositionally derive a specification which involves a non-trivial assumption on higher-order variables based on a simple operation: when the function $f$ is applied to an argument, the result $f \bullet x$ is *peeled-off* and replaced by a new anchor name $u$. For example, we can derive

$$\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\}f3 :_u \{Even(u)\}$$

from two small specifications (1) $\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} f :_m \{\forall x^{\mathsf{Nat}}.Even(m \bullet x)\}$ (an instance of the axiom for variable: $f$ named as $m$ satisfies the same predicate as the one assumed for $f$, replacing $f$ by $m$); and (2) $\{\mathsf{T}\} 3 :_m m = 3$ ("3" named as $m$ satisfies a predicate $m = 3$) and by combining them together as $(\forall x^{\mathsf{Nat}}.Even(f \bullet x) \wedge m = 3) \supset Even(f \bullet m)$ (which is a simple instance of $(A(x,x) \wedge x = y) \supset A(x,y)$, the standard axiom in predicate logic with equality [25, §2.8]). The same framework works for higher-order programs where multiple higher-order variables share assumptions.

$$\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} f3 + f(f5) + 1 :_u \{Odd(u)\} \tag{2.3}$$

where $Odd(n)$ says $n$ is odd. Now by just combining two specifications $N$ in (2.2) and $L \stackrel{\text{def}}{=} f3 + f(f5) + 1$ in (2.3), we can derive:

$$\{\mathsf{T}\} \mathtt{let}\ f = N\ \mathtt{in}\ L :_u \{Odd(u)\} \tag{2.4}$$

where $\mathtt{let}\ f = N\ \mathtt{in}\ L$ is encoded as $(\lambda f.L)N$. The property which is guaranteed by $N$ is simply plugged into the assumption for $L$. This derivation is similar to a composition rule of Hoare logic, where we infer $\{C\}P_1;P_2\{C'\}$ from $\{C\}P_1\{C_1\}$ and $\{C_1\}P_2\{C\}$.

**Mutable Higher-Order Functions.** The idea of naming behaviours is naturally extended to stateful computation. A typical example is the following specification of a program that reads the number 7 from a global storage cell $x$ and then returns 9.

$$\{!x = 7\}\ 2+!x :_u \{u = 9 \wedge !x = 7\}$$

where the logical term $!x$ represents dereferencing $x$ [29]. The resulting behaviour is located at the anchor name $u$. The assertion says: the program $2+\,!x$ returns 9 whenever $x$ initially stores 7, and it does not change this content of $x$. As another example, this time with side-effects, we can derive:

$$\{!x = 3\}\ x := (2+!x)\ ;\ !x :_u \{u = 5 \wedge !x = 5\}$$

where ";" is sequential composition (encodable into call-by-value application).

We now move to assertions describing more complex behaviour where functions cause side-effects during evaluation. Let us consider $W \stackrel{\text{def}}{=} \lambda x.(w := (1+!w)\,; x + x)$, slightly modified from $\lambda x.x + x$ in the previous paragraph. Recalling $L$ from (2.3), we define $M \stackrel{\text{def}}{=} \mathtt{let}\ f = W\ \mathtt{in}\ L$. Now this function does not only satisfy $Odd(u)$, but also *changes a memory cell* $w$ when invoked. Hence we would expect the following:

$$\{!w = 0\}\ \mathtt{let}\ f = W\ \mathtt{in}\ L :_u\ \{Odd(u) \wedge !w = 3\} \tag{2.5}$$

How can we specify the behaviour of $W$ to reach (2.5)? A simple method is to attach pre and post-conditions to invocations of functions by an argument, and assert them as a single predicate. Thus we write:

$$\{C\}\ f \bullet x \searrow u\, \{C'\}$$

This assertion reads: if the state of memory and the environment satisfy $C$, the invocation of $f$ with an argument $x$ yields a value named $u$ and a final state, together satisfying $C'$. "$\searrow$" indicates the evaluation of $f \bullet x$ resulting in $u$, which is asymmetric unlike the equality $e = e'$. This is due to the non-reversibility of state change. Based on this idea, $W$ named as $f$ has the following specification:

$$EvenS(w, f) \stackrel{\text{def}}{=} \forall x.\forall i.\{!w = i\}\ f \bullet x \searrow u\ \{Even(u) \wedge !w = i+1\}$$

$EvenS(w, f)$ specifies a procedure which, when invoked, would not only increment $w$ but also return an even number: if function $f$ is used when $!w = 0$, then $f$'s return value is even and $!w = 1$. In fact from $EvenS(w, f)$ we can derive:

$$\forall x.\{!w = 0\}\, f \bullet x \searrow u\, \{Even(u) \wedge !w = 1\}$$

using standard axioms for universal quantification. Now assume $f$ satisfies the above specification. Then we can derive the following judgement.

$$\{EvenS(w, f) \wedge !w = 0\}\, f\, 5 :_v\, \{Even(v) \wedge !w = 1\}$$

The key idea here is that when the function (named $f$) is applied to the argument 3, not only is the result $u$ replaced by a new anchor $v$, but we also *split* the assumption ($EvenS(w, f)$) in two pieces, its pre-condition $C$ placed in the pre-condition of the judgement (together with the assumption itself) and its post-condition $C'$ in the post-condition of the judgement. Repeating this, we can derive (2.5) in a compositional way, using essentially the same `let`-rule as for stateless functions.

When working with higher-order functions, assertions with pre/post-conditions can be nested in an arbitrary way and may appear in the pre/post-conditions of other assertions. For example, let $V \stackrel{\text{def}}{=} \lambda y.(!x)y$. If $x$ stores a function with side effects, like $W$ above, then calling $V$ may involve writing to memory. Thus we may derive:

$$\{\mathsf{T}\}\lambda y.(!x)y :_u \{\ \{EvenS(w, !x) \wedge !w = 0\}u \bullet n \searrow z\{Even(z) \wedge EvenS(w, !x) \wedge !w = 1\}\ \}$$

which says: if $V$ is applied to a natural number $n$ under the condition that $x$ stores a function which satisfies $EvenS(w, u)$, and $w$ initially stores 0, then resulting term evaluates to an even number, with $w$'s final state being 1.

A merit of the present approach in comparison with conventional Hoare logic is that our's can directly assert formulae about the combined behaviour of (possibly higher-order) procedures. Consider the following program:

$$M \quad \stackrel{\text{def}}{=} \quad \lambda x^{\mathsf{Nat}}.\,(y := x\ ;\ g(f)\ ;\ g(f)\ ;\ !y + 1) \tag{2.6}$$

where $f$ is of type $\mathsf{Unit} \Rightarrow \mathsf{Unit}$ and $g$ of $(\mathsf{Unit} \Rightarrow \mathsf{Unit}) \Rightarrow \mathsf{Unit}$. Assume we *only* know the abstract property of $f$ and $g$ which says: if we apply $f$ to $g$, then the content of $y$ changes its parity (i.e. if it is initially even then it becomes odd and vice versa). The present logic can formally describe this property as follows (omitting return values).

$$A(fg) \stackrel{\text{def}}{=} \{Odd(!y)\}\, f \bullet g\, \{Even(!y)\}\ \wedge\ \{Even(!y)\}\, f \bullet g\, \{Odd(!y)\}$$

Then a property of $M$ under these assumptions may be asserted as:

$$\{A(fg)\}\ M :_u\ \{\forall x^{\mathsf{Nat}}.\{Even(x)\}u \bullet x \searrow z\{Odd(z) \wedge Even(!y)\}\} \tag{2.7}$$

which says: under the assumption about $f$ and $g$ as given, if the argument is even, then the result is odd and the content of $y$ is even. Let the above post-condition be written $Even\_then\_Odd(u, y)$. From the same pre-condition, we can also infer the dual property $Odd\_then\_Even(u, y) \stackrel{\text{def}}{=} \forall x^{\mathsf{Nat}}.\{Odd(x)\}u \bullet x \searrow z\{Even(z) \wedge Odd(!y)\}$ or even a conjunction of the two, $Even\_then\_Odd(u, y) \wedge Odd\_then\_Even(u, y)$, as its post-condition. This specification relies on the property of the combined behaviour of $f$ and $g$ and demonstrates the immediate benefit of having named higher-order procedures and specifications of their behaviour as an integral part of assertions: we can transparently specify and reason about the complex interplay among two or more procedures which may call each other and which as a whole demonstrate a specific behaviour of interest. Further examples will be treated in Section 6, after formally introducing the logic and proof rules in the following sections.

## 3 Logic for Imperative Call-by-Value PCF

### 3.1 Imperative PCF

Below we briefly review the programming language we use, the call-by-value PCF with unit, sums and products, augmented with imperative variables (henceforth also called *references*). The grammar of programs is standard [33], given below. Assume given an infinite set of *variables* ($x, y, z, \ldots$, also called *names*).

$$
\begin{array}{ll}
\text{(value)} & V, W ::= \mathbf{c} \;\mid\; x \;\mid\; \lambda x^{\alpha}.M \;\mid\; \mu f^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M \;\mid\; \langle V, W \rangle \;\mid\; \mathtt{in}_i(V) \\
\text{(program)} & M, N ::= V \mid MN \mid x := N \mid\, !M \mid \mathtt{op}(\vec{M}) \mid \pi_i(M) \mid \langle M, N \rangle \;\mid\; \mathtt{in}_i(M) \\
& \qquad \mid\; \mathtt{if}\, M\, \mathtt{then}\, M_1\, \mathtt{else}\, M_2 \;\mid\; \mathtt{case}\, M\, \mathtt{of}\, \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}}
\end{array}
$$

The grammar uses types ($\alpha, \beta, \ldots$), which are given later. Constants are ranged over by $\mathbf{c}, \mathbf{c}', \ldots$. Examples include the unit $()$, natural numbers and booleans. $\mathsf{op}(\vec{M})$ (where $\vec{M}$ is a vector of programs) is a standard $n$-ary arithmetic or boolean operation, such as $+$, $-$, $\times$, $=$ (equality of two numbers/booleans), $\neg$ (negation), $\wedge$ and $\vee$. $!M$ dereferences $M$ while $x := N$ is assignment. All these constructs are standard, cf. [13, 33].

The dynamics of this language is given by the call-by-value reduction relation, using store [13, 33]. A *store* ($\sigma, \sigma', \ldots$) is a finite map from imperative variables to values. We write $\sigma[x \mapsto V]$ for the store which maps $x$ to $V$ and otherwise agrees with $\sigma$. The call-by-value reduction, written $(M, \sigma) \longrightarrow (M', \sigma')$, is standard [13, 33]. We only list the rules for assignment and dereference. Below $\sigma(x)$ and $\sigma[x \mapsto V]$ indicate $x \in \mathsf{dom}(\sigma)$.

$$
(!x,\ \sigma) \;\rightarrow\; (\sigma(x),\ \sigma) \qquad (x := V,\ \sigma) \;\rightarrow\; ((),\ \sigma[x \mapsto V])
$$

We also write $(M, \sigma) \Downarrow (V, \sigma')$ for $(M, \sigma) \rightarrow^* (V, \sigma')$, and $M \Downarrow V$ for $(M, \emptyset) \rightarrow^* (V, \emptyset)$.

The grammar of types is standard [13, 33], summarised below for reference.

$$
\alpha, \beta ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \qquad \rho ::= \alpha \mid \mathsf{Ref}(\alpha)
$$

We call $\alpha, \beta, \ldots$ *value types*, and $\mathsf{Ref}(\alpha), \ldots$ *reference types*. Reference types are restricted to carrying only non-reference types. We can easily take off this restriction with a syntactically minor extension to the present logic (see Section 6). Note a reference can still carry arbitrary higher-order procedures. A *basis* is a finite map from names to types. $\Gamma, \Gamma' \ldots$ range over bases whose codomains are value types, while $\Delta, \Delta', \ldots$ range over bases whose codomains are reference types. $\mathsf{dom}(\Gamma)$ (resp. $\mathsf{dom}(\Delta)$) denotes the domain of $\Gamma$ (resp. of $\Delta$). The typing rules are standard [33] and omitted. We write $\Gamma; \Delta \vdash M : \alpha$ when $M$ has type $\alpha$ under $\Gamma$ and $\Delta$, always assuming $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$.

### 3.2 Terms, Formulae and Judgement

**Terms and Formulae.** The logical language is that of first-order logic with equality [25, § 2.8] together with an assertion for the evaluation of stateful expressions. The grammar of terms and formulae follows.

$$
\begin{array}{lcl}
e & ::= & x^{\rho} \;\mid\; () \;\mid\; \mathbf{n} \;\mid\; \mathbf{b} \;\mid\; \mathbf{op}(\vec{e}) \;\mid\; \langle e, e' \rangle \;\mid\; \pi_i(e) \;\mid\; \mathsf{inj}_i^{\alpha+\beta}(e) \;\mid\; !e \\
C & ::= & e = e' \mid \neg C \mid C \wedge C' \mid C \vee C' \mid C \supset C' \mid \forall x^{\alpha}.C \mid \exists x^{\alpha}.C \mid \{C\}\, e \bullet e' \searrow x\, \{C'\}
\end{array}
$$

The first set of expressions (ranged over by $e, e', \ldots$) are *terms* while the second set are *formulae* (ranged over by $A, B, C, C' \ldots$). Terms, which are from [19, 21] except $!x$, include all the constants and first-order operations (including multiplication) of the target programming language. In the first line of the grammar of terms, we have an explicit paring, projection and injection operation. The final term, $!e$, denotes the dereference of $e$. $\mathsf{fv}(e)$ denotes the free variables occurring in $e$.

The predicate $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ is called *evaluation formula*, where the name $x$ binds its free occurrences in $C'$. Intuitively, $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ asserts that an invocation of $e$ with an argument $e'$ under the initial state $C$ terminates with a final state and a resulting value, named as $u$, both described by $C'$. $\bullet$ is non-commutative.

Terms and formulae are typed starting from type-annotated variables. Two names in a formula should have the same type, similarly for a pair of equated terms. $!(x^{\rho})$ is typed as $\alpha$ iff $\rho = \mathsf{Ref}(\alpha)$. In $\{C\}\ e_1 \bullet e_2 \searrow z\ \{C'\}$, $e_1$, $e_2$ and $z$ are typed as $\alpha \Rightarrow \beta$, $\alpha$ and $\beta$, respectively. The remaining well-typedness conditions are naturally given [2]. A boolean typed term is also used as a formula (though we do need boolean typed terms since they are used as, e.g. arguments of evaluation formulae). Hereafter we only consider well-typed terms and formulae and often omit type annotations. We shall write $\Theta \vdash C$ if $C$ is well-typed with its free names typed following $\Theta$, where $\Theta, \Theta', \ldots$ range over finite maps which may combine two kinds of bases.

**Convention 1.** $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$ (the logical equivalence of $C_1$ and $C_2$). We use truth $\mathsf{T}$ (definable as $1 = 1$) and falsity $\mathsf{F}$ (which is $\neg\mathsf{T}$). The standard binding convention is always assumed. $\mathsf{fv}(C)$ denotes the set of *free variables* in $C$. $\{C\}\ e_1 \bullet e_2 \searrow e'\ \{C'\}$ with $e'$ not a variable, stands for $\{C\}\ e_1 \bullet e_2 \searrow x\ \{x = e' \wedge C'\}$ with $x$ fresh; and $\{C\}\ e_1 \bullet e_2\ \{C'\}$ for $\{C\}\ e_1 \bullet e_2 \searrow ()\ \{C'\}$. Formulae are often called *assertions*.

Some small examples: $y = 6$ is an assertion which says $y$ is equal to 6; $!y = 6$ says the content of a memory cell $y$ is equal to 6. $C \stackrel{\text{def}}{=} \forall i, n.\ \{!w = n\}\, !x \bullet i \searrow 2 \times i\, \{!w = n+1\}$ says that a variable $x$ stores a function which, when invoked, would increment $w$, returning the double of the argument. This predicate is satisfied when, for example, $f(w) \stackrel{\text{def}}{=} \lambda z.(w := !w+1; z \times 2)$ is stored in $x$. $D \stackrel{\text{def}}{=} \{C \wedge !w = 0\} u \bullet 3 \searrow 6 \{C \wedge !w = 1\}$ says that, if $u$ is invoked with 3 in a state satisfying $!w = 0$ as well as $C$, then the returned value is 6 and the final state is $!w = 1$. This is satisfied by $\lambda y.(!x)y$ named as $u$, with $x$ storing $f(w)$ above.

**Judgement.** Following Hoare [17], a judgement in the present program logic consists of a pair of formulae and a program, which takes the following shape.

$$\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$$

This sequent is used for both validity and provability. If we wish to be specific, we prefix it with either $\vdash$ (for provability) or $\models$ (for validity). In $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, we assume $\Gamma;\Delta \vdash M : \alpha$. $u$ is called the *anchor* of the judgement and should not be in $\mathsf{dom}(\Gamma,\Delta) \cup \mathsf{fv}(C)$. The formula $C$ is the *pre-condition*; and $C'$ is the *post-condition*. We say $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *well-typed* if (1) $\Gamma;\Delta \vdash M : \alpha$; and (2) $\Gamma, \Delta, \Theta \vdash C$ and $u : \alpha, \Gamma, \Delta, \Theta \vdash C'$ for some $\Theta$ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma,\Delta) \cup \{u\}) = \emptyset$. The same condition applies to judgements on provability. Then the *primary names* in this well-formed judgement are $\mathsf{dom}(\Gamma,\Delta) \cup \{u\}$. The *auxiliary names* in this judgement are those free names in $C$ and $C'$ which are not primary (for example, in "$\{x = i\} 2 \times x :_u \{u = 2 \times i\}$", $x$ and $u$ are primary while $i$ is auxiliary; $u$ is in addition its anchor). We often omit the typing of a program from a judgement, writing $\{C\}\ M :_u \{C'\}$.

Intuitively, $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ says that: *if $M$ is closed by values satisfying $C$ (for non-reference variables), and is evaluated starting from a store satisfying $C$ (for reference variables), then it terminates so that the final state and the resulting value, named $u$, together satisfy $C'$.* This informal reading will be made precise in Section 5.

### 3.3 Axioms and Rules for Validity

In the proof rules presented in the next section, one often needs to calculate validity of assertions. Formally, a formula $C$ is *valid* if it is true under arbitrary models (the class of models we consider will be discussed in Section 5). Practically, we can often calculate validity syntactically. As is standard [17], we shall freely use axioms, rules and theorems from propositional calculus, first-order logic with equality [25, §2.8], and number theory (peano arithmetic). There are also natural axioms for data types, such as $C[()/x^{\mathsf{Unit}}] \equiv C$, $e = \mathbf{t} \equiv e$, $e = \mathbf{f} \equiv \neg e$, $\mathtt{in}_i(e) = \mathtt{in}_j(e') \supset i = j \wedge e = e'$ etc. Further, the following axioms for evaluation formulae are often useful.

**Convention 2.** (1) A formula is *stateless* if, in the formula, a name of a reference type occurs only inside the pre/post conditions of evaluation formulae. $A, B, \ldots$ range over stateless formulae. (2) $C^{\text{-}\vec{x}}$ is $C$ in which no name from $\vec{x}$ freely occurs.

$$
\begin{array}{lrcl}
\text{(e1)} & \{C_1\}\, e_1 \bullet e_2 \searrow z\, \{C\} \;\wedge\; \{C_2\}\, e_1 \bullet e_2 \searrow z\, \{C\} & \equiv & \{C_1 \vee C_2\}\; e_1 \bullet e_2 \searrow z\; \{C\} \\
\text{(e2)} & \{C\}\, e_1 \bullet e_2 \searrow z\, \{C_1\} \;\wedge\; \{C\}\, e_1 \bullet e_2 \searrow z\, \{C_2\} & \equiv & \{C\}\; e_1 \bullet e_2 \searrow z\; \{C_1 \wedge C_2\} \\
\text{(e3)} & \{\exists x^{\alpha}.C\}\; e_1 \bullet e_2 \searrow z\; \{C'^{\text{-}x}\} & \equiv & \forall x^{\alpha}.\{C\}\; e_1 \bullet e_2 \searrow z\; \{C'\} \\
\text{(e4)} & \{C^{\text{-}x}\}\; e_1 \bullet e_2 \searrow z\; \{\forall x^{\alpha}.C'\} & \equiv & \forall x^{\alpha}.\{C\}\; e_1 \bullet e_2 \searrow z\; \{C'\} \\
\text{(e5)} & \{A \wedge C\}\; e_1 \bullet e_2 \searrow z\; \{C'\} & \equiv & A \supset \{C\}\; e_1 \bullet e_2 \searrow z\; \{C'\} \\
\text{(e6)} & \{C\}\; e_1 \bullet e_2 \searrow z\; \{A^{\text{-}z} \supset C'\} & \supset & A \supset \{C\}\; e_1 \bullet e_2 \searrow z\; \{C'\} \\
\text{(e7)} & \{C\}\, e_1 \bullet e_2 \searrow z\, \{C'\} & \supset & \{C \wedge A\}\, e_1 \bullet e_2 \searrow z\, \{C' \wedge A\} \\
\text{(e8)} & (C \supset C_0) \wedge \{C_0\}\, e_1 \bullet e_2 \searrow z\, \{C_0'\} \wedge (C_0' \supset C') & \supset & \{C\}\; e_1 \bullet e_2 \searrow z\; \{C'\}
\end{array}
$$

## 4 Proof Rules

The proof rules of the logic are given in Figure 1. In each rule, we follow Conventions 1 and 2, and assume all occurring judgements are well-typed, and no primary names in the premise(s) occur as auxiliary names in the conclusion. When we derive a judgement for a program using these proof rules, the derivation precisely follows the structure of the program, except for the intermittent use of structural rules (in which only formulae are manipulated). Below we illustrate key aspects of these rules.

**[*Var, Const*]** say that, if we wish to assert $C$ about a datum named $u$, we should assume the same property, with the datum substituted for $u$.

**[*Add*]** is the rule for the addition operator, which assumes the left-to-right evaluation order, indicating both the state change induced by evaluation and the resulting values. Similarly for other first-order operators.

**[*Abs*]** says: if we know, under the assumptions $A$ (which does not talk about an argument $x$ to be abstracted), whenever $M$ starts evaluating from $C$, it returns a result $m$ and reaches a state which together satisfy $C'$, then we can guarantee $\lambda x.M$ named as $u$ satisfies the same property under the same assumption $A$, with the property now presented as an evaluation formula replacing $M$ with a call to $u$, $u \bullet x$, with the result $m$.

**[*App*]** says: if we know $M$ reaches $C_0$ starting from $C$, and $N$ reaches $C_1$ starting from $C_0$, and, moreover, we know putting them together and applying them reaches $C'$ starting from $C_1$, then $MN$ reaches $C'$ starting from $u$.

**Fig. 1** Proof Rules

$$[\mathit{Var}]\ \frac{-}{\{C[x/u]\}\ x :_u \{C\}} \qquad [\mathit{Const}]\ \frac{-}{\{C[\mathbf{c}/u]\}\ \mathtt{c} :_u \{C\}}$$

$$[\mathit{Add}]\ \frac{\{C\}\,M_1 :_{m_1} \{C_0\} \quad \{C_0\}\,M_2 :_{m_2} \{C'[m_1+m_2/u]\}}{\{C\}\,M_1+M_2 :_u \{C'\}}$$

$$[\mathit{Abs}]\ \frac{\begin{array}{c}\{C \wedge A^{\text{-}x}\}\ M :_m \{C'\}\\ \forall x.\{C\}\,u\bullet x \searrow m\{C'\} \supset B\end{array}}{\{A\}\ \lambda x.M :_u \{B\}} \qquad [\mathit{App}]\ \frac{\begin{array}{c}\{C\}\ M :_m \{C_0\}\\ \{C_0\}\ N :_n \{\ C_1 \wedge \ \{C_1\}\ m\bullet n \searrow u\ \{C'\}\}\end{array}}{\{C\}\ MN :_u \{C'\}}$$

$$[\mathit{If}]\ \frac{\{C\}\ M :_b \{C_0\} \quad \{C_0[\mathbf{t}/b]\}\ M_1 :_u \{C'\} \quad \{C_0[\mathbf{f}/b]\}\ M_2 :_u \{C'\}}{\{C\}\ \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u \{C'\}}$$

$$[\mathit{In}_1]\ \frac{\{C\}\ M :_v \{C'[\mathtt{in}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}} \qquad [\mathit{Case}]\ \frac{\{C^{\text{-}\vec{x}}\}\ M :_m \{C_0^{\text{-}\vec{x}}\} \quad \{C_0[\mathtt{in}_i(x_i)/m]\}\ M_i :_u \{C'^{\,\text{-}\vec{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[\mathit{Pair}]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1,m_2\rangle/u]\}}{\{C\}\ \langle M_1,M_2\rangle :_u \{C'\}} \qquad [\mathit{Proj}_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

$$[\mathit{Deref}]\ \frac{-}{\{C[!x/u]\}\ !x :_u \{C\}} \qquad [\mathit{Assign}]\ \frac{\{C\}\ M :_m \{C'[m/\,!x][()/u]\}}{\{C\}\ x := M :_u \{C'\}}$$

$$[\mathit{Rec}]\ \frac{\{A^{\text{-}x} \wedge \forall j \precneqq i.B(j)[x/u]\}\ \lambda y.M :_u \{B(i)\}}{\{A\}\ \mu x.\lambda y.M :_u \{\forall i.B(i)\}}$$

$$[\mathit{Promote}]\ \frac{\{A\}\ V :_u \{B\}}{\{A \wedge C\}\ V :_u \{B \wedge C\}} \qquad [\mathit{Consequence}]\ \frac{C \supset C_0 \quad \{C_0\}\ M :_u \{C_0'\} \quad C_0' \supset C'}{\{C\}\ M :_u \{C'\}}$$

**[*If, In*$_1$*, Case, Pair, Proj*$_1$]** are natural rules for standard data types, similar to **[*Add*]**.

**[*Deref*]** is understood as **[*Var, Const*]**. If we wish to have $C$ for $!x$ (as a program) named $u$, then we should assume the same thing about the content of $x$, substituting $!x$ for $u$.

**[*Assign*]** uses two substitutions $C'[m/!x][()/u]$. The notation $[m/!x]$ stands for replacing all occurrences of $!x$ by $m$, while $[()/u]$ is the standard substitution of $()$ for a unique anchor $u$. The first substitution $C'[m/!x]$ says the result of the assignment $x := M$ is turning what is stated about $m$ in $C'[m/!x]$ into the property of $!x$. The second substitution $[()/u]$ says, in effect, the assignment command terminates (because $()$ is the unique value of type Unit).

**[*Rec*]** is the proof rule for the total correctness of recursion (slightly strengthening the rule in [19, 21]). It is based on mathematical induction, though by choosing an appropriate domain and a well-ordering on (part of) the domain, we can extend the rule to well-founded induction. We later show how this rule can cleanly and precisely encode known proof rules for total correctness of the while loop and recursive procedures.

**[*Promote*]** extends the stateless pre/post-conditions to general ones by conjunction. The rule is sound because a value does not (immediately) cause state change.

**[*Consequence*]** The rule follows the standard consequence rule in Hoare logic. Checking the validity of two entailments is in general intractable, even though in practice one can often appeal to syntactic reasoning, cf. §3.3. Kleymann's powerful extension of the consequence rule [22] may as well be used (for its precise formulation in the present setting, as well as for other structural rules, see [2, §4]).

## 5 Models and Soundness

This section gives a concise summary of models for assertions and judgements. Their operational nature has the merit of immediate faithfulness to programs' behaviours (e.g. a predicate claiming the existence of unrealisable functions is unsatisfiable); extensibility (e.g. polymorphisms); and conciseness. Different models are possible: [20] constructs a more uniform universe of models from typed processes.

The semantics centres on programs which do not own free non-reference variables.

**Definition 3.** (semi-closed programs [26]) $\Gamma;\Delta \vdash M : \alpha$ is *semi-closed* (resp. *closed*) when $\mathsf{dom}(\Gamma) = \emptyset$ (resp. $\mathsf{dom}(\Gamma) = \Delta = \emptyset$), often written $\Delta \vdash M : \alpha$ (resp. $\vdash M : \alpha$).

Let $\cong$ be the standard observational congruence for the imperative PCFv [13], based on convergence to semi-closed values. An *abstract value of type* $\Delta;\alpha$ is a $\cong$-congruence class of semi-closed values typed $\alpha$ under $\Delta$. We write $[V]^{\Delta;\alpha}$ for an abstract value represented by $\Delta \vdash V : \alpha$. A model is defined using abstract values.

**Definition 4.** (models) A *model of type* $\Gamma;\Delta$ is a pair $(\xi,\sigma)$ such that $\xi$ is a finite map from $\mathsf{dom}(\Gamma)$ to abstract values such that each $x \in \mathsf{dom}(\Gamma)$ is mapped to an abstract value typed as $[V]^{\Delta;\Gamma(x)}$; and $\sigma$ is a finite map from $\mathsf{dom}(\Delta)$ to abstract values such that each $x \in \mathsf{dom}(\Delta)$ is mapped to an abstract value typed as $[V]^{\Delta;\alpha}$ with $\Delta(x) = \mathsf{Ref}(\alpha)$. We let $\mathcal{M},\ldots$ range over models.

We write $\Gamma;\Delta \vdash \mathcal{M}$ when $\mathcal{M}$ is a model of type $\Gamma;\Delta$. Intuitively, $\xi$ and $\sigma$ in $(\xi,\sigma)$ respectively denote a standard functional environment and a store, taken modulo $\cong$.

Assume given a formula $C$ and a model $\mathcal{M}$, both typed under $\Gamma;\Delta$. Then each term in $C$ is inductively interpreted under $\mathcal{M}$ as an abstract value in the obvious way, except each name of a reference type is interpreted as that name itself (so that, in effect, we treat reference-typed names as constants). As examples, given $\mathcal{M} = (\xi,\sigma)$, a functional variable $x^{\alpha}$ is interpreted as $\xi(x)$; dereferencing $!y$ is interpreted as $\sigma(y)$; and a pair $\langle e,e'\rangle$ is interpreted as a pair of abstract values interpreting $e$ and $e'$ (the full definition is found in [2, 21, §4.3]). We write $[\![e]\!]\mathcal{M}$ for the interpretation of $e$ under $\mathcal{M}$.

The satisfaction relation is defined using the component-wise union of two disjoint models, one interpreting primary names and another auxiliary names. Writing $\mathcal{I},\mathcal{I}',\ldots$ for models used for interpreting auxiliary names, the satisfaction relation is written:

$$\mathcal{M}^{\Gamma;\Delta} \models^{\mathcal{I}} C$$

which reads: *under $\mathcal{I}$, $C$ is satisfied by $\mathcal{M}$*. The satisfaction relation is defined following the standard first-order logic with equality [25, §2.8] with the equality predicate interpreted as the identity relation, adding the following clause for evaluation formulae. Below let $\mathcal{M} = (\xi,\sigma_0)$. $\Downarrow$ is defined from that of concrete programs.

$$\begin{aligned}\mathcal{M}^{\Gamma;\Delta} \models^{\mathcal{I}} \{C\}e_1 \bullet e_2 \searrow x\{C'\} \ \text{ if } \forall\sigma.\ (\ & \Delta \vdash \sigma \wedge (\xi,\sigma) \models^{\mathcal{I}} C \\ & \supset \exists V,\sigma'.\ ([\![e_1]\!]\mathcal{M}\cdot\mathcal{I}[\![e_2]\!]\mathcal{M}\cdot\mathcal{I},\ \sigma) \Downarrow ([V]^{\Delta;\beta},\sigma') \\ & \quad \text{such that } (\xi \cup x{:}[V]^{\Delta;\beta},\sigma') \models^{\mathcal{I}} C'\ )\end{aligned}$$

The left-hand side says: if the interpretation of $e_1$ is invoked with that of $e_2$ as an argument, then for any state $\sigma$ satisfying $C$, the invocation starting from $\sigma$ will converge with a value named $x$ and a state $\sigma'$ together satisfying $C'$.

We are now ready to formalise the semantics of judgements. Below $M\xi$ denotes the substitution of values following $\xi$, confusing abstract values and concrete values.

**Definition 5.** (semantics of judgement) $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ iff, for each $I$, $\xi$ and $\sigma$, whenever $(\xi, \sigma) \models^I C$, we have $(M\xi, \sigma) \Downarrow (V, \sigma')$ such that $(\xi \cdot u : [V]^{\Delta;\alpha}, \sigma') \models^I C'$.

Description of programs by valid judgements is extensional in the following sense.

**Proposition 6.** *If* $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ *and* $\Gamma;\Delta \vdash M \cong N : \alpha$*, then* $\models \{C\}\, N^{\Gamma;\Delta;\alpha} :_u \{C'\}$.

The proof is direct from Definition 5. We conclude this section with the main theorem of the paper, proved in [2, §5].

**Theorem 7.** (soundness) *If* $\vdash \{C\}\; M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ *then* $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$.

# 6 Reasoning Examples

## 6.1 Deriving Hoare Logic for Total Correctness

We first embed the standard proof rules of Hoare logic for total correctness with recursive procedures [22] in the present logic. The main interest in the embedding is because it offers a lucid analysis of the known proof rules for basic imperative idioms, notably the proof rules for assignment, loops and recursions. The syntax of programs is given as follows. Let $p, q, \ldots$ range over procedure labels.

$$
\begin{array}{rcl}
e & ::= & x \mid \mathbf{c} \mid\ !x \mid \mathbf{op}(e_1, \ldots, e_n) \\
P, Q, .. & ::= & \mathtt{skip} \mid x := e \mid P; Q \mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid \mathtt{while}\ e\ \mathtt{do}\ P \\
& \mid & \mathtt{call}\ p \mid \mathtt{proc}\ p = P\ \mathtt{in}\ Q
\end{array}
$$

In $\mathtt{proc}\ p = P\ \mathtt{in}\ Q$, a procedure body $P$ is named $p$, where we allow calls to $p$ to occur in $P$. The reduction rules are standard and are omitted.

We consider a logic for total correctness. Formulae, ranged over by $C, C', \ldots$, are a proper subset of the logic for the imperative PCF, having only natural numbers as data types and missing evaluation formulae. The judgement takes the shape $\Sigma \vdash \{C\}\, P\, \{C'\}$, where $\{C\}\, P\, \{C'\}$ is the standard Hoare triple and $\Sigma$ is a finite map from procedural labels to triples, each taking the form $\{C\}p\{C'\}$. Figure 2 presents the standard proof rules [22]. Solely for brevity we use a single recursion in [*RecPro*] and mathematical induction in [*While*] and [*RecPro*]. The encoding of programs into PCF-terms is standard (procedure labels are simply taken to be variables).

$$
\begin{array}{l}
[\![\mathtt{skip}]\!] \stackrel{\text{def}}{=} () \qquad [\![x := e]\!] \stackrel{\text{def}}{=} x := e \qquad [\![P; Q]\!] \stackrel{\text{def}}{=} [\![P]\!]; [\![Q]\!] \\
[\![\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \stackrel{\text{def}}{=} \mathtt{if}\ e\ \mathtt{then}\ [\![P]\!]\ \mathtt{else}\ [\![Q]\!] \\
[\![\mathtt{while}\ e\ \mathtt{do}\ P]\!] \stackrel{\text{def}}{=} (\mu w.\lambda().\mathtt{if}\ e\ \mathtt{then}\ [\![P]\!]; (w\,())\ \mathtt{else}\ ())\,() \\
[\![\mathtt{call}\ p]\!] \stackrel{\text{def}}{=} p() \qquad [\![\mathtt{proc}\ p = P\ \mathtt{in}\ Q]\!] \stackrel{\text{def}}{=} (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])
\end{array}
$$

Note all commands have the unit type. When $M$ has a unit type in $\{C\}M :_u \{C'\}$, we can safely omit the anchor because $C[()/u] \equiv C$. Hence, just like a Hoare triple, hereafter we often write $\{C\}M\{C'\}$. The judgement $\Sigma \vdash \{C\}P\{C'\}$ is translated as

$$
\{[\![\Sigma]\!] \wedge C\}[\![P]\!]\{C'\} \qquad \text{with} \qquad [\![\emptyset]\!] \stackrel{\text{def}}{=} \mathsf{T} \text{ and } [\![\Sigma, \{C\}p\{C'\}]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!] \wedge \{C\}\, p \bullet ()\{C'\}
$$

**Fig. 2** Hoare Logic with Recursive Procedure (total correctness)

$$[\mathit{Skip}]\frac{-}{\Sigma\vdash\{C\}\mathtt{skip}\{C\}}\quad[\mathit{AssignH}]\frac{-}{\Sigma\vdash\{C[e/u]\}x:=e\{C\}}\quad[\mathit{Seq}]\frac{\Sigma\vdash\{C\}P\{C_0\}\quad\Sigma\vdash\{C_0\}Q\{C'\}}{\Sigma\vdash\{C\}P;Q\{C'\}}$$

$$[\mathit{IfH}]\frac{\Sigma\vdash\{C\wedge e\}P_1\{C'\}\quad\Sigma\vdash\{C\wedge\neg e\}P_2\{C'\}}{\Sigma\vdash\{C\}\,\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2\,\{C'\}}\quad[\mathit{While}]\frac{C\wedge e\supset e'\gneq 0\quad \Sigma\vdash\{C\wedge e\wedge e'=n\}P\{C\wedge e'\lneq n\}}{\Sigma\vdash\{C\}\mathtt{while}\ e\ \mathtt{do}\ P\{C\wedge\neg e\}}$$

$$[\mathit{Call}]\frac{\{C\}p\{C'\}\in\Sigma}{\Sigma\vdash\{C\}\,\mathsf{call}\,p\,\{C'\}}\quad[\mathit{RecProc}]\frac{\Sigma,\{\exists j\lneq i.C(j)\}p\{C_0\}\vdash\{C(i)\}P\{C_0\}\quad \Sigma,\{\exists i.C(i)\}p\{C_0\}\vdash\{C\}Q\{C'\}}{\Sigma\vdash\{C\}\mathtt{proc}\ p=P\ \mathtt{in}\ Q\{C'\}}$$

Below we show two interesting cases, [*AssignH*] and [*RecProc*]. For [*AssignH*], we use the following rule derivable from the rules in Figure 1 (for a derivation see [2]).

$$[\mathit{Simple}]\frac{-}{\{C[e/u]\}e:_u\{C\}}$$

Then the standard assignment rule, [*AssignH*], is cleanly decomposed into [*Simple*] and [*Assign*] of Figure 1 (we omit $\Sigma$ which plays no role; by definition we set $u\notin\mathsf{fv}(C)$).

| | | |
|---|---|---|
| 1. | $\{C[m/!x][e/m]\,(\stackrel{\text{def}}{=}C[e/!x])\}\ e:_m\ \{C[m/!x][()/u]\}$ | (Simple) |
| 2. | $\{C[e/!x]\}\ x:=e:_u\ \{C\}$ | (Assign) |

For [*RecProc*], the main interest is how the recursion rule in the present logic, [*Rec*], offers a precise account of the induction principle for recursive procedures. Below (e3) etc. are axioms from §3.3, indicating their use in the consequence rule.

| | | |
|---|---|---|
| 1. | $\{[\![\Sigma]\!]\wedge\{\exists j\lneq i.C(j)\}p\bullet()\{C'\}\wedge C(i)\}\ [\![P]\!]\ \{C_0\}$ | (Assumption) |
| 2. | $\{[\![\Sigma]\!]\wedge\forall j^{\mathsf{Nat}}.\{j\lneq i\wedge C(j)\}\,p\bullet()\,\{C'\}\wedge C(i)\}\ [\![P]\!]\ \{C_0\}$ | (e3) |
| 3. | $\{[\![\Sigma]\!]\wedge\forall j^{\mathsf{Nat}}\lneq i.\{C(j)\}p\bullet()\{C'\}\wedge C(i)\}\ [\![P]\!]\ \{C_0\}$ | (e5) |
| 4. | $\{[\![\Sigma]\!]\wedge\forall j^{\mathsf{Nat}}\lneq i.\{C(j)\}p\bullet()\{C'\}\}\ \lambda().[\![P]\!]:_m\ \{\{C(i)\}m\bullet()\{C_0\}\}$ | (Abs) |
| 5. | $\{[\![\Sigma]\!]\}\,\mu p.\lambda().[\![P]\!]:_m\ \{\forall i.\{C(i)\}m\bullet()\{C_0\}\}$ | (Rec) |
| 6. | $\{[\![\Sigma]\!]\}\,\mu p.\lambda().[\![P]\!]:_m\ \{\{\exists i.C(i)\}m\bullet()\{C_0\}\}$ | (e3) |
| 7. | $\{[\![\Sigma]\!]\wedge\{\exists i.C(i)\}\,p\bullet()\,\{C_0\}\wedge C\}\ [\![Q]\!]\ \{C'\}$ | (Assumption) |
| 8. | $\{[\![\Sigma]\!]\}\,\lambda p.[\![Q]\!]:_n\ \{\forall p.\{\exists i.C(i)\}\,p\bullet()\,\{C_0\}\ \supset\ \{C\}n\bullet p\{C'\}\}$ | (Abs) |
| 9. | $\{[\![\Sigma]\!]\wedge C\}\ (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])\,\{C'\}$ | (6, 8, App) |

Lines 2 and 3 above use $\exists j\lneq i.C\stackrel{\text{def}}{=}\exists j.(j\lneq i\wedge C)$ and $\forall j\lneq i.C\stackrel{\text{def}}{=}\forall j.(j\lneq i\supset C)$.

For other rules, [*Skip*] and [*Call*] are immediate; [*Seq*] is derived from [*App*] and [*Abs*], using $C[()/u]\equiv C$, cf. §3.3; [*IfH*] is from [*Simple*] and [*If*]; [*While*] again uses [*Rec*]. See [2] for derivations of these rules. Thus the following holds up to the correspondence in validity.

**Proposition 8.** (embedding of Hoare logic for total correctness) $\Sigma\vdash\{C\}P\{C'\}$ *implies* $\{[\![\Sigma]\!]\wedge\{C\}\}[\![P]\!]\{C'\}$.

### 6.2 Simple Imperative Higher-Order Functions

We illustrate derivations of assertions using simple higher-order imperative programs. We start from first-order imperative commands, using [*AssignH*] of Figure 2 in § 6.1.

$$\begin{array}{lll} 1. & \{!x\times 2 = 6\}\ y :=\ !x \times 2\ \{!y = 6\} & \text{(AssignH)} \\ \hline 2. & \{!x = 3\}\ y :=\ !x \times 2\ \{!y = 6\} & (!x = 3 \supset !x \times 2 = 6,\ \text{Conseq}) \end{array}$$

If we wish to reason about assignment of a higher-order program, we can no longer use [*AssignH*]. Below we let $Double(u) \stackrel{\text{def}}{=} \forall i.(u \bullet i = i \times 2)$.

$$\begin{array}{lll} 1. & \{z = i\} z \times 2 :_m \{m = i \times 2\} & \text{(Var, Num, } \times) \\ \hline 2. & \{\mathsf{T}\} \lambda z.(z \times 2) :_v \{Double(v)\} & \text{(Abs)} \\ \hline 3. & \{\mathsf{T}\}\, x := \lambda z.(z \times 2)\, \{Double(!x)\} & \text{(Promote, Assign)} \end{array}$$

Next we infer a function with dereference (cf. Section 2).

$$\begin{array}{lll} 1. & \{Double(!x)\}\, !x :_m \{Double(m)\} & \text{(Deref)} \\ \hline 2. & \{y = 3\}\, y :_n \{n = 3\} & \text{(Var)} \\ \hline 3. & \{Double(!x) \wedge y = 3\}\, (!x)y :_u \{Double(!x) \wedge u = 6\} & \text{(App)} \\ \hline 4. & \{\mathsf{T}\}\, \lambda y.(!x)y :_u \{\{Double(!x)\}\, u \bullet 3 \searrow 6\, \{Double(!x)\}\} & \text{(Abs)} \\ \hline 5. & \{Double(!x)\}\, (\lambda y.(!x)y)3 :_u \{u = 6 \wedge Double(!x)\} & \text{(App)} \end{array}$$

Next we use the following slightly enriched version of [*Seq*] of Figure 2 in § 6.1.

$$[Seq'] \ \frac{\{C\} M \{C_0\} \quad \{C_0\} N :_u \{C'\}}{\{C\} M; N :_u \{C'\}}$$

Using [*Seq'*], we can plug-in the post and pre-conditions of the conclusions of the last two derivations.

$$\{\mathsf{T}\}\ x := \lambda z.(z \times 2)\, ;\, (\lambda y.(!x)y)3 :_u \{u = 6 \wedge Double(!x)\}$$

By a similar reasoning, we obtain (using $C$ in § 3.2, page 7, for the post-condition):

$$\{\mathsf{T}\}\, x := \lambda z.(w :=!w + 1; z \times 2) :_u \{\forall i, n. \{!w = n\}\, !x \bullet i \searrow 2 \times i\, \{!w = n + 1\}\}$$

Combining this and $\{\mathsf{T}\}\lambda y.(!x)y :_u \{\{C \wedge\ !w = 0\}\, u \bullet 3 \searrow 6\, \{C \wedge\ !w = 1\}\}$ (which uses $D$ in § 3.2) by [*Seq'*] gives us:

$$\{C \wedge !w = 0\}\, x := \lambda z.(w :=!w + 1; z \times 2)\, ;\, (\lambda y.(!x)y)3 :_u \{u = 6 \wedge C \wedge\ !w = 1\}$$

Finally we reason for $M$ in (2.6), Section 2, Page 5, using $A(fg)$ given there.

$$\begin{array}{lll} 1. & \{A(fg) \wedge Even(x)\}\ y := x\ \{A(fg) \wedge Even(!y)\} & \text{(AssignH, Conseq)} \\ \hline 2. & \{A(fg) \wedge Even(!y)\}\ g(f)\ \{A(fg) \wedge Odd(!y)\} & \text{(Var, Var, App)} \\ \hline 3. & \{A(fg) \wedge Odd(!y)\}\ g(f)\ \{A(fg) \wedge Even(!y)\} & \text{(Var, Var, App)} \\ \hline 4. & \{A(fg) \wedge Even(!y)\}\ !y + 1 :_z \{Odd(z) \wedge Even(!y)\} & \text{(Simple, Conseq)} \\ \hline 5. & \{A(fg)\}\ M :_u\ \{Even\_then\_Odd(u, x)\} & \text{(Seq', Seq', Seq', Abs)} \end{array}$$

We can further derive $\{A(fg)\}M :_u \{Even\_then\_Odd(u,x) \wedge Odd\_then\_Even(u,x)\}$ through the structure rule for conjunction corresponding to (e2) in §3.3.

### 6.3 Two Programming Examples Revisited

Now we come back to some the examples from the introduction. First, the specification for `closureFact` is now easy to make precise, for instance with the following judgement.

$$\{\mathsf{T}\}\,\texttt{closureFact} :_u \{\forall i^{\mathsf{Nat}}.\{\mathsf{T}\}\,u \bullet i\,\{\,\{\mathsf{T}\}\,!y\bullet()\searrow z\{z = i!\}\,\}\}$$

This assertion concisely describes the program's observable behaviour and we invite the reader to compare it with its informal description in the introduction. The proof that `closureFact` satisfies this specification is straightforward from the syntax of the program and hence omitted.

Next is an assertion describing basic aspects of `scheduler`'s behaviour (we use an assertion $C(i)$ with a free variable $i$ to represent a sequence of states, assuming list operations like ++ and Nil as abbreviations of the assertion language).

$$\begin{aligned}\mathsf{Sched}(u) \stackrel{\text{def}}{=}\ & \{C_0\}u\bullet\mathsf{Nil}\{C_0\} \quad \wedge \quad \forall g^{\alpha\Rightarrow\mathsf{Unit}}, a^{\alpha}, y^{\mathsf{List}((\alpha\Rightarrow\mathsf{Unit})\times\alpha)}.\\ & (\{C(i+1)\}\,g\bullet a\,\{C(i)\} \wedge \{C(i)\}\,u\bullet y\,\{C(0)\} \supset \{C(i+1)\}\,u\bullet(\langle g,a\rangle{+}{+}y)\,\{C(0)\})\end{aligned}$$

The assertion is natural and concise. It says: given for example a list $l \stackrel{\text{def}}{=} [(f,a),(g,b)]$, if we can prove $\{C_0\}\,fa\,\{C_1\}$ and $\{C_1\}\,gb\,\{C_2\}$, then $\{C_0\}\,\texttt{scheduler}\ l\,\{C_2\}$ can be derived. Below we outline how the main judgement

$$\{\mathsf{T}\}\,\texttt{scheduler} :_u \{\mathsf{Sched}(u)\}, \tag{6.1}$$

can be inferred. Setting $\texttt{scheduler} \stackrel{\text{def}}{=} \texttt{map}\ \mathsf{app}$ where $\mathsf{app} \stackrel{\text{def}}{=} \lambda z^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.(\pi_1(z)\pi_2(z))$, we derive:

$$\{\mathsf{T}\}\,\texttt{map} :_m \{\forall f^{\beta\Rightarrow\mathsf{Unit}}\,\mathsf{Map}'(m,f)\} \qquad \{\mathsf{T}\}\,\mathsf{app} :_n \{\forall z^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.\ \mathsf{App}(n,z)\} \tag{6.2}$$

where:

$$\begin{aligned}\mathsf{Map}'(m,f) &\stackrel{\text{def}}{=} \{\mathsf{T}\}\,m\bullet f\searrow u\{\mathsf{Map}(u,f)\}\\ \mathsf{Map}(u,f) &\stackrel{\text{def}}{=} \{C_0\}u\bullet\mathsf{Nil}\{C_0\} \quad \wedge \quad \forall x^{\beta}, y^{\mathsf{List}(\beta)}.\\ &\qquad (\{C(i+1)\}\,f\bullet x\,\{C(i)\} \wedge \{C(i)\}\,u\bullet y\,\{C(0)\} \supset \{C(i+1)\}\,u\bullet(x{+}{+}y)\,\{C(0)\})\\ \mathsf{App}(n,z) &\stackrel{\text{def}}{=} (\{C(i+1)\}\,\pi_1(z)\bullet\pi_2(z)\,\{C(i)\} \supset \{C(i+1)\}\,n\bullet z\,\{C(i)\}).\end{aligned}$$

The derivation for `map` in (6.2) needs an inductive inference [21, § 5]. The inference of the judgement for `app` is straightforward. Deriving (6.1) from (6.2) is interesting and uses the following inferences on validity.

$$\begin{aligned}\forall f.\mathsf{Map}'(m,f) \wedge \forall z.\mathsf{App}(n,z) &\supset \mathsf{Map}'(m,n) \wedge \mathsf{App}(n,z) && (\forall\text{-inst})\\ &\supset \{\mathsf{T}\}\,m\bullet n\searrow u\{\mathsf{Map}(u,n) \wedge \mathsf{App}(n,z)\} && (\text{e7})\\ &\supset \{\mathsf{T}\}\,m\bullet n\searrow u\{\mathsf{Map}(u,\langle g,a\rangle) \wedge \mathsf{App}(n,\langle g,a\rangle)\} && (\forall\text{-inst, (e8)})\\ &\supset \{\mathsf{T}\}\,m\bullet n\searrow u\{\mathsf{Sched}(u)\} && (\text{modus ponens, (e8)})\end{aligned}$$

We can now deduce:

| | | |
|---|---|---|
| 1. | $\{\mathsf{T}\}\ \texttt{map} :_m \{\forall f.\mathsf{Map}'(m,f)\}$ | (6.2) |
| 2. | $\{\mathsf{T}\}\ \mathsf{app} :_n \{\forall z.\mathsf{App}(n,z)\}$ | (6.2) |
| 3. | $\{\forall f.\mathsf{Map}'(m,f)\}\ \mathsf{app} :_n \{\forall f.\mathsf{Map}'(m,f) \wedge \forall z.\mathsf{App}(n,z)\}$ | (Constancy) |
| 4. | $\{\forall f.\mathsf{Map}'(m,f)\}\ \mathsf{app} :_n \{\{\mathsf{T}\}\,m\bullet n\searrow u\{\mathsf{Sched}(u)\}\}$ | (Consequence) |
| 5. | $\{\forall f.\mathsf{Map}'(m,f)\}\ (\texttt{map}\ \mathsf{app}) :_u \{\mathsf{Sched}(u)\}$ | (1, 4, App) |

where (Constancy) in Line 3 is the structural rule corresponding to (e7) in § 3.3, allowing conjunction with a stateless formula in the pre/post-conditions. Line 4 uses the above inference on validity for the entailment. We have reached (6.1).

## 7 Discussion

### 7.1 Related Work

The logic and proof rules presented in this work is built on many preceding works on semantics and logics of programming languages. For typed and untyped $\lambda$-calculi, equational logics have been studied since the classical work by Church and Curry. Among many studies, LCF [11] augments the standard equational theories of the $\lambda$-calculus with Scott's fixpoint induction. In this tradition, we find program logics which can treat higher-order features and various data types elegantly, but only as long as they are confined to stateless computation. This is due to the difficulties inherent in applying syntactic equations based on reductions, such as $\beta\eta$-conversions, to stateful languages.

Compositional program logics for imperative languages have been studied extensively since Hoare's work (which in turn is partially based on earlier work by Floyd [10] and Naur [31]). The program logics studied in this tradition directly treat a basic feature of imperative programming languages, state change. However, higher-order procedures and data structures have not been fully captured in Hoare's logic and its extensions, in spite of their prominent status in both theories and practice. A central reason for the difficulty lies in that the basic notion of semantics of judgements in Hoare logics centres on a pair of first-order universes, is hard to adapt to higher-order behaviours, as partly addressed by Clarke's negative result [8].

Some of the central elements of the proposed program logic come from these two traditions. Direct treatment of operators for (higher-order) data types in assertions is inherited from logics for $\lambda$-calculi; whereas the use of pairs of assertions to describe state change is progeny of logics for imperative languages. The universe of discourse now contains higher-order behaviours themselves. This is essential for allowing general description of higher-order imperative programs, and is made possible by the systematic use of names in assertions, which comes from logics for the $\pi$-calculus [19, 20, 28]. Names have been used in Hoare Logic since an early work by Kowaltowski [23], and are found in the work by Von Oheimb [38], Leavens and Baker [24], Abadi and Leino [5], Bierman and Parkinson [7] and others, for treating parameter passing and return values. These authors however do not treat higher-order procedures, which are uniformly captured in the present logic along with parameters and return values through the use of names. Another merit of the proposed approach is its tight correspondence with semantics of the target programming language, as indicated by Proposition 6 in Section 5.

Reynolds, O'Hearn and others [32, 36] study extensions of Hoare logic in which non-classical logical connectives are used for reasoning about low-level operations such as garbage collection. Because of a difference in levels of abstraction (for example, Reynolds [36] treats references to memory cells as numerals), direct technical comparisons are difficult. Different aspects of low-level computation may be elucidated through a precise logical articulation of imperative higher-order functions in the present logic (cf. [12, 30, 37]), contributing to an integrated understanding of diverse programming constructs.

Dynamic Logic [15], introduced by Pratt [34] and studied by Harel and others [14], uses programs and predicates on them as part of formulae, facilitating detailed specifications of various properties of programs such as (non-)termination as well as intensional features. As far as we know, higher-order procedures have not been treated in Dynamic Logic, even though we believe part of the proposed method to treat higher-order functions would work consistently in their framework.

A logic for higher-order imperative procedures, called specification logic, is developed by Reynolds for Idealised Algol [35]. Apart from the difference of the shape of assertions and target languages, his logic [35] does not have compositional proof rules.

### 7.2 Extensions and Further Topics

The recursion rule given in Figure 1 is for total correctness, and is used for embedding and justifying the known proof rules for loop and recursive procedures that guarantee termination [14]. Another recursion rule, studied in [20], is for partial correctness and has the following shape.

$$[\mathit{Rec\text{-}Partial}]\,\frac{\{A \wedge B[x/u]\}\,\lambda y.M :_i \{B\} \quad B \text{ admissible at } u \quad A \supset \exists x.B}{\{A\}\,\mu x.\lambda y.M :_u \{B\}}$$

"Admissibility" in the above rule intuitively says that $B$ is about partial correctness (see [20, §5.3]). This rule can encode and justify the existing rules for loops and recursive procedures for partial correctness. There is a valid judgement which can be proved with [*Rec-Partial*] but not without.

For simplicity, we have omitted polymorphisms and recursive types in the present paper. Their integration is however straightforward following our previous study [21]. Also the same line of development is possible for the call-by-name version [20]. The type structure in § 3.1 is restricted so that a reference type is never carried by other types. Another extension is the introduction of local state. As is well-known, a simple proof rule suffices for local references which are used only during a run of a command. However ML-like languages allow extrusion of a local datum [27] beyond its original scope. These extensions are treated in sequels to the present paper.

The use of side-effect-free expressions on assignment has been one of the basic principles in compositional program logics. Freedom from side effects is however hard to maintain in the higher-order setting because of complex interplay between higher-order procedures. The clean embedding of Hoare's assignment rule in Section 6 may suggest the presented framework effectively extends the traditional approach while retaining its virtues. Experiment of the possible extensions in the context of an integrated verification framework such as JML [4] would be an interesting subject for further study.

Finally, the origin of the assertions and judgements introduced in the present work is the logic for typed $\pi$-calculi [19, 20], where linear types lead to a compositional process logic. The known precise embeddings of high-level languages into these typed $\pi$-calculi can be used to determine the shape of name-based logics like the one presented here for the embedded languages. Once found, they can be embedded back with precision into the originating process logics. [18–20] discuss process logics and their relationship to the program logics in detail.

## References


1. C– home page. http://www.cminusminus.org.
2. A full version of the present paper. Available at www.doc.ic.ac.uk/˜yoshida/logics.html.
3. The Haskell home page. http://haskell.org.
4. The Java Modeling Language (JML) home page. http://www.jmlspecs.org/.
5. Martín Abadi and Rustan Leino. A logic for object-oriented programs. In *Verification: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.
6. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
7. Gavin M. Bierman and Matthew J. Parkinson. Separation logic and abstractions. In *POPL'05*, 2005.

8. E Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. In *POPL'79*, pages 129–147, 1979.
9. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
10. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
11. Mike Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh LCF. LNCS 78, Springer Verlag, 78.
12. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.
13. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
14. David Harel. Proving the correctness of regular deterministic programs. *TCS*, 12(16), 1980.
15. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logigc*. The MIT Press, 2000.
16. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
17. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
18. Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, November 2003. Typescript, 50 pages.
19. Kohei Honda. From process logic to program logic. In *Proc. ICFP'04*. ACM Press, 2004. A short version of Sections 1–3 and 6 in [20].
20. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
21. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*. ACM Press, 2004.
22. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.
23. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
24. Gary Leavens and Alber L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM'99: World Congress on Formal Methods*. Springer, 1999.
25. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
26. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.
27. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
28. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
29. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
30. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
31. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
32. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
33. Benjamin C. Pierce. *Type Systems and Programming Languages*. MIT Press, 2002.
34. Vaughn R. Pratt. Six lectures on dynamic logic. In *Foundations of Computer Science III, Part 2*, number 109 in Mathematical Centre Tracts, pages 53–82, 1980.
35. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
36. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
37. Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
38. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.