Program Logic and Program Analysis

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. One of the central merits of compositional program logics [18] as a foundation of software engineering is their potential ability to describe a large class of properties of programs relative to a chosen notion of observability. One open issue in achieving such descriptive power has been the lack of a precise logical treatment of pragmatically significant language features such as higher-order procedures, expressions with side effects and shared mutable data structures.
This paper introduces a compositional program logic for a higher-order polymorphic imperative language with local state. A central feature of the logic is its systematic use of names and operations on them. Assertions in the logic describe all and only observable properties of programs in the sense that valid assertions for a program precisely characterise its behaviour up to the observational congruence. After illustrating the logic through examples, we demonstrate its descriptive power by presenting a logical justification of a type-based secure information flow analysis. The key idea is to translate typability in the secrecy typing to an assertion in the logic which directly describes non-interference. This has the following merits: (1) a precise logical description of the noninterference property; (2) a very short, mechanical correctness proof of the secrecy typing via soundness of the logic; and (3) generalisation and classification of secrecy analyses on a uniform logical basis.

Table of Contents

1 Introduction 3
2 Call-by-Value Polymorphic PCF with References 10
 2.1 Syntax and Reduction 10
 2.2 Types and Typing 11
3 Logic for Higher-Order Functions with Global State 14
 3.1 Terms and Formulae 14
 3.2 Judgement 17
 3.3 Models, Interpretation and Satisfaction 18
 3.4 Proof Rules 21
 3.5 Reasoning Examples 25
 3.6 Soundness with Global State 26
4 Incorporating Local State 33
 4.1 Syntax and Reduction 33
 4.2 Assertions for Shared Local State (1): Introduction 34
 4.3 Assertions for Shared Local State (2): Syntax and Typing 35
 4.4 Models with Local State 37
 4.5 Some Laws for Local State 39
 4.6 Proof Rules 40
 4.7 Examples of Reasoning with Local State 42
 4.8 Soundness with Local State and Further Results 45
5 Information Flow Analysis in Program Logic 48
 5.1 Total Correctness with Partiality 49
 5.2 Predicates for Noninterference 52
 5.3 Secrecy Typing and Noninterference 54
 5.4 Global State (1): Preparation 59
 5.5 Global State (2): Proof of Noninterference 62
 5.6 Local State (1): Preparation 63
 5.7 Local State (2): Proof of Noninterference 68
 5.8 Extensions and Refinement 69
6 Discussion 70
A Remaining Proofs for Theorem 34 76
B Satisfaction Relation without Opaque Types 79

1 Introduction

One of the promises of compositional program logics is to make programming and systems building *exact science* [18]. The reason why compositional program logics have the potential to become the core part of science of systems building is that they offer a means to describe essentially the whole class of the properties of programs relative to a chosen notion of observables [12, 18, 19]. Given an assertion language with general descriptive power, one can pinpoint any specific property of interest, in order to use the description as a basis of engineering activity, be it specification, refinement, verification, testing, or program analysis. One open issue in achieving such descriptive power has been the lack of a precise logical treatment of pragmatically significant language features such as higher-order procedures, expressions with side effects and shared mutable data structures.

Based on our previous work [22, 25], this paper introduces a compositional program logic for a prototypical higher-order polymorphic imperative language with global and local state. The language is simple, yet is expressive enough to represent major imperative and higher-order procedural idioms and data structures. On the basis of the type discipline of the language, the proposed logic extends specifiable properties to encompass essentially arbitrary observable properties of programs. A central feature of the logic is its systematic use of names and operations on them. Names are used for naming diverse entities, including higher-order functions and procedures, data structures, and pure and imperative variables. Dynamics and effects of these constructs are then captured by suitable operators on names. This use of names comes from π-calculus logic [22, 23, 36] and enables assertions to capture diverse language constructs on a uniform technical basis with flexibility and precision.

After illustrating the logic through examples, we demonstrate its descriptive power with a logical justification of a type-based secure information flow analysis. Static analyses of programming languages complement program logics by the former offering effectively calculable methods to guarantee safety of software; logics on the other hand offer a universal framework for description and verification, at the cost of the lack of automatic verifiability. This complementary nature suggests a possible merit in relating these two methodologies at a formal level. Among others the property to be guaranteed by an analysis may be described precisely as an assertion in the program logic. This will make explicit the safety policy underlying the program analysis; it may also offer a uniform formal basis to discuss correctness and other properties of the analysis based on the logical machinery. Further, the unifying representation of diverse analyses in a compositional program logic may lead to their integrated treatment in a single conceptual as well as implementational framework, firmly built on a common theoretical basis.

Information flow analysis is about preventing unsafe flow of information in a program (we may not want our income information held by the tax office leaked to a public web page). Not only is this relevant for privacy concerns, it is also one of the most subtle analyses, because of the need to precisely capture mutual effects among language constructs (due to the existence of covert channels [31]). The analysis and its justification become particularly intricate in the presence of higher-order features and side-effects. We shall show how advanced type-based secrecy analysis [3, 24, 47, 57] can be given a clean logical description and justification using the proposed program logic. The key idea is to translate typability in the secrecy typing to an assertion in the logic which directly describes non-interference. We show that precise logical description of the noninterference property leads to a very short correctness proof of a basic secrecy typing for three classes of behaviours, first monomorphic higher-order functions, second their extension with global state, and finally the extension with local state. We shall also discuss how the assertional descriptions and proofs can be extended to other classes of behaviours as well as different secrecy policies.

We now illustrate some of the key ideas of assertions for imperative higher-order computation, starting from a brief review of the logic for pure higher-order functions studied in our previous work [22, 25].

Pure Higher-Order Functions Consider a simple program which computes a doubling function, $N \stackrel{\text{def}}{=} \lambda x.x + x$, with type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ where Nat 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 Nat, then it returns (if ever) a value of type Nat; 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 [12, 18]. 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

$$f \bullet 5 = 10 \tag{1.1}$$

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

$$\forall x^{\mathsf{Nat}}.Even(f \bullet x) \tag{1.2}$$

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. This idea – naming functions and programs – naturally comes from encodings of functions into the π-calculus [35, 36] 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 [35] whose specifications are given via interrogating its behaviour one by one. As an example, we can write down the specification for N as:

$$\mathsf{T} \vdash N :_f \forall x^{\mathsf{Nat}}.Even(f \bullet x) \tag{1.3}$$

which says that the program N named as f, under the trivial assumption T on its free variables, satisfies $\forall x^{\mathsf{Nat}}.Even(f \bullet x)$ (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.

$$\mathsf{T} \vdash \lambda x^{\mathsf{Nat}}.x \ :_u \ \forall y^{\mathsf{Nat}}.\ u \bullet y = y. \tag{1.4}$$

The judgement says that the given program, when applied to any argument of type Nat, will return that same argument as a result.

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

$$\mathsf{T} \vdash (\lambda x^{\mathsf{Nat}}.x)2 \ :_m \ m = 2 \tag{1.5}$$

Let us take a brief look at how we can compositionally derive (1.5). We first need the predicate for the argument.

$$\vdash 2 :_z z = 2 \tag{1.6}$$

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

$$\frac{(1.4),\ \ (1.6),\ \ (\forall y^{\mathsf{Nat}}.\,(u \bullet y = y) \wedge z = 2) \supset u \bullet z = 2}{\mathsf{T} \vdash (\lambda x^{\mathsf{Nat}}.x)2 \ :_m \ m = 2}$$

In the above derivation (which follows the proof rule for application, formally discussed in Section 3 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. [33, §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.

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

Here, instead of calculating validity of entailment, we simply match the first Nat of the function's type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ with the argument's type Nat, obtaining Nat. 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.

$$\forall x^{\mathsf{Nat}}.Even(f \bullet x) \vdash f3 + 1 :_u Odd(u) \tag{1.7}$$

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

$$\forall x^{\mathsf{Nat}}.Even(f \bullet x) \vdash f3 + f(f5) + 1 :_u Odd(u) \tag{1.8}$$

By using name f, the term which contains multiple f can share the single specification in the environment. The derivation of (1.7) and (1.8) starts from the following instance of the axiom for a variable.

$$\forall x^{\mathsf{Nat}}.Even(f \bullet x) \vdash f :_m \forall x^{\mathsf{Nat}}.Even(m \bullet x) \tag{1.9}$$

The specification says that if we assume $\forall x^{\mathsf{Nat}}.Even(f \bullet x)$ 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 (1.9), we can easily reach (1.7) and (1.8), using the rule for application as well as a similar rule for first-order operations (all of which are later presented in Section 3).

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.

$$\mathsf{T} \vdash \mathtt{let}\ f = N\ \mathtt{in}\ L :_u Odd(u) \tag{1.10}$$

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 (1.10) follows.

$$\frac{(1.3), \quad (1.8), \quad \forall x^{\mathsf{Nat}}.Even(f \bullet x) \supset \forall x^{\mathsf{Nat}}.Even(f \bullet x)}{\mathsf{T} \vdash \mathtt{let}\ f = N\ \mathtt{in}\ L :_u Odd(u)} \tag{1.11}$$

Note the third condition in the antecedent is a trivially valid tautology. In this tautology, the first $\forall x^{\mathsf{Nat}}.Even(f \bullet x)$ is the conclusion of (1.3) while the second one is the assumption of (1.8): 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 $C_1 \supset C_2$. (1.11) may also be contrasted with the type derivation for the same program.

$$\frac{\vdash N : \mathsf{Nat} \Rightarrow \mathsf{Nat} \quad f : \mathsf{Nat} \Rightarrow \mathsf{Nat} \vdash L : \mathsf{Nat}}{\vdash \mathtt{let}\ f = N\ \mathtt{in}\ L : \mathsf{Nat}} \tag{1.12}$$

This way of describing and reasoning about functional behaviour easily extends to polymorphic behaviours. For example, the universal identity will be given the following judgement:

$$\mathsf{T} \vdash \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.x :_u \forall \mathrm{X}.\forall x^{\mathrm{X}}.(u^{\forall \mathrm{X}.\mathrm{X} \Rightarrow \mathrm{X}} \bullet \mathrm{X}) \bullet x = x. \tag{1.13}$$

Above $u \bullet \mathrm{X}$ is the result of applying a type X to the polymorphic abstraction u. Again the algorithmic entity (of a rather subtle kind as is well-known from our foregoing semantic studies) is named as u which makes it amenable for logical description. Further examples of assertions and derivations for pure functions are explored in [23, 25].

Mutable Higher-Order Functions with Locality In our approach to program logics, behaviour is located at names. We can naturally extend this view to stateful computation. A typical example is a memory location x, storing 7: it can be sent a read message to trigger its reply 7. Or we can ask it to store a new value. For example, the following is a specification of a program with store x.

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

where the logical term $!x$ represents dereferencing the memory cell x, cf. [37]. The resulting behaviour of the program is located in the anchor name u. The assertion says: it is true that the program $2{+}!x$ returns 9 whenever x stores 7. In this stateful language, judgements have the following form:

$$\{C\}\ M :_u \{C'\} \tag{1.14}$$

In retrospect, this may be seen as a Hoare triple integrated with functional specifications. Above, M's behaviour is located at u and it can be referred to via u in C'. For example, with side-effects, we can derive:

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

where ";" is sequential composition, as can be encoded using `let`. Naming also does work to simply specify functions stored into the memory. For example, $z := N$ with $N = \lambda x.x + x$ is represented just by replacing the anchor u in the specification of N by $!z$ as follows.

$$\{\mathsf{T}\} z := N :_f \{f = () \wedge \forall x.Even(!z \bullet x)\} \tag{1.15}$$

where $()$ is a unit; since unit represents the termination of command, we can abbreviate this formula to $\{\mathsf{T}\} z := N \{\forall x.Even(!z \bullet x)\}$ like the original Hoare triple.

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 (1.8), we define $M \stackrel{\text{def}}{=} \mathtt{let}\ f = W\ \mathtt{in}\ L$. Now this function does not just satisfy $Odd(u)$, but also *changes memory cell w* when invoked. Hence we would expect the following:

$$\{!w = 0\}\ \mathtt{let}\ f = W\ \mathtt{in}\ L :_u\ \{Odd(u) \wedge !w = 3\} \tag{1.16}$$

How can we specify the behaviour of W to reach (1.16)? 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'.

Note that this predicate mimics the judgement (1.14). "$\searrow$" indicates the evaluation of $f \bullet x$ results 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:

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

$EvenS(w, f)$ specifies a procedure which, when invoked, would increment w as well as returning an even number: if function f is used when $!w = 0$, then f's return value is even and $!w = 1$. In fact we can derive, from $EvenS(w, f)$:

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

using the standard entailment 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, the precondition C (with the assumption itself placed in the pre-condition of the judgement) and C' as post-condition. Repeating this, we can derive (1.16) in a compositional way, using essentially the same `let`-rule.

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:

$$\begin{aligned}\mathtt{T} \vdash \lambda y.(!x)y :_u\ & \{EvenS(w, !x) \wedge !w = 0\} \\ & u \bullet n \searrow z\ \{Even(z) \wedge EvenS(w, !x) \wedge !w = 1\}\end{aligned}$$

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.

The presence of local variables leads to further complexity. Consider the following program which creates a local reference w, shared by global variables a and b.

$$a := (\mathtt{new\ var}\ w := 0\ \mathtt{in}\ W);\ b := !a;\ (!b)((!a)2)$$

where `newvar` is a local variable declaration [52] and $W = \lambda x.(w := 1 + !w); x + !w$. Note that w is initially local, or *hidden*; moreover (as we can examine using the standard reduction semantics [14]), when b is assigned the content of a, the scope of w is extruded [36] beyond a, so that it is now shared between a and b. The content of w is updated to 1 when a is dereferenced and applied to 2, and is further updated when b is dereferenced and applied. Precise description of the behaviour of this program can be given with a predicate that uses a *scope opening mechanism* combined with a *name hiding operator* (νw), originating with logics for the π-calculus [22, 23, 36]. We shall introduce this concept with detailed illustration in a later section.

Summary of Contributions

- Introduction of a compositional program logic for polymorphic higher-order procedures with local state and shared mutable data structures, extending our earlier work [25]. Valid assertions characterise programs' behaviour up to the observational congruence of the language. The descriptive power of assertions and the use of proof rules are demonstrated through examples.
- A logical justification of a type-based secure information flow analysis [11, 31] as an application of the proposed program logic. Noninterference properties in different classes of languages are precisely characterised as assertions in the logic. This leads to a short correctness proof of the secrecy typing [24] via soundness of the proof rules.

Structure of the Paper Section 2 reviews the syntax and typing of the language. Section 3 introduces assertions for global state, their semantics and proof rules. Section 4 extends the logic to local state. Section 5 applies the logic to secrecy analysis. Section 6 concludes the paper. Appendix (and parts of [23]) contains proofs and axially definitions.

2 Call-by-Value Polymorphic PCF with References

2.1 Syntax and Reduction

This section briefly reviews the syntax and types of the programming language we shall use in the present inquiry. The language is call-by-value PCF with unit, sums and products, universal and existential second-order polymorphism, and recursive types, extended with references (imperative variables) [42]. The grammar of programs is given by the following BNF, assuming an infinite set of *(term) variables* ($x, y, z, \ldots$, also called *names*) and another infinite set of *type variables* ($\mathrm{X}, \mathrm{Y}, \ldots$). $\vec{M}$ denotes a vector of terms.

$$
\begin{array}{rcl}
V,W & ::= & \mathbf{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \overset{\varphi}{\Rightarrow} \beta}.\lambda y^{\alpha}.M \mid \langle V,W \rangle \\
& \mid & \mathtt{in}_i(V) \mid \Lambda \mathrm{X}.V \mid \mathtt{pack}\,\langle \beta, V \rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha \\
M,N & ::= & V \mid VW \mid V := W \mid\ !V \mid \mathtt{op}(\vec{V}) \mid \pi_i(V) \\
& \mid & \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N \\
& \mid & \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}} \mid V\beta \\
& \mid & \mathtt{unpack}\ V\ \mathtt{as}\ \langle \mathrm{X}, x \rangle\ \mathtt{in}\ M \mid \mathtt{let}\ x = N\ \mathtt{in}\ M
\end{array}
$$

The grammar uses types ($\alpha, \beta, \ldots$) which are given later. We have not included the local variable declaration (the "new" construct) which will be introduced in Section 4. Binding etc. is standard and $\mathsf{fv}(M)/\mathsf{ftv}(M)$ denote the set of free variables/type variables. Types annotating bound variables are often omitted. $V, W, \ldots$ are *values*, while $M, N, \ldots$ are *programs*. Constants are ranged over by $\mathbf{c}$, examples are the unit $()$, natural numbers $\mathtt{n}$ (or concretely $0, 1, \ldots$) and booleans $\mathtt{b}$ (or $\mathtt{t}, \mathtt{f}$). Likewise, $\mathtt{op}(V_1, ..., V_n)$ is a standard n-ary arithmetic or boolean operation, such as $M + N$, $M - N$, $M \times N$, $M = N$ (equality of two numbers), $\neg M$ (negation), $M \wedge N$ and $M \vee N$. $!V$ is dereference and $V := W$ is an assignment. All these and the remaining constructs are standard [14, 42]. The syntax restricts the place where evaluation of a program occurs the $\mathtt{let}$ construct [38, 47], which does not lead to a loss of expressiveness. In the present inquiries, the syntax gives a simpler, decomposed presentation of proof rules for our logic. For readability, we shall sometimes write e.g. $x := !y$, which in fact stands for $\mathtt{let}\ z = !y\ \mathtt{in}\ x := z$.

The dynamics of this language is given by the reduction relation, using a store [14, 42]. 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 σ. The reduction is written:

$$(M, \sigma) \ \longrightarrow\ (M', \sigma')$$

which says: *M in the presence of a store σ reduces to M' with another store σ*. When M neither refers to nor changes a store, we also write:

$$M \ \longrightarrow\ M' \ .$$

We also write $(M, \sigma) \Downarrow (V, \sigma')$ for $(M, \sigma) \rightarrow^* (V, \sigma')$, and $M \Downarrow V$ for $M \rightarrow^* V$. The reduction relation is generated from the following rules, which are those of the standard one-step call-by-value evaluation with store [14, 42]. We first list the rules for assignment and dereference. Below $\sigma(x)$ and $\sigma[x \mapsto V]$ indicate $x \in \mathsf{dom}(\sigma)$.

$$\begin{array}{rcl} (!x,\ \sigma) & \to & (\sigma(x),\ \sigma) \\ (x := V,\ \sigma) & \to & ((),\ \sigma[x \mapsto V]) \end{array}$$

The following is the standard reduction rules for call-by-value PCF (we omit obvious symmetric rules; the rules for first-order operations are also omitted).

$$\begin{array}{rcl} (\lambda x.M)V & \to & M[V/x] \\ \pi_1(\langle V_1, V_2\rangle) & \to & V_1 \\ \mathtt{case}\ \mathtt{in}_1(W)\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} & \to & M_1[W/x_1] \\ \mathtt{if}\ \mathtt{t}\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 & \to & M_1 \\ (\mu f.\lambda g.N)W & \to & N[W/g][\mu f.\lambda g.N/f] \\ \mathtt{let}\ x = V\ \mathtt{in}\ N & \to & N[V/x] \end{array}$$

The rules for the type application and unpacking are also standard.

$$\begin{array}{c} (\Lambda \mathrm{X}.V)\alpha \to V[\alpha/\mathrm{X}] \\ \mathtt{unpack}\ \mathtt{pack}\ \langle \beta, V\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha\ \mathtt{as}\ \langle \mathrm{X}, x\rangle\ \mathtt{in}\ N \to N[\beta/\mathrm{X}][V/x] \end{array}$$

Finally there are two rules for reduction under contexts.

$$\frac{M \to M'}{(M,\sigma) \to (M',\sigma)} \qquad \frac{(M,\sigma) \to (M',\sigma')}{(\mathtt{let}\ x = M\ \mathtt{in}\ N, \sigma) \to (\mathtt{let}\ x = M'\ \mathtt{in}\ N, \sigma')}$$

2.2 Types and Typing

The grammar of types is standard [14, 42] except for equipping each arrow type with a finite sequence of variables, called *effect* [56], ranged over by φ, φ'. For simplicity we mostly treat effects as if they were sets. An effect indicates the variables a procedure may read or write during evaluation. This enriched typing will become useful later when we consider types for assertions. The grammar of types follows.

$$\begin{array}{rl} \alpha, \beta ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \overset{\varphi}{\Rightarrow} \beta \mid \alpha \times \beta \mid \alpha + \beta \\ & \mid\ \mathrm{X} \mid \forall \mathrm{X}.\alpha \mid \exists \mathrm{X}.\alpha \mid \mu \mathrm{X}.\alpha \\ \rho ::= & \alpha \mid \mathsf{Ref}(\alpha) \end{array}$$

Here $\alpha, \beta, \ldots$ are *function types*, while $\mathsf{Ref}(\alpha), \ldots$ are *reference types*. Reference types are restricted to carrying only non-reference types. As is well-known, in the presence of local state, this does not lead to a loss of expressiveness since a reference can be encoded using a product (for further discussions on this point, see Section 6). An *environment* is a finite map from variables to types and $\mathsf{dom}(\Gamma)$ (resp. $\mathsf{dom}_f(\Gamma)$, resp. $\mathsf{dom}_r(\Gamma)$) denotes the set of variables (resp. of non-reference or function types, resp. of reference types) in Γ. $\mathsf{ftv}(\Gamma)$ is the set of type variables occurring in types in Γ.

Fig. 1 Typing Rules.

$$[\mathit{Var}]\ \frac{-}{\Gamma, x{:}\rho \vdash x : \rho} \quad [\mathit{Unit}]\ \frac{-}{\Gamma \vdash () : \mathsf{Unit}} \quad [\mathit{Bool}]\ \frac{-}{\Gamma \vdash \mathtt{b} : \mathsf{Bool}}$$

$$[\mathit{Num}]\ \frac{-}{\Gamma \vdash \mathtt{n} : \mathsf{Nat}} \quad [\mathit{Eq}]\ \frac{\Gamma \vdash V_{1,2} : \mathsf{Nat}}{\Gamma \vdash V_1 = V_2 : \mathsf{Bool}}$$

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

$$[\mathit{App}]\ \frac{\Gamma \vdash V : \alpha \stackrel{\varphi}{\Rightarrow} \beta \quad \Gamma \vdash W : \alpha}{\Gamma \vdash VW : \beta, \varphi} \quad [\mathit{Promote}]\ \frac{\Gamma \vdash M : \alpha}{\Gamma \vdash M : \alpha, \emptyset}$$

$$[\mathit{If}]\ \frac{\Gamma \vdash V : \mathsf{Bool} \quad \Gamma \vdash N_{1,2} : \alpha, \varphi}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha, \varphi} \quad [\mathit{Inj}]\ \frac{\Gamma \vdash V : \alpha_i}{\Gamma \vdash \mathtt{in}_i(V) : \alpha_1 + \alpha_2}$$

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

$$[\mathit{Pair}]\ \frac{\Gamma \vdash V_i : \alpha_i\ (i = 1,2)}{\Gamma \vdash \langle V_1, V_2 \rangle : \alpha_1 \times \alpha_2} \quad [\mathit{Proj}]\ \frac{\Gamma \vdash V : \alpha_1 \times \alpha_2}{\Gamma \vdash \pi_i(V) : \alpha_i\ (i = 1,2)}$$

$$[\mathit{Deref}]\ \frac{\Gamma \vdash x : \mathsf{Ref}(\alpha)}{\Gamma \vdash !x : \alpha, \{x\}} \quad [\mathit{Assign}]\ \frac{\Gamma \vdash V : \alpha \quad \Gamma \vdash x : \mathsf{Ref}(\alpha)}{\Gamma \vdash x := V : \mathsf{Unit}, \{x\}}$$

$$[\mathit{Weak}]\ \frac{\Gamma \vdash M : \alpha, \varphi \quad \Gamma \vdash x : \mathsf{Ref}(\beta)}{\Gamma \vdash M : \alpha, \varphi \cup \{x\}}$$

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

$$[\mathit{Unpack}]\ \frac{\Gamma \vdash V : \exists \mathrm{X}.\alpha \quad \Gamma, x : \alpha \vdash N : \beta, \varphi \quad \mathrm{X} \notin \mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\beta)}{\Gamma \vdash \mathtt{unpack}\ V\ \mathtt{as}\ \langle \mathrm{X}, x \rangle\ \mathtt{in}\ N : \beta, \varphi}$$

$$[\mathit{Pack}]\ \frac{\Gamma \vdash V : \alpha[\beta/\mathrm{X}]}{\Gamma \vdash \mathtt{pack}\ \langle \beta, V \rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha : \exists \mathrm{X}.\alpha} \quad [\mathit{Iso}]\ \frac{\Gamma \vdash M : \alpha, \varphi \quad \alpha \approx \beta}{\Gamma \vdash M : \alpha, \varphi}$$

$$[\mathit{Let}]\ \frac{\Gamma \vdash N : \alpha, \varphi_2 \quad \Gamma, x : \alpha \vdash M : \beta, \varphi_1}{\Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \beta, \varphi_1 \cup \varphi_2}$$

The typing judgements are $\Gamma \vdash V : \rho$ for values and $\Gamma \vdash M : \alpha, \varphi$ for programs. We write Γ, Δ for the union of Γ and Δ, always assuming $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. These typing judgements are derived with the rules in Figure 1 (omitted rules for first-order operations are just as [*Eq*]). All rules are standard [42, 56]. [*Deref*] and [*Assign*] record read and write effects; [*Abs*] enforces that the function type $\alpha \stackrel{\varphi}{\Rightarrow} \beta$ "remembers" the effects M may have; when a function is used by [*App*], its effects φ is the effect of the application. In [*Iso*] (based on the equi-recursive approach [42]), $\approx$ denotes the tree equivalence of two types, which includes standard (un)folding, $\mu \mathrm{X}.\alpha \approx \alpha[\mu \mathrm{X}.\alpha/\mathrm{X}]$.

Write $\Gamma \vdash \sigma$ for $\Gamma \vdash \sigma(x) : \Gamma(x)$ for each $x \in \mathsf{dom}_r(\Gamma)$. We observe:

Proposition 1. *Assume* $\Gamma \vdash \sigma$. *Then* $\Gamma \vdash M : \beta, \varphi$ *and* $(M, \sigma) \to (N, \sigma')$ *imply* $\Gamma \vdash N : \beta, \varphi$ *and* $\Gamma \vdash \sigma'$.

PROOF: Standard by the substitution lemma: $\Gamma, x : \rho \vdash M : \rho', \varphi$ and $\Gamma \vdash V : \rho$ implies $\Gamma \vdash M[V/x] : \rho, \varphi$. $\square$

We also introduce two basic notions we shall use later.

Definition 2. (semi-closed programs) A well-typed program of a non-reference type, say $\Gamma \vdash M : \alpha, \varphi$, is *semi-closed* [34] (resp. *closed*) when $\mathsf{ftv}(\Gamma) = \emptyset$ and $\mathsf{dom}(\Gamma) = \varphi$ (resp. $\mathsf{ftv}(\Gamma) = \emptyset$ and $\mathsf{dom}(\Gamma) = \varphi = \emptyset$). Similarly for $\Gamma \vdash M : \alpha$.

Definition 3. (contextual congruence) Let $\Gamma \vdash M_{1,2} : \alpha, \varphi$. Then $\Gamma \vdash M_1 \cong M_2 : \alpha, \varphi$ is the maximum typed congruence satisfying, for each semi-closed $\Delta \vdash M_{1,2} : \mathsf{Unit}$, and for each σ such that $\Delta \vdash \sigma$, we have $(M_1, \sigma) \Downarrow$ iff $(M_2, \sigma) \Downarrow$.

3 Logic for Higher-Order Functions with Global State

3.1 Terms and Formulae

The logical language we shall use is that of the first-order logic with equality [33, §2.8] except a predicate that asserts on evaluation of stateful expressions. The grammar of the language follows.

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

The first set of expressions (ranged over by $e, e', \ldots$) are *terms* of our logical language, while the second set are its *formulae* (ranged over by $A, B, C, C' \ldots$). Both terms and formulae are from [25] except $!x$ and $\{C\}\ e \bullet e' \searrow x\ \{C'\}$.[1] Terms include all the constants and first-order operations (including multiplication) from the target programming language. The first three terms in the second line are a paring, a projection and an injection, respectively. The next two are a type application for a universal type and a packing for an existential type, respectively. $e \bullet [\beta]$ corresponds to $M\beta$, while $\langle \beta, e \rangle$ corresponds to a packed pair. The final term, $!e$, denotes the dereference of a reference e (in fact, by the typing we introduce later, this term is essentially always used in the form $!x$). We let $\mathsf{fv}(e)$ denote the *free variables* in e. Variables are sometimes called *names*.

Formulae are those of the first-order predicate calculus with equality except the last three. In $\forall x^{\alpha}.A$ and $\exists x^{\alpha}.A$, the type of x should be a non-reference type since we in effect treat reference variables as constants (as will be made explicit in the semantics of assertions discussed later). We also have quantifications over types, $\forall \mathrm{X}.A$ and $\exists \mathrm{X}.A$, by which we can describe polymorphic behaviours. The predicate $\{C\}\ e \bullet e' \searrow x\ \{C'\}$, called *evaluation formulae*, plays a central rôle in representing behaviours of stateful higher-order programs. In $\{C\}\ e \bullet e' \searrow x\ \{C'\}$, 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. As noted below, e in $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ is typed with an arrow type $\alpha \overset{\varphi}{\Rightarrow} \beta$, where names in φ indicate references which may be read/written during the evaluation of $e_1 \bullet e_2$, as described in C and C'. The effect φ plays an important rôle in the semantics of the predicate, as we shall discuss later.

Convention 4. Logical connectives are used with their standard precedence/association. For example, $\neg A \wedge B \supset \forall x.C \vee D \supset E$ is parsed as $((\neg A) \wedge B) \supset (((\forall x.C) \vee D) \supset E)$. We write $A \equiv B$ for $(A \supset B) \wedge (B \supset A)$, that is when A and B are logically equivalent. We also use the truth T (definable as $1 = 1$) and the falsity F (that is $\neg\mathsf{T}$). We always assume the standard bound name convention on terms and formulae. $\mathsf{fv}(A)$ and $\mathsf{ftv}(A)$ respectively denote the sets of free variables and type variables in A. Formulae are often called *assertions*.

[1] In the logics for pure functions studied in [22, 25], terms include $\perp^{\alpha}$ for each α which denotes divergence. Their inclusion in the present logic is harmless but is unnecessary. See §5.1 for related discussions.

Typing on terms and formulae play an important rôle in the present logic, which we introduce below.

Definition 5. (well-typedness (1): terms) We say *e is typed with* ρ if one of the following clauses hold.

- $()$, $\mathbf{n}$, $\mathbf{b}$ and x^{ρ} are well-typed with types Unit, Nat, Bool and ρ, respectively.
- $\neg e$ is well-typed with type Bool, if e is well-typed with type Bool. $e_1 + e_2$ is well-typed with type Nat if $e_{1,2}$ are well-typed with type Nat. Similarly for $\mathtt{succ}$, $\times$, $-$, and other first-order operators.
- $!(x^{\mathsf{Ref}(\alpha)})$ is well-typed with type α.
- $\langle e_1, e_2 \rangle$ 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 $\pi_2(e)$.
- $\mathsf{inj}_1^{\alpha+\beta}(e)$ is well-typed with type $\alpha + \beta$ if e is well-typed with type α. Similarly for $\mathsf{inj}_2^{\alpha+\beta}(e)$.
- $e \bullet [\beta]$ 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 $\alpha[\beta/\mathrm{X}]$.

We write $\Gamma \vdash e : \rho$ (read: "e has type ρ under Γ") if e is well-typed with type ρ and each variable occurring in e (if any) is given an identical type in e and Γ.

Observe each well-typed e has a unique type. Further, given a term e, it suffices to go through the syntax tree of e once to know the well-typedness of, and the assignable type for, e, so these properties are linearly decidable. We next type formulae.

Definition 6. (well-typedness (2): formulae) The relation $\Gamma \vdash A$ (read: *A is well-typed under* Γ) is generated from the following rules.

- $\Gamma \vdash e_1 = e_2$ if $\Gamma \vdash e_i : \rho$ for $i = 1, 2$.
- $\Gamma \vdash \neg A$ if $\Gamma \vdash A$.
- $\Gamma \vdash A_1 \star A_2$ if $\Gamma \vdash A_i$ $(i = 1, 2)$, for $\star \in \{\cup, \cap, \supset\}$.
- $\Gamma \vdash \forall x^{\alpha}.A$ and $\Gamma \vdash \exists x^{\alpha}.A$ if $\Gamma, x\!:\!\alpha \vdash A$ with x fresh.
- $\Gamma \vdash \forall \mathrm{X}.A$ and $\Gamma \vdash \exists \mathrm{X}.A$ if $\Gamma \vdash A$ where $\mathrm{X} \notin \mathsf{ftv}(\Gamma)$.
- $\Gamma \vdash \{C\}\ e_1 \bullet e_2 \searrow z\ \{C'\}$ if $\Gamma \vdash e_1 : \alpha \stackrel{\varphi}{\Rightarrow} \beta$ and $\Gamma \vdash e_2 : \alpha$ and, moreover, $\Gamma \vdash C$ and $\Gamma, z : \beta \vdash C'$.

Again it is linearly calculable whether $\Gamma \vdash A$ given Γ and A. Further given A, the minimum Γ for which $\Gamma \vdash A$ holds is also linearly calculable.

Notation 7. We often use the following abbreviations.

1. If e_1 has a type of the form $\alpha \stackrel{\emptyset}{\Rightarrow} \beta$, we often regard $e_1 \bullet e_2$ as a term.
2. If e' is not a variable, $\{C\}\ e_1 \bullet e_2 \searrow e'\ \{C'\}$ stands for $\{C\}\ e_1 \bullet e_2 \searrow x\ \{x = e' \wedge C'\}$ with x fresh.
3. $\{C\}\ e_1 \bullet e_2\ \{C'\}$ stands for $\{C\}\ e_1 \bullet e_2 \searrow z\ \{z = () \wedge C'\}$.

We give simple examples of assertions for illustration. *Below and henceforth we often omit type annotations on free variables in terms and formulae.*

Example 8. (assertions and their types)

1. $y = 6$ is an assertion which says y is equal to 6. On the other hand, $!y = 6$ says the content of a memory cell y is equal to 6. In the former, y is typed as Nat, while, in the latter, y is typed as $\mathsf{Ref}(\mathsf{Nat})$. Thus we have $y : \mathsf{Ref}(\mathsf{Nat}) \vdash\ !y = 6$.

2. The assertion $\mathit{Double}(u) \stackrel{\text{def}}{=} \forall n^{\mathsf{Nat}}.(u \bullet n = 2 \times n)$ says that a function named as u with type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ always returns double of the argument. This is satisfied by, for example, $\lambda z.2 \times z$. This predicate is typed as: $u : \mathsf{Nat} \Rightarrow \mathsf{Nat} \vdash \mathit{Double}(u)$.

3. Using the notation in (2) above, $\mathit{Double}(!x) = \forall n^{\mathsf{Nat}}.(!x \bullet n = 2 \times n)$ says that x stores a doubling function. Note we have $x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat}) \vdash \mathit{Double}(!x)$. This assertion is satisfied, for example, after $x := \lambda z.2 \times z$ is executed.

4. The assertion $\{\mathit{Double}(!x)\}\, u \bullet 3 \searrow 6\, \{\mathit{Double}(!x)\}$ says that, if 3 is fed to the function named by u under the initial state $\mathit{Double}(!x)$, then the result is 6, with the same final state. We have a typing

$$u : \mathsf{Nat} \stackrel{x}{\Rightarrow} \mathsf{Nat} \ \vdash\ \{\mathit{Double}(!x)\}\, u \bullet 3 \searrow 6\, \{\mathit{Double}(!x)\}$$

   where the type of u indicates not only its argument and target are natural numbers but also an invocation may access x. The assertion is satisfied by, for example, $\lambda y.(!x)y$, named as u.

5. The assertion

$$C \stackrel{\text{def}}{=} \forall i, n.\ \{!w = n\}\, !x \bullet i \searrow 2 \times i\, \{!w = n + 1\}$$

   says x stores a function which, when invoked, would increment w as well as returning the double of the argument. For typing we have $x : \mathsf{Ref}(\mathsf{Nat} \stackrel{w}{\Rightarrow} \mathsf{Nat}), w : \mathsf{Ref}(\mathsf{Nat}) \vdash C$. This predicate is satisfied when, for example, $f(w) \stackrel{\text{def}}{=} \lambda z.(w := !w + 1; z \times 2)$ is stored in x.

6. Using C in (4) above, let:

$$C' \stackrel{\text{def}}{=} \{C \wedge\ !w = 0\}\, u \bullet 3 \searrow 6\, \{C \wedge\ !w = 1\}$$

   This predicate 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 $!w = 1$. For typing, we have $x : \mathsf{Ref}(\mathsf{Nat} \stackrel{w}{\Rightarrow} \mathsf{Nat}), w : \mathsf{Ref}(\mathsf{Nat}), u : \mathsf{Nat} \stackrel{w}{\Rightarrow} \mathsf{Nat} \vdash C$. This is again satisfied by $\lambda y.(!x)y$ named as u, under the assumption x stores $f(w)$ above.

3.2 Judgement

As in Hoare Logic, a judgement in the present program logic consists of a pair of formulae and a program. There are two kinds of judgements. One is for programs which do not change state, while the other is for general programs. Below we present the sequents for validity: those for provability simply replace $\models$ with $\vdash$.

$$A \models M^{\Gamma;\rho} :_u B \qquad\qquad \models \{C\}\, M^{\Gamma;\rho;\varphi} :_u \{C'\}$$

We often write $\{C\}\ M^{\Gamma;\alpha;\varphi} :_u \{C'\}$ for the second sequent, either for validity or for provability (which should be clear from the context). In $A \models M^{\Gamma;\rho} :_u B$, the annotated program $M^{\Gamma;\rho}$ is the *subject* of the judgement; u its *anchor*; A its *rely condition* and B its *guarantee condition*. Similarly, in $\models \{C\}\, M^{\Gamma;\rho;\varphi} :_u \{C'\}$, $M^{\Gamma;\rho;\varphi}$ is the *subject* of the judgement; u its *anchor*; C its *pre-condition*; and C' its *post-condition*. We only treat well-typed judgements, in the following sense.

Definition 9. (well-typedness (3): judgement) The following clauses apply to judgements on provability by replacing $\models$ with $\vdash$.

- $A \models M^{\Gamma;\rho} :_u B$ is well-typed if: (1) $\Gamma \vdash M : \rho$ and (2)$\Gamma, \Theta \vdash A$ and $u : \rho, \Theta \vdash B$ for some Θ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma) \cup \{u\}) = \emptyset$.
- $\models \{C\}\, M^{\Gamma;\rho;\varphi} :_u \{C'\}$ is well-typed if (1) $\Gamma \vdash M : \rho, \varphi$ and (2) $\Gamma, \Theta \vdash C$ and $\Gamma, u : \rho, \Theta \vdash C'$ for some Θ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma) \cup \{u\}) = \emptyset$.

Note Θ in each clause in Definition 9 assigns types to those names which are used in formulae but which are neither an anchor nor free variables of the typed program. We call these names *auxiliary*, as in Hoare logic. Formally:

Definition 10. (primary/auxiliary names in a judgement) Let $\models \{C\}\, M^{\Gamma;\rho;\varphi} :_u \{C'\}$ be well-typed. Then the set of *primary names* in this judgement are $\mathsf{dom}(\Gamma) \cup \{u\}$. The set of *auxiliary names* in this judgement are those free names in C and C' which are not primary. Similarly for $A \models M^{\Gamma;\alpha} :_u B$.

Example 11. In an assertion "$x = i \models 2 \times x :_u u = 2 \times i$", x and u are primary while i is auxiliary. u is in addition its anchor.

Intuitively, $A \models M^{\Gamma;\rho} :_u B$ asserts that if M can rely on the environment to supply values satisfying A, then M with those values substituted terminates and satisfies B. On the other hand, $\{C\}\ M^{\Gamma;\alpha;\varphi} :_u \{C'\}$ says that if M can rely on the environment to supply values satisfying C (for the non-reference free variables), and if we evaluate M starting from any state satisfying C (for reference-typed free variables), then the evaluation terminates so that the final state and the resulting value, named as u, together[2] guarantee C'.

These informal readings are formalised in the next subsection.

[2] By the typing, the post-condition C' in fact also talk about non-reference primary names, which are stateless and which are therefore cannot differ from what is stated in the pre-condition C. So, in the logic with global state, what a post-condition would specify about non-reference free variables does not count (and may not have any practical value). This convention however makes proof rules simpler and becomes essential when we incorporate local state.

3.3 Models, Interpretation and Satisfaction

Assertions and judgements in the present logic talk about behaviours of programs. One of the best places to find a universe of behaviours we are interested in, therefore, is those programs themselves (other options include typed processes, as discussed in [23] which gives the equivalent model). We thus construct a model from programs themselves.

A *model* is a pair (ξ,σ) where ξ is a finite map from names to semi-closed values (cf. Definition 2, Section 2) in which no free type variables occur; σ is another map of the same kind. We write $\mathcal{M}_1 \cdot \mathcal{M}_2$ for the component-wise union of $\mathcal{M}_1$ and $\mathcal{M}_2$ under the assumption their subjects are disjoint.

Intuitively, ξ and σ in (ξ,σ) respectively denote a standard functional environment and a store. Models are typed following this intuitive idea.

$$\frac{\forall x \in \mathsf{dom}(\xi).\ \Gamma \vdash \xi(x) : \Gamma(x)}{\Gamma \vdash \xi} \qquad \frac{\forall x \in \mathsf{dom}(\sigma).\ \Gamma(x) = \mathsf{Ref}(\alpha)}{\Gamma \vdash \sigma} \qquad \frac{\Gamma \vdash \xi \quad \Gamma \vdash \sigma}{\Gamma \vdash (\xi,\sigma)}$$

We can now formalise the semantics of assertions, after a convention.

Convention 12. By the need to close types when discussing semantics of assertions, we henceforth let I denote a model combined with a finite map from type variables to closed types. For convenience we write I_a for the model part of I and I_t for the part of a map from free type variables to closed types in I (so that we have $I_a \cup I_t = I$). Accordingly we write $\Theta \vdash I$ to indicate the well-typedness so that Θ is an environment extended with free type variables.

First we interpret terms under a model. Below, assuming $\Gamma \vdash \mathcal{M}$ and $\mathcal{M} = (\xi,\sigma)$, we write $x^{\alpha} : V \in \mathcal{M}$ when $\Gamma(x) = \alpha$ and $\xi(x) = V$; similarly we write $x^{\mathsf{Ref}(\alpha)} \mapsto V \in \mathcal{M}$ when $\Gamma(x) = \mathsf{Ref}(\alpha)$ and $\sigma(x) = V$.

Definition 13. (interpretation of terms) Let $\Gamma \vdash e : \rho$ for some ρ and $\Gamma \vdash \mathcal{M} \cdot I$. Then the *interpretation of e under* $\mathcal{M} \cdot I$, denoted $[\![e]\!]\mathcal{M} \cdot I$, is given by the following clauses.

$$\begin{aligned}
[\![x^{\alpha}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} V && (x^{\alpha} : V \in \mathcal{M}\cdot I)\\
[\![!x^{\mathsf{Ref}(\alpha)}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} V && (x^{\mathsf{Ref}(\alpha)} \mapsto V \in \mathcal{M}\cdot I)\\
[\![x^{\mathsf{Ref}(\alpha)}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} x\\
[\![\mathbf{c}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \mathbf{c}\\
[\![\mathbf{op}(\vec{e})]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \mathbf{op}([\![\vec{e}]\!]\mathcal{M}\cdot I)\\
[\![\langle e, e'\rangle]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \langle [\![e]\!]\mathcal{M}\cdot I, [\![e']\!]\mathcal{M}\cdot I\rangle\\
[\![\pi_i(e)]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \pi_i([\![e]\!]\mathcal{M}\cdot I)\\
[\![\mathsf{inj}^{i}_{\alpha+\beta}(e)]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \mathtt{in}_i([\![e]\!]\mathcal{M}\cdot I)\\
[\![e \bullet [\beta]]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} ([\![e]\!]\mathcal{M}\cdot I)([\![\beta]\!]I_t)\\
[\![\langle \beta, e\rangle^{\exists \mathrm{X}.\alpha}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} \mathtt{pack}\,\langle [\![\beta]\!]I_t, [\![e]\!]\mathcal{M}\cdot I\rangle\ \mathtt{as}\ \exists \mathrm{X}.\alpha
\end{aligned}$$

where, in the 5th, 7th and 9th clauses, the expressions on the right-hand side are taken to be the values resulting from evaluation of the given expressions.

Remark 14. (interpretation of terms)

1. In the fifth line, $\mathbf{op}([\![\vec{e}]\!]\mathcal{M}\cdot I)$ evaluates to a closed value since, for arguments which are closed values, each first-order operation always defines a total function. One may also observe any semi-closed values of atomic types are always closed values.
2. In the seventh line, a semi-closed value of a product type always has the shape $\langle V_1, V_2\rangle$ with $V_{1,2}$ semi-closed, hence the expression converges to a semi-closed value. Similarly, in the ninth line, a semi-closed value of type $\forall \mathrm{X}.\alpha$ has the shape $\Lambda \mathrm{X}.V$ with V semi-closed, so that the expression evaluates to a semi-closed value.
3. If we add $e_1 \bullet e_2$ as a term under the assumption $\Gamma \vdash e_1 : \alpha \overset{\emptyset}{\Rightarrow} \beta$ and $\Gamma \vdash e_2 : \alpha$, then the interpretation of $e_1 \bullet e_2$ can be divergent, so that the co-domain of the interpretation should include divergent semi-closed programs, as we have done in [22, 25] (in which case we include a term denoting divergence, $\perp$, for that type).

On the basis of interpretation of terms, we define the satisfaction relation for formulae. Below, in the first clause, $\cong$ is defined in Definition 3. In the sixth and seventh clauses, $[\![\alpha]\!]$ denotes the set of semi-closed values of type α. In the final clause, $\Gamma \restriction \varphi$ is the projection of Γ onto φ.

Definition 15. (satisfaction) Given $\Gamma \vdash \mathcal{M}$ and $\Gamma \vdash C$, the relation $\mathcal{M}^\Gamma \vdash^I C$ (read: *$\mathcal{M}$ satisfies C under a typing Γ and an interpretation I*) is generated from the following clauses.

$$
\begin{array}{rcl}
\mathcal{M}^\Gamma \models^I e_1 = e_2 & \text{if} & \Gamma \vdash [\![e_1]\!]\mathcal{M}\cdot I \cong [\![e_2]\!]\mathcal{M}\cdot I \\
\mathcal{M}^\Gamma \models^I C_1 \wedge C_2 & \text{if} & (\mathcal{M}^\Gamma \models^I C_1) \wedge (\mathcal{M}^\Gamma \models^I C_2) \\
\mathcal{M}^\Gamma \models^I C_1 \vee C_2 & \text{if} & (\mathcal{M}^\Gamma \models^I C_1) \vee (\mathcal{M}^\Gamma \models^I C_2) \\
\mathcal{M}^\Gamma \models^I C_1 \supset C_2 & \text{if} & (\mathcal{M}^\Gamma \models^I C_1) \supset (\mathcal{M}^\Gamma \models^I C_2) \\
\mathcal{M}^\Gamma \models^I \neg C & \text{if} & \neg (\mathcal{M}^\Gamma \models^I C) \\
\mathcal{M}^\Gamma \models^I \forall x^\alpha.C & \text{if} & \forall V \in [\![\alpha]\!].\ \mathcal{M}^\Gamma \models^{I\cdot x:V} C \\
\mathcal{M}^\Gamma \models^I \exists x^\alpha.C & \text{if} & \exists V \in [\![\alpha]\!].\ \mathcal{M}^\Gamma \models^{I\cdot x:V} C \\
\mathcal{M}^\Gamma \models^I \forall \mathrm{X}.C & \text{if} & \forall \alpha \text{ closed}.\ \mathcal{M}^\Gamma \models^{I\cdot \mathrm{X}:\alpha} C \\
\mathcal{M}^\Gamma \models^I \exists \mathrm{X}.C & \text{if} & \exists \alpha \text{ closed}.\ \mathcal{M}^\Gamma \models^{I\cdot \mathrm{X}:\alpha} C \\
\mathcal{M}^\Gamma \models^I \{C\} e_1 \bullet e_2 \searrow x \{C'\} & \text{if} & \forall \sigma.\ (\ \Gamma \restriction \varphi \vdash \sigma \wedge (\mathcal{M} \setminus \varphi)\cdot \sigma \models^I C \\
& & \quad \supset\ \exists V, \sigma'.\ ([\![e_1]\!]\mathcal{M}\cdot I [\![e_2]\!]\mathcal{M}\cdot I,\ \sigma) \Downarrow (V, \sigma') \\
& & \qquad\qquad \text{such that } (\mathcal{M} \setminus \varphi)\cdot(x{:}V, \sigma') \models^I C'\ )
\end{array}
$$

where, in each clause, the logical connective on the right-hand side, if any, is used semantically and, in the last clause, we assume $\Gamma \vdash e : \alpha \overset{\varphi}{\Rightarrow} \beta$. We often omit Γ from $\mathcal{M}^\Gamma \models^I C$, writing simply $\mathcal{M} \models^I C$.

Remark 16. (satisfaction) Except for the first clause and the last three clauses, the definition of the satisfaction is entirely standard.

1. In the first clause, the equality of terms is interpreted as the contextual congruence. If we used semi-closed values modulo $\cong$ as the elements of our universe, then we could have used the genuine equality. The present formulation is essentially equivalent to this more abstract formulation.

2. $\forall \mathrm{X}.A$ is satisfied by a model when for each closed type α, the result of substituting α for X in A is satisfied by a model under a similar substitution. Dually for $\exists \mathrm{X}.A$. There are two notes on the semantics of $\forall \mathrm{X}.A$ and $\exists \mathrm{X}.A$. First, closed types include empty types, i.e. those types which are not inhabited by any closed programs (if we allow divergent behaviours among elements of our model, cf. Remark 14 (3), each closed type is inhabited by such a behaviour. Second, throughout the present inquiries, we are viewing types as syntactic entities and nothing else. A semantic view is possible if, for example, we regard types as classes of untyped behaviours under certain algebraic laws [20]. A recent technical development which shows such a notion does allow a consistent treatment of various type constructors [59].
3. The evaluation formula $\mathcal{M} \models^I \{C\}e_1 \bullet e_2 \searrow x\{C'\}$ is satisfied by a model if the following holds, assuming e_1 is typed as $\alpha \overset{\varphi}{\Rightarrow} \beta$:
   If the interpretation of e_1 is invoked with that of e_2 as an argument, and if the initial state of φ satisfies C, then it will converge with a value named x, where x and the final state of φ together satisfy C'.

   Note φ may as well be modelled in $\mathcal{M}$: however, when interpreting the evaluation formula, their content in $\mathcal{M}$ is neglected, since we wish to describe the behaviour of invocation under hypothetical states, not the present one (consider for example specifying the behaviour of $\lambda().x := !x+1$, which, whatever natural number x stores, increments that stored value).

Finally we have reached semantics of judgement. We draw attention of the reader to the fact that I below includes assignment of closed types to free type variables. Below $M\xi$ denotes the obvious substitution.

Definition 17. (semantics of judgement, global state)

1. $A \models M^{\Gamma;\rho} :_u B$ iff, for each I and ξ, whenever $\xi \models^I A$, we have $M I_t \xi \Downarrow V$ such that $u : V \models^I B$.
2. $\models \{C\} M^{\Gamma;\rho;\varphi} :_u \{C'\}$ iff, for each I, ξ and σ, whenever $(\xi, \sigma) \models^I C$, we have $(M I_t \xi, \sigma) \Downarrow (V, \sigma')$ such that $u : V \cdot \sigma' \models^I C'$.

Remark 18. (semantics of judgement) Definition 17 say both forms of judgements are about total correctness of programs. A close scrutiny however reveals that describable properties do include what may be called partial correctness. This is because we can describe divergence of by the negation of evaluation formulae:

$$\neg\{C\}e_1 \bullet e_2 \searrow z\{\mathsf{T}\}$$

which says that it is not the case that invoking e_1 with an argument e_2 under the initial state C terminates, that is it diverges. Write $\{C\}e_1 \bullet e_2 \Uparrow$ for this formula. Then we can write, with the type of u to be $\mathsf{Unit} \overset{x}{\Rightarrow} \mathsf{Unit}$,

$$\forall i.\ (\ \{!x = i\}u \bullet () \searrow ()\{!x = i+1\}\ \vee\ \{!x = i\}u \bullet () \Uparrow\ )$$

which says the result of running a program u under any state of a reference x means either the content of x is incremented or divergence. This is a typical partial correctness assertion. We shall come back to this point in Section 6 later.

Fig. 2 Proof Rules (1): Values

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

$$[Op]\,\frac{A\vdash V_i :_{v_i} B_i \;\; (1\le i\le n) \quad (\wedge_i B_i)\supset B[\mathsf{op}(v_1,..,v_n)/u]}{A\vdash \mathsf{op}(V_1,..,V_n) :_u B}$$

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

$$[Pair]\,\frac{A\vdash V_i :_{v_i} B_i \quad B_1\wedge B_2\supset B[\langle v_1,v_2\rangle/u]}{A\vdash \langle V_1,V_2\rangle :_u B}$$

$$[Proj_1]\,\frac{A\vdash V :_v B[\pi_1(v)/u]}{A\vdash \pi_1(V) :_u B} \qquad [In_1]\,\frac{A\vdash V :_v B[\mathtt{in}_1(v)/u]}{A\vdash \mathtt{in}_1(V) :_u B}$$

$$[T\text{-}Abs]\,\frac{A^{\text{-}\mathrm{X}}\vdash V :_v B}{A\vdash \Lambda \mathrm{X}.V :_u \forall \mathrm{X}.B[u\bullet[\mathrm{X}]/v]} \qquad [T\text{-}App]\,\frac{A\vdash V :_v \forall \mathrm{X}.B[v\bullet[\mathrm{X}]/u]}{A\vdash V\beta :_u B[\beta/\mathrm{X}]}$$

$$[T\text{-}Pack]\,\frac{A\vdash V :_v B[\langle \mathrm{X},v\rangle/u][\beta/\mathrm{X}]}{A\vdash \mathtt{pack}\,\langle\beta,V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha :_u \exists \mathrm{X}.B} \qquad [T\text{-}Rec]\,\frac{A\approx A' \quad A'\vdash V :_v B' \quad B'\approx B}{A\vdash V :_v B}$$

3.4 Proof Rules

In the following we present the proof rules for the logic with global state. As mentioned in §3.2, we use the two forms of the judgements for provability, $A\ \vdash M^{\Gamma;\rho} :_u B$ for side-effect-free programs and $\vdash \{C\}\, M^{\Gamma;\alpha;\varphi} :_u \{C'\}$ for general programs. The latter is almost always written $\{C\}\, M^{\Gamma;\alpha;\varphi} :_u \{C'\}$. Figure 2 lists the proof rules which use the first form of the judgements, while Figure 3 list those rules which use the second form as well as those which mediate these two forms. These rules precisely follow the corresponding typing rules, except the structural rules (the last four rules in Figure 3). In each rule, we assume judgements are well-typed (cf. §3.2) including one in the conclusion. The rules use the following conventions.

Convention 19.

- $A^{\text{-}\vec{y}}$ indicates no name in $\vec{y}$ occurs in A, while $A^{\vec{y}}$ indicates all *primary* names in A are in $\{\vec{y}\}$.
- $i, j, \ldots$ exclusively range over auxiliary names (in a given judgement).
- In each proof rule, a primary name in a judgement in its premise should never coincide with an auxiliary name in a judgement in either its premise or its conclusion.

Proof Rules for Values and Effect-Free Programs. First we illustrate the rules in Figure 2, which are for programs without side-effects (which are identical with those for the pure polymorphic PCFv, studied in [25]).

[*Var, Const, Op*] [*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*]. For arithmetical/boolean operations, we have, for example:

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

Similarly for the equality operator:

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

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.

[*Rec*] Under A, the program satisfies, without assuming anything about x, already $B(0)$ (where $B(\ \cdot\ )$ is a formula with a hole). The program 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. The presented rule uses mathematical induction. The rule can easily be extended to the form which uses well-founded induction. There is another rule for recursion which is applicable to a special class of formulae (which are called *admissible formulae* and which correspond to partial correctness). This rule will be discussed in Section 5.1.

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

[*T-Abs, T-App, T-Pack*] These are the rules for the second order polymorphism. [*T-Abs*] and [*T-App*] are similarly explained as [*Abs*] and [*App*] which are defined later, respectively. See also [25]. The rule [*T-Pack*] says that, if $V^{\alpha[\beta/\mathrm{X}]}$, named as v, satisfies $B[\langle \beta, v^{\alpha[\beta/\mathrm{X}]}\rangle/u]$, then $\mathtt{pack}\,\langle \beta, V\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha$ 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 V of β by replacing a pack of its anchor name v and X by fresh anchor name u (note the essentially same technique is used in the rule for paring, [*Pair*]). We can also use the entailment $\forall \mathrm{X}.A \supset A[\alpha/\mathrm{X}]$ and $A[\alpha/\mathrm{X}] \supset \exists \mathrm{X}.A$.

[*T-Rec*] In the rule for recursive types, $A \approx A'$ denotes, as before, two formulae are identical except for substituting occurrences of $\approx$-related types.

Proof Rules for Programs with Side Effects. Next we illustrate the rules in Figure 3, which are for programs with side effects. These rules are in precise correspondence with the typing rules (except the last four structural rules). Among them, **[*Abs*]** is the rule which turns a judgement for programs with side effects to a judgement for programs without side effects. **[*Promote*]** does the converse.

[*Abs*] says that if, under the assumptions A (which does not talk about x) and A_1 (which is about x), M, named m, starts evaluating from an initial state C, always returns a result and reaches a state which together satisfy C', then $\lambda x.M$ named as u satisfies the same property as M which is now represented as an evaluation formula replacing M with the

Fig. 3 Proof Rules (2): Programs with Side Effects

$$[Abs]\ \frac{\{C^{\text{-}x} \wedge A^{\text{-}x} \wedge A_1^x\}\ M :_m \{C'\} \quad \forall x.(A_1 \supset \{C\}\ u \bullet x \searrow m\ \{C'\}) \supset B}{A \vdash \lambda x.M :_u B} \qquad [App]\ \frac{\{C\}\ V :_v \{C_1\} \quad \{C\}\ W :_w \{C_2\} \quad C_1 \wedge C_2 \supset \{C\}\ v \bullet w \searrow u\ \{C'\}}{\{C\}\ VW :_u \{C'\}}$$

$$[If]\ \frac{\{C\}\ V :_b \{A\} \quad \{C \wedge A[\mathsf{T}/b]\}\ M_1 :_u \{C'\} \quad \{C \wedge A[\mathsf{F}/b]\}\ M_2 :_u \{C'\}}{\{C\}\ \mathtt{if}\ V\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u \{C'\}}$$

$$[Case]\ \frac{\{C\}\ V :_v \{C_1\} \quad \{C^{\text{-}\vec{x}} \wedge C_1[\mathtt{in}_i(x_i)/v]\}\ M_i :_m \{C'\}}{\{C\}\ \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[Assign]\ \frac{\{C\}\ V :_v \{A\} \quad (C[i/x] \wedge A[!x/v]) \supset C' \quad i \text{ fresh}}{\{C\}\ x := V :_u \{C'\}} \qquad [Deref]\ \frac{C \supset C'[!x/u]}{\{C\}\ !x :_u \{C'\}}$$

$$[Unpack]\ \frac{\{C\}\ V :_v \{\exists \mathrm{X}.C_1\} \quad \{C^{\text{-}x\mathrm{X}} \wedge C_1[\langle \mathrm{X}, x\rangle/v]\}\ N :_u \{C'\}}{\{C\}\ \mathtt{unpack}\ V\ \mathtt{as}\ \langle \mathrm{X}, x\rangle\ \mathtt{in}\ N :_u \{C'\}}$$

$$[Let]\ \frac{\{C\}\ M :_x \{C_1\} \quad \{C_1\}\ N :_u \{C'\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C'\}}$$

$$[Promote]\ \frac{A \vdash V^{\Gamma;\rho} :_v B \quad A \wedge B \supset A'}{\{A\}\ V^{\Gamma;\rho;\emptyset} :_v \{A'\}} \qquad [Weak]\ \frac{\{C\}\ M^{\Gamma;\alpha;\varphi} :_m \{C'\} \quad \varphi \cap \varphi' = \emptyset}{\{C \wedge C_1^{\varphi'}\}\ M^{\Gamma;\alpha;\varphi\cup\varphi'} :_m \{C' \wedge C_1^{\varphi'}\}}$$

$$[Aux_\forall]\ \frac{\{C_0^{\text{-}i}\}\ M :_u \{C\}}{\{C_0\}\ M :_u \{\forall i.C\}} \qquad [Aux_\exists]\ \frac{\{C\}\ M :_u \{C_0^{\text{-}i}\}}{\{\exists i.C\}\ M :_u \{C_0\}}$$

$$[Consequence]\ \frac{C \supset C_0 \quad \{C_0\}\ M :_u \{C_0'\} \quad C_0' \supset C'}{\{C\}\ M :_u \{C'\}}$$

call $u \bullet x$. Since B is weaker than this specification. it can be safely guaranteed as the behaviour of this abstraction.

[*App*] says that: assume a function V, named v, satisfies C_1 whenever C is satisfied for the environment. Assume further an argument W satisfies C_2 under the same condition C for the environment. Then if $C_1 \wedge C_2$ (which combines the condition on both the function and the argument) implies that an evaluating $v \bullet w$ from the initial state satisfying C will surely terminate with a resulting value (named u) and a final state which together satisfy C', then VW named as u should satisfy the same condition.

[*If*] says that, under C, if a boolean program V (named as b) satisfies A, and C' holds for (1) M_1 under the assumption A holds and (2) M_2 under the assumption A does not hold, then, again under C, surely C' holds for $\mathtt{if}\, V\, \mathtt{then} M_1\, \mathtt{else} M_2$. Note the rule keeps clean symmetry, as in the corresponding rule in Hoare logic. Observe also $b \notin \mathsf{fv}(C)$ by the well-formedness of $\{C\}V :_b \{A\}$. Another observation is that $\{C\}V :_b \{A\}$ indicates (as far as C is non-trivial) V terminates. Thus, in a closed program, the conditional branch can surely be evaluated, reaching one of M_i, which in turn is guaranteed to terminate and satisfies C'.

[*Assign*] uses the substitution $C_1[!x/v]$ which stands for replacing all occurrences of v by $!x$. This rule says for $x := V$ (which surely terminates) to ensure the final state C, C should come from $C_1[!x/v]$ for its x-part (which is about V) together with C on other variables. x is replaced by i which is fresh (so that $i \notin \mathsf{fv}(A,C,C')$) in the pre-condition to deduce C', so that x becomes irrelevant in C' (since the content of x is to be overwritten). The well-known backward rule for assignment in Hoare Logic is derivable from this rule in combination with other rules.

[*Deref*] says that, in essence that names in the logic and program variables work in similar ways. More precisely, for the content of a reference x, named as u, to satisfy C', x itself should satisfy a property stronger than C' which is now about x via the substitution $[!x/u]$.

[*Unpack*] says that, if V named as v satisfies $\exists \mathrm{X}.C_1$ and N named as u satisfies C' with v packed as $\langle \mathrm{X}, x\rangle$, then the resulting program satisfies C'. Unpacking is simply handled by an instantiation of a pack of x and X. Intuitively the abstraction is ensured by linking one name in a pack by another name in another pack.

[*Let*] states that, if evaluation of M (named x) that starts from (a state satisfied by) C reaches a state and a result which together satisfy C_1, and if the evaluation of N (named u) that starts from C surely reaches C', then $\mathtt{let}\ x = M\ \mathtt{in}\ N$ (also named u) can start from C to reach the result and state satisfying C'.

[*Promote*] corresponds to the typing rule with the same name in Figure 1, turning the judgement for an effect-free program to that for a general one. The new post-condition combines the original rely and guarantee conditions. This is not so significant in the logic for global state, but becomes indispensable when local state is present.

[*Weak*] uses effect information to weaken a formula, corresponding to the typing rule with the same name. The use of "constancy rule" in Hoare Logic is often reducible to this rule. Note $\varphi \cap \varphi' = \emptyset$ ensures M can never read or write on weakened name.

[*Consequence, Aux-$\forall$, Aux-$\exists$*] These structural rules are immediately justifiable from the semantics of judgement. For (Consequence), the entailment involves calculation of validity w.r.t. inhabitations in a formula, in addition to that of number theory. In practice, we often only need to use the standard inference rules (e.g. those of the predicate calculus with equality).

This concludes the illustration of the proof rules for global state. As a fundamental property of the proof system, we observe:

Theorem 20. (soundness of proof system, global state)

1. *$A \vdash M^{\Gamma;\rho} :_u B$ implies $A \models M^{\Gamma;\rho} :_u B$.*
2. *$\vdash \{C\} M^{\Gamma;\rho;\varphi} :_u \{C'\}$ implies $\models \{C\} M^{\Gamma;\rho;\varphi} :_u \{C'\}$.*

Corollary 21. *If $\{\top\}\ M :_u \{C\}$ with M closed for some C then we always $M \Downarrow$.*

Proof. Immediate from Theorem 20 and semantics of judgements. $\square$

The proof of Theorem 20 will be given at the end of the present section, after presenting a few reasoning examples.

3.5 Reasoning Examples

In this subsection we explore the small reasoning examples. First we derive a familiar first order Hoare-Logic rule for the assignment. We call M *simple* if M is defined by:

$$M ::= \mathbf{c} \mid !x \mid \mathbf{op}(M_1, ..., M_n)$$

Then we can derive the following rule by the structure induction on M and [*Assign*].

$$[\mathit{Ass\text{-}Simple}]\ \frac{M \text{ simple}}{\{C[M/!x]\}x := M\ \{C\}}$$

(see § 10.6 in [23] for admissibility of this rule). This rule can quickly infer assignments involving simple terms as follows.

1.	$\{!x\times 2=6\}\ y :=\ !x \times 2\ \{!y=6\}$	(Ass-simple)
2.	$!x = 3 \quad \supset \quad !x\times 2 = 6$	
3.	$\{!x=3\}\ y :=\ !x \times 2\ \{!y=6\}$	(1,2, Conseq)

We next consider the assignment of a higher-order program (hence we can no longer use [*Ass-Simple*]), taking Example 8 (2). Below we assume the typing $x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat})$ and let $\mathit{Double}(u) \stackrel{\text{def}}{=} \forall i.(u \bullet i = i \times 2)$.

1.	$z = i \vdash z \times 2 :_m m = i \times 2$	(Var, Num, $\times$)
2.	$\mathsf{T} \vdash \lambda z.(z \times 2) :_v \mathit{Double}(v)$	(Abs)
3.	$\{\mathsf{T}\}\, x := \lambda z.(z \times 2)\, \{\mathit{Double}(!x)\}$	(Promote, Assign)

where we set $A = \mathsf{T}$, $B = \mathit{Double}(v)$ and $C = \mathit{Double}(!x)$ in [*Assignment*]. Next we infer the function with dereference with type $\mathsf{Nat} \stackrel{x}{\Rightarrow} \mathsf{Nat}$.

1.	$\{\mathit{Double}(!x)\}\ !x :_m \{\mathit{Double}(m)\}$	(Deref)
2.	$\{\mathit{Double}(!x)\}\ !x :_m \{m \bullet 3 = 6\}$ by $\mathit{Double}(m) \supset m \bullet 3 = 6$	
3.	$\{y = 3\}\, y :_n \{n = 3\}$	(Var)
4.	$\{\mathit{Double}(!x) \wedge y = 3\}\ (!x)y :_u \{\mathit{Double}(!x) \wedge u = 6\}$	(App)
5.	$\mathsf{T} \vdash \lambda y.(!x)y :_u \{\mathit{Double}(!x)\}\, u \bullet 3 \searrow 6\, \{\mathit{Double}(!x)\}$	(Abs)
6.	$\{\mathit{Double}(!x)\}(\lambda y.(!x)y)3 :_u \{u = 6 \wedge \mathit{Double}(!x)\}$	(App)

To derive Line 4, let $C \stackrel{\text{def}}{=} \mathit{Double}(!x) \wedge y = 3$, $C_1 \stackrel{\text{def}}{=} m \bullet 3 = 6$ and $C_2 \stackrel{\text{def}}{=} n = 3$. Then we have:

$$C_1 \wedge C_2 \quad \supset \quad \{C\}m \bullet n \searrow u\{\mathit{Double}(!x) \wedge u = 6\}$$

so we can apply [*App*]. To derive Line 5, we set $A \stackrel{\text{def}}{=} \mathsf{T}$, $A_1 \stackrel{\text{def}}{=} y = 3$, $C \stackrel{\text{def}}{=} \mathit{Double}(!x)$, $C' \stackrel{\text{def}}{=} m = 6 \wedge \mathit{Double}(!x)$, then we derive

$$(A_1 \supset \{C\}u \bullet y \searrow m\{C'\}) \quad \supset \quad \{C\}u \bullet 3 \searrow m\{C'\}$$

Line 6 is derived with the similar reasoning as Line 4. Next we introduce the derived rule for $M;N$.

$$[Seq]\ \frac{\{C\}\, M\, \{C_1\} \quad \{C_1\} N :_u \{C'\}}{\{C\} M;N :_u \{C'\}}$$

Now we plug-in the post and pre-conditions in the last line of the second and third inferences as:

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

By a similar reasoning, we have:

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

which corresponds to assertion C in Example 8 (5). Similarly we can derive:

$$\mathsf{T} \vdash \lambda y.(!x)y :_u \{C \wedge\ !w = 0\}\, u \bullet 3 \searrow 6 \{C \wedge\ !w = 1\}$$

which corresponds to assertion D in Example 8 (5). Finally we have:

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

We can also derive the total correctness rule of the while loop [15] as well as the rule for multiple recursion as admissible rules. See § 10.6 in [23].

3.6 Soundness with Global State

In this subsection we prove Theorem 20 by rule induction on proof rules. We use a couple of lemmas. Below an *injective renaming* is an injective function on the set of names. In addition to names, we apply the injective renaming to environments, expressions, formulae, assertions, models and interpretations, usually in postfix form.

Lemma 22. *Let* $\Gamma, \Theta \vdash A : \alpha$ *and assume given* $\Gamma \vdash \mathcal{M}$, $\Theta \vdash I$.

1. (thinning) *Assume* x *does not occur in the domains of* Γ *and* Θ. *Then*

$$\mathcal{M} \cdot x : V \models^{I} A \quad \textit{implies} \quad \mathcal{M} \models^{I} A$$

2. (weakening) *For all* $\Gamma' \vdash \mathcal{M}'$ *and* $\Theta' \vdash I'$ *such that* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \mathsf{dom}(\Theta) \cap \mathsf{dom}(\Theta') = \emptyset$,

$$\mathcal{M} \models^{I} A \quad \textit{implies} \quad \mathcal{M} \cdot \mathcal{M}' \models^{I \cdot I'} A$$

Lemma 23. 1. (injective renaming) *For all injective renamings* ξ,

$$\mathcal{M} \models^{I} C \quad \textit{iff} \quad \mathcal{M}\xi \models^{I\xi} C\xi$$

Note that $\mathcal{M}$ *on the right of the equivalence is typed under* $\Gamma\xi$ *and* $I\xi$ *under* $\Theta\xi$.

2. (iso) $C \approx C'$ *and* $\mathcal{M} \models^{I} C$ *imply* $\mathcal{M} \models^{I} C'$.

Lemma 24. (substitution) *Assume* $x \notin \mathsf{fv}(e) \cap \mathsf{dom}(\mathcal{M})$.

$$\mathcal{M} \models^{I} C[e/x] \quad \textit{iff} \quad \mathcal{M} \cdot x : [\![e]\!]\mathcal{M} \cdot I \models^{I} C$$

Similarly, if $\mathrm{X} \notin \mathsf{ftv}(\alpha) \cap \mathsf{dom}(I)$,

$$\mathcal{M} \models^{I} C[\alpha/\mathrm{X}] \quad \textit{iff} \quad \mathcal{M} \models^{I \cdot \mathrm{X} \mapsto \alpha} C$$

Convention 25. Since substitution into type variables, I_t, is not significant for most of cases, we only demonstrate its need in [*T-App,T-Pack*]; in other cases, we just need to replace ξ by $\xi \cdot I_t$, hence we omit I_t in the proofs.

We begin with [*Var*]. We prove:

$$A[x/u] \ \models V :_u A$$

By definition, this is equivalent to prove: for each I and ξ, whenever $\xi \models^{I} A[x/u]$, we have $x\xi \Downarrow V$ such that $u{:}V \models^{I} A$. We define $\xi = \xi', x : W$. Then we know $x\xi = W$. Hence we prove:

$$\xi', x : W \models^{I} A[x/u] \quad \text{implies} \quad u : W \models^{I} A$$

First we note that x is the sole primary name in $A[x/u]$. Hence applying the thinning (Lemma 22(1)), we have:

$$\xi', x : W \models^{I} A[x/u] \text{ implies } x : W \models^{I} A[x/u]$$

From this, using injective renaming (Lemma 23(1)), we have:

$$(x : W)[u/x] \models^{I} A[x/u][u/x]$$

which implies $u : W \models^{I} A$, as required.

Next we prove the case [*Op*]. For simplicity we'll demonstrate the proof using the binary addition function. [*Const*], [*Pair*], [*Proj*$_i$] and [*In*$_i$] are similarly proved. Assume

$$A \vdash V_1 + V_n :_u B \quad \text{and} \quad \xi \models^{I} A.$$

Let $(V_1 + V_2)\xi \stackrel{\text{def}}{=} V_1\xi + V_2\xi \Downarrow n$. Our aim is to prove:

$$u : n \models^{I} B.$$

By the inductive hypothesis, with $\xi \models^{I} A$, we know:

$$v_i : V_i\xi \models^{I} B_i$$

By assumptions, $v_1 \neq v_2$, so with thinning we infer

$$\xi' \models^{I} B_i$$

where $\xi' = v_1 : V_1\xi, v_2 : V_2\xi$. By definition:

$$\xi' \models^{I} B_1 \wedge B_2$$

From the rightmost premise of the rule, $B_1 \wedge B_2 \supset B[v_1 + v_n/u]$, we infer:

$$\xi' \models^I B[v_1 + v_2/u]$$

Applying Lemma 24 gives

$$\xi', u : [\![v_1 + v_2]\!]\xi' \cdot I \models^I B$$

where $u \notin \{v_1, v_2\}$. Since $[\![v_1 + v_2]\!]\xi' \cdot I = V_1\xi + V_2\xi = n$, we reach:

$$\xi', u : n \models^I B$$

But u is B's sole primary name, so we can apply thinning once more and obtain:

$$u : n \models^I B.$$

Next we prove [*Rec*]. First define $W \stackrel{\text{def}}{=} (\lambda y.M)[\mu x.\lambda y.M/x]$, $\{\vec{z}\} \stackrel{\text{def}}{=} \mathsf{fv}(M) \setminus \{x, y\}$ and $V_i \stackrel{\text{def}}{=} \xi(z_i)$. Note that

$$(\lambda y.M)[W/x] \;\cong\; (\mu x.\lambda y.M)$$

Hence we shall show, for all I and ξ, whenever $\xi \models^I A$,

$$u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(n) \text{ for all } n.$$

For the inductive base we reason using thinning and the outer (IH).

$$\begin{aligned} \xi \models^I A^{\text{-}x} &\Rightarrow \xi, x : W \models^I A \\ &\Rightarrow u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(0). \end{aligned}$$

For the inductive step we have:

$$\begin{aligned} u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(n) &\Rightarrow \xi, u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(n) \\ &\Rightarrow \xi, x : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(n)[x/u] \qquad (3.1) \\ &\Rightarrow u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I B(n+1) \end{aligned}$$

Here implication (3.1) is thinning and (3.1) uses injective renaming. The last implication is just the outer (IH). Thus we have shown

$$u : (\lambda y.M)[\vec{V}/\vec{z}][W/x] \models^I \forall n.B(n),$$

as required.

For [*T-Abs*] we start with defining $V' \stackrel{\text{def}}{=} V[\vec{W}/\vec{y}]$ where $\xi(y_i) = W_i$. Then we prove for all I and ξ, whenever $\xi \models^I A^{\text{-X}}$, we have:

$$v : \forall \mathrm{X}.V' \models^I \forall \mathrm{X}.B[u \bullet [\mathrm{X}]/v]$$

First by (IH), we have:

$$\xi \models^I A^{\text{-X}} \;\Rightarrow\; v : V' \models^I B$$

Then we can use weakening to have:

$$u : \Lambda \mathrm{X}.V', v : V' \models^{I} B$$

Next we use Lemma 24 to obtain:

$$u : \Lambda \mathrm{X}.V', v : V'[\alpha/\mathrm{X}] \models^{I,\mathrm{X}\mapsto\alpha} B$$

But clearly $V'[\alpha/\mathrm{X}] = [\![u \bullet [\mathrm{X}]]\!](\xi, u : \Lambda\mathrm{X}.V) \cdot (I, \mathrm{X} \mapsto \alpha)$, so in fact

$$u : \Lambda \mathrm{X}.V' \models^{I,\mathrm{X}\mapsto\alpha} B[u \bullet [\mathrm{X}]/v]$$

by Lemma 24. Hence by definition of $\mathcal{M} \models \forall \mathrm{X}.C$, we have the required result.

Next we prove [*T-App*]. Let $\xi = \vec{z} : \vec{W}$ and $I_t = \vec{\mathrm{Y}} : \vec{\alpha}$ with $I = I_a \cdot I_t$. Let $\xi' = \xi \cdot I_t$. We prove for each I and ξ,

$$\xi \models^{I} A \text{ implies } u : (V\beta)\xi' \models^{I} B[\beta I_t/\mathrm{X}]$$

We have:

$$\begin{aligned}
\xi \models^{I} A &\Rightarrow v : V\xi' \models^{I} \forall \mathrm{X}.B[v \bullet [\mathrm{X}]/u] && (3.2)\\
&\Rightarrow v : V\xi' \models^{I,\mathrm{X}\mapsto\beta I_t} B[v \bullet [\mathrm{X}]/u] && (3.3)\\
&\Rightarrow v : V\xi', u : [\![v \bullet [\mathrm{X}]]\!]_{v:V\xi'\cdot(I\cdot\mathrm{X}\mapsto\beta I_t)} \models^{I,\mathrm{X}\mapsto\beta I_t} B && (3.4)\\
&\Rightarrow v : V\xi', u : (V\beta)\xi' \models^{I,\mathrm{X}\mapsto\beta I_t} B && (3.5)\\
&\Rightarrow u : (V\beta)\xi' \models^{I} B[\beta I_t/\mathrm{X}] && (3.6)
\end{aligned}$$

Here (3.2) is by (IH), (3.3) is from the definitions, (3.4) is an application of Lemma 24.

Next is [*T-Pack*]. Let $\xi = \vec{z} : \vec{W}$ and $I_t = \vec{\mathrm{Y}} : \vec{\alpha}$ with $I = I_a \cdot I_t$. Let $\xi' = \xi \cdot I_t$. We prove for each I and ξ,

$$\xi \models^{I} A \text{ implies } u : (\mathtt{pack}\,\langle\beta, V\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha)\xi \models^{I} \exists\mathrm{X}.B$$

Then we have:

$$\begin{aligned}
\xi \models^{I} A &\Rightarrow v : V\xi' \models^{I} B[\langle\mathrm{X}, v\rangle/u][\beta I_t/\mathrm{X}] && (3.7)\\
&\Rightarrow v : V\xi' \models^{I,\mathrm{X}\mapsto\beta I_t} B[\langle\mathrm{X}, v\rangle/u] && (3.8)\\
&\Rightarrow v : V\xi', u : [\![\langle\mathrm{X}, v\rangle]\!]_{v:V\xi\cdot(I\cdot\mathrm{X}\mapsto\beta I_t)} \models^{I,\mathrm{X}\mapsto\beta I_t} B && (3.9)\\
&\Rightarrow v : V\xi', u : \mathtt{pack}\,\langle\beta I_t, V\xi'\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha I_t \models^{I,\mathrm{X}\mapsto\beta I_t} B\\
&\Rightarrow u : (\mathtt{pack}\,\langle\beta, V\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha)\xi' \models^{I,\mathrm{X}\mapsto\beta I_t} B && (3.10)\\
&\Rightarrow u : (\mathtt{pack}\,\langle\beta, V\rangle\,\mathtt{as}\,\exists\mathrm{X}.\alpha)\xi' \models^{I} \exists\mathrm{X}.B && (3.11)
\end{aligned}$$

Here (3.7) is by (IH), (3.8) and (3.9) are by Lemma 24 since $u \neq v$, $u \notin \langle\mathrm{X}, v\rangle$ by our convention on anchors and $\mathrm{X} \notin \mathsf{ftv}(\alpha) \cap \mathsf{dom}(I)$ by the variable convention. Finally (3.10) follows by thinning as $v \notin \mathsf{fv}(B)$.

[*T-Rec*] is immediate.

For [*Abs*]. Assume $\mathcal{M} \models^I C^{\text{-}x} \wedge A^{\text{-}x} \wedge A_1^x$ with $\mathcal{M} = (\xi, \sigma)$. Then we prove:

$$u : (\lambda x.M)\xi \models^I B$$

with $\forall x.\,(A_1 \supset \{C\}\ u \bullet x \searrow m\ \{C'\}) \supset B$. By (IH) and noting $x \notin \mathsf{fv}(A) \cup \mathsf{fv}(C)$, we know, under the assumption $\xi \models^I A \wedge x : V\xi \models^I A_1 \wedge \sigma \models^I C$, we have:

$$(M[V/x]\xi, \sigma)\ \Downarrow (W, \sigma') \text{ s.t. } (m : W, \sigma') \models^I C'$$

Note that $((\lambda x.M)V)\xi, \sigma)\ \Downarrow (W, \sigma')$. From this and by definition of the satisfaction, under the assumption $\xi \models^I A$, we have:

$$x : V\xi \models^I A_1 \supset \mathcal{M} \cdot x : V\xi \cdot u : \lambda x.M\xi \models^I \{C\}\ u \bullet x \searrow m\ \{C'\}$$

Then by definition of the satisfaction, we have:

$$\mathcal{M} \cdot u : \lambda x.M\xi \models^I \forall x.(A_1 \supset \{C\}\ u \bullet x \searrow m\ \{C'\})$$

Since u is the sole primary name, the thinning gives us:

$$u : \lambda x.M\xi \models^I \forall x.(A_1 \supset \{C\}\ u \bullet x \searrow m\ \{C'\})$$

Finally by $\forall x.(A_1 \supset \{C\}\ u \bullet x \searrow m\ \{C'\}) \supset B$, we have the required result.

Next we prove [*App*]. We assume $\Gamma \vdash V : \alpha \overset{\varphi}{\Rightarrow} \beta$ and $\Gamma \vdash W : \alpha$. Assume

$$\mathcal{M} = (\xi, \sigma_\varphi \cdot \sigma)$$

where $\mathsf{dom}(\sigma_\varphi) = \varphi$. Then if

$$((VW)\xi, \sigma_\varphi \cdot \sigma) \Downarrow (V', \sigma'_\varphi \cdot \sigma)$$

(where the substitution ξ makes VW appropriately semi-closed) then we have to show

$$(u : V', \sigma'_\varphi, \sigma) \models^I C'$$

By the (IH) and weakening, we have

$$(\xi \cdot v : V\xi, \sigma_\varphi \cdot \sigma) \models^I C_1$$
$$(\xi \cdot w : W\xi, \sigma_\varphi \cdot \sigma) \models^I C_2$$

so the definition of the satisfaction of $\wedge$ delivers

$$\mathcal{M}' \overset{\text{def}}{=} (\xi \cdot v : V\xi \cdot w : W\xi, \sigma_\varphi \cdot \sigma) \models^I C_1 \wedge C_2$$

By the other premise then

$$\mathcal{M}' \models^I \{C\}\ v \bullet w \searrow u\ \{C'\}$$

Clearly $[\![v]\!]\mathcal{M}' \cdot I = V\xi$ and $[\![w]\!]\mathcal{M}' \cdot I = W\xi$. Then by the definition of the satisfaction (noting $\mathcal{M} \setminus \varphi = (\xi, \sigma)$), we must have

$$(\xi, v : V\xi \cdot w : W\xi \cdot u : V', \sigma'_{\varphi} \cdot \sigma) \models^{I} C'$$

Finally by thinning, we have

$$(u : V', \sigma'_{\varphi} \cdot \sigma) \models^{I} C'$$

because neither v nor w can occur in C'.

[*If*] and [*Case*] are mechanical and similar to the proofs of (1).

Next case is [*Assign*]. We'll prove it for the following slightly more general form of the rule.

$$\frac{\{C\}\ V :_{v} \{A\} \quad (C[i/x] \wedge A[!x/v]) \supset C'[()/u] \quad i \text{ fresh}}{\{C\}\ x := V :_{u} \{C'\}}$$

With $\mathcal{M} = (\xi, \sigma \cdot x \mapsto W)$, assume $\mathcal{M} \models^{I} C$. Then we prove:

$$(\xi \cdot u : (), x \mapsto V\xi \cdot \sigma) \models^{I} C'$$

We derive:

$$\begin{aligned}
\mathcal{M} \models^{I} C &\Rightarrow (\xi, \sigma \cdot x \mapsto W \cdot i \mapsto W) \models^{I} C && (3.12)\\
&\Rightarrow (\xi, \sigma \cdot i \mapsto W) \models^{I} C[i/x] && (3.13)\\
&\Rightarrow (\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^{I} C[i/x] && (3.14)
\end{aligned}$$

Here (3.12) and (3.13) are weakening, which is applicable as i, is fresh in the former and x in the latter application; (3.14) is Lemma 24. By (IH) and $\mathcal{M} \models^{I} C$ we get

$$\begin{aligned}
(\xi, \sigma \cdot v \mapsto V\xi \cdot x \mapsto W) \models^{I} A &\Rightarrow (\xi, \sigma \cdot x \mapsto V\xi) \models^{I} A[!x/v] && (3.15)\\
&\Rightarrow (\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^{I} A[!x/v] && (3.16)\\
&\Rightarrow (\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^{I} C[i/x] \wedge A && (3.17)\\
&\Rightarrow (\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^{I} C'[()/u] && (3.18)\\
&\Rightarrow (\xi \cdot u : (), \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^{I} C' && (3.19)\\
&\Rightarrow (u : (), \sigma \cdot x \mapsto V\xi) \models^{I} C' && (3.20)
\end{aligned}$$

Here (3.15) is by Lemma 24. (3.16) is weakening. (3.17) uses (3.14). (3.18) comes from the second premise $(C[i/x] \wedge A[!x/v]) \supset C'[()/u]$. Then (3.19) is by the substitution lemma. The last inference is an application of thinning, which is permissible as i does not occur in C'.

For [*Deref*] assume $\mathcal{M} = (\xi, \sigma \cdot x \mapsto V)$. and

$$(!x, \sigma \cdot x \mapsto V) \Downarrow (V, \sigma \cdot x \mapsto V)$$

Then we prove:

$$(u : V, \sigma \cdot x \mapsto V) \models^I C'$$

We observe:

$$\begin{aligned}
\mathcal{M} \models^I C \Rightarrow \mathcal{M} \models^I C'[!x/u] && (3.21)\\
\Rightarrow (\xi \cdot u : [\![!x]\!]\mathcal{M} \cdot I, \sigma \cdot x \mapsto V) \models^I C' && (3.22)\\
\Rightarrow (\xi \cdot u : V, \sigma \cdot x \mapsto V) \models^I C'
\end{aligned}$$

Here (3.21) follows from the premise of the rule, (3.22) is a direct consequence of Lemma 24 because u does not appear in ξ, V and σ. The final equivalence comes from evaluating $[\![!x]\!]\mathcal{M} \cdot I = V$. Then thinning derives the required result.

[*Unpack*] is similar to [*T-Pack*], hence we omit.

[*Promote*] is obvious by (IH) and the definition of the satisfaction.

For [*Weak*] let $\mathcal{M} = (\xi, \sigma)$. Then $(M\xi, \sigma) \Downarrow (V, \sigma')$ together with the (IH) implies

$$(\xi \cdot m : V, \sigma') \models^I C'. \tag{3.23}$$

Under the assumption $\varphi \cap \varphi' = \emptyset$ and $\mathcal{M} \models^I C \wedge C_1$, We have to prove

$$(\xi \cdot m : V, \sigma') \models^I C' \wedge C_1$$

Define $\sigma_\varphi \stackrel{\text{def}}{=} \{\sigma(w) \mid w \in \varphi\}$. Then we have:

$$\begin{aligned}
\mathcal{M} \models^I C \wedge C_1 \Rightarrow \mathcal{M} \models^I C_1 && (3.24)\\
\Rightarrow (\xi, \sigma|_{\varphi'}) \models^I C_1 && (3.25)\\
\Rightarrow (\xi, \sigma'|_{\varphi'}) \models^I C_1 && (3.26)\\
\Rightarrow (\xi, \sigma') \models^I C_1 && (3.27)\\
\Rightarrow (\xi \cdot m : V, \sigma') \models^I C_1 && (3.28)
\end{aligned}$$

where $\sigma|_{\varphi'} = \sigma'|_{\varphi'}$ and $\vec{y} \cap \varphi' = \vec{z} \cap \varphi'$. Hence we have (3.26). Then weakening gives (3.27). Now by the definition of $\wedge$, we derive the required result.

The rules [*Consequence*], [$Aux_\forall$] and [$Aux_\exists$] are immediate from the definition of the satisfaction relation and the laws of first-order logic.

Fig. 4 Typing Rule for Local Variable Declaration.

$$[New]\ \frac{\Gamma \vdash V : \alpha \quad \Gamma, x : \mathsf{Ref}(\alpha) \vdash N : \beta, \varphi \quad x \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \mathtt{new\ var}\ x := V\ \mathtt{in}\ N : \beta \setminus x, \varphi \setminus x}$$

4 Incorporating Local State

4.1 Syntax and Reduction

Local variables that can be dynamically created and shared are the source of both great expressive power and semantic intricacies. For example, they have been one of the essential elements of object-oriented programming languages leading to their flexibility and expressive power, starting with Simula [9]. Semantically, once state is hidden, the behaviour of a procedure or an object can no longer be described as a function, leading to difficulties in their precise formal understanding [28, 34, 45]. This section presents a simple logical articulation of local state in the presence of higher-order functions. A central feature of our approach, whose origin lies in logics for mobile processes, is the explicit representation of sharing of hidden state. This leads to a clean stratification of the logic for local state on the basis of that for global state.

In spite of its expressiveness, the construct which introduces local state to our prototypical programming language is quite simple (cf. [14, 42, 50]).

$$M \quad ::= \quad \ldots \quad | \quad \mathtt{new\ var}\ x := V\ \mathtt{in}\ N$$

In $\mathtt{new\ var}\ x := V\ \mathtt{in}\ N$, x binds free occurrences of x in N. Reductions now take the form $(\nu\,\varphi)(M,\sigma) \longrightarrow (\nu\,\varphi')(M',\sigma')$, where $\varphi \subseteq \mathsf{dom}(\sigma)$ and $\varphi' \subseteq \mathsf{dom}(\sigma')$. In this rule, φ indicates hidden (local) references. It induces binding, hence α-equality, in the standard way. By regarding (M,σ) as $(\nu\,\emptyset)(M,\sigma)$, we regain the original forms of reduction $(M,\ \sigma) \longrightarrow (M',\ \sigma')$ (if no hiding is involved) and $M \longrightarrow M'$ (if no effects are involved).

The reduction rules add the following three, each assuming the standard bound name convention.

$$(\nu\,\varphi)(\mathtt{new\ var}\ x := V\ \mathtt{in}\ N, \sigma) \longrightarrow (\nu\,\varphi x)(N, \sigma \cup [x \mapsto V])$$

$$\frac{(M,\sigma) \longrightarrow (M',\sigma')}{(\nu\,\varphi)(M,\sigma) \longrightarrow (\nu\,\varphi)(M',\sigma')}$$

$$\frac{(\nu\,\varphi)(N,\sigma) \longrightarrow (\nu\,\varphi')(N',\sigma')}{(\nu\,\varphi)(\mathtt{let}\ x = N\ \mathtt{in}\ M, \sigma) \longrightarrow (\nu\,\varphi')(\mathtt{let}\ x = N'\ \mathtt{in}\ M, \sigma')}$$

Note the bound name convention demands $x \notin \varphi \cup \mathsf{dom}(\sigma)$ in the first rule and $\varphi \cap \mathsf{fv}(M) = \emptyset$ in the third.

For typing, we add the rule given in Figure 4 where $\beta \setminus x$ delete x from effects in β (we shall later introduce a more refined typing). As a result of this operation, the effects on x as recorded in the typing of N are now hidden from the outside.

4.2 Assertions for Shared Local State (1): Introduction

Let us illustrate key ideas of assertions for local state, incrementally on the basis of the logic discussed in Section 3. starting from a simple function with a global side effect, $V \stackrel{\text{def}}{=} \lambda().x := !x+1; !x$, whose type is $\mathsf{Unit} \stackrel{x}{\Rightarrow} \mathsf{Nat}$. Using the logic in Section 3, we can give its specification as:

$$Inc(u,x) \stackrel{\text{def}}{=} \forall i.\{!x = i\}\, u \bullet () \searrow i+1\, \{!x = i+1\}$$

Next we consider a program with a local declaration.

$$M \stackrel{\text{def}}{=} \mathtt{new\,var}\, x := 0\, \mathtt{in}\, \lambda().x := !x+1; !x \tag{4.1}$$

When this function is called, it creates a local reference x whose state is hidden from the outside. However, this state does affect M's global behaviour. What assertion can accurately describe this behaviour? One may consider using the name hiding operator "νx" (cf. [8, 36]) to declare locality. Thus we may assert:

$$A(u,0) \stackrel{\text{def}}{=} (\nu x)(!x = 0 \,\wedge\, Inc(u,x)) \tag{4.2}$$

Unfortunately, sharing of local references leads to inconsistencies. Consider the following programs:

$$\begin{array}{lllll}
a := M; & b := !a; & \mathtt{let}\, z = (!a)()\, \mathtt{in} & \mathtt{let}\, z' = (!b)()\, \mathtt{in} & z+z' \\
[\text{i}] & [\text{ii}] & [\text{iii}] & [\text{iv}] & [\text{v}] \\
!x = 0 & !x = 0 & !x = 1 & !x = 2 &
\end{array}$$

This program first [i] assigns M to a, then [ii] assigns the content of a to b, invokes [iii] the content of a, then does the same for that of b [iv], and finally [v] adds up the two numbers returned from these invocations. The initial value of x is 0 (at the points [i] and [ii]) then it is incremented at [iii] and [iv], so x's final value is 2 and the answer of this program is 3. Note that x is incremented every time the function V is called. Using the experimental formula (4.2), we may infer:

$$\{\mathsf{T}\}\; a := M\; \{A(!a,0)\} \quad \text{and} \quad \{A(!a,0)\}\; b := !a\; \{A(!a,0) \wedge A(!b,0)\}$$

Since the judgement for $(!a)()$ is easily given, we use weakening to obtain:

$$\frac{\{A(!a,0)\}\,(!a)()\,\{A(!a,0)\}}{\{A(!a,0) \wedge A(!b,0)\}\,(!a)()\,\{A(!a,1) \wedge A(!b,0)\}}$$

which is clearly wrong: b should point to the same reference as a does, but its value has not been updated. So how about defining a predicate in which x *is* shared, as given below?

$$C(n) \stackrel{\text{def}}{=} (\nu x)(!x = n \wedge Inc(!a,x) \wedge Inc(!b,x))$$

Then we can surely write down a correct state change, for example $\{C(0)\}\,(!a)()\,\{C(1)\}$. Unfortunately $C(n)$ allows us to derive conclusions which we do not want. For many reasons we expect laws such as "$A \supset A'$ implies $(\nu x)A \supset (\nu x)A'$" to hold for the hiding operator. But then we easily obtain (via $A = A \wedge A$, $A \wedge B \supset A$ etc.):

$$\begin{aligned} &(\nu x)(!x = 0 \wedge Inc(!a,x) \wedge Inc(!b,x)) \\ \supset\ &(\nu x)(!x = 0 \wedge Inc(!a,x)) \wedge (\nu x)(!x = 0 \wedge Inc(!b,x)) \end{aligned}$$

Again we end up with disparate predicates for a and b, without sharing x, leading to the same inconsistent update. These observations suggest that the only way to have consistent assertions for accurately describing the behaviour of these programs is to record the sharing of local references explicitly. Thus we assert:

$$(\nu x)ab \ \mathsf{in}\ (!x = 0 \wedge Inc(!a,x) \wedge Inc(!b,x)) \tag{4.3}$$

The predicate in (4.3) is called *opening formula*, since it opens a and b (called *subjects*) to specify their behaviour involving a hidden, shared state x (called *hiding*). Inside the opening formula, a reference x is global between a and b (usually recorded as an effect name of types annotating a and b), which however still remain local for global observers. Syntactically (νx) is a binder while a, b are free. The focus of this formula is on the names of two opaque procedures, a and b. By having these names, we obtain a hook to keep track of predicates which share hidden state. Note that the use of names is fundamental for the construction of this formula.

4.3 Assertions for Shared Local State (2): Syntax and Typing

We formally define the assertions and the proof rules for local state. Using the same set of terms, the grammar of formulae is extended as:

$$C \quad ::= \quad .. \mid (\nu \vec{x}^{\vec{\rho}})\vec{y}\ \mathsf{in}\ C$$

In $(\nu \vec{x}^{\vec{\rho}})\vec{y}$ in C, $(\nu \vec{x})$ bind the free occurrences of $\vec{x}$ in C, inducing α-equality. Names in $\vec{x}\vec{y}$ are pairwise distinct. We consider this formula modulo permutation among $\vec{x}$ and those among $\vec{y}$. The type annotation on hidden names is often omitted. Intuitively, we may read $(\nu \vec{x})\vec{y}$ in C as follows. Below the term "opaque name" denotes a name assigned to a program with a hidden state (e.g. a primary name given to M in (4.1)).

> *Opaque names $\vec{y}$ share local references $\vec{x}$, with C specifying the behaviour of (now open) $\vec{y}$ with respect to the internally global reference $\vec{x}$.*

For typing, we use a refined notion of types that distinguishes types given to opaque names from those given to non-opaque ones. While we can use the same type structure as is used so far, a stricter type discipline leads to a simpler and more intuitive semantics of assertions.[3] The grammar of types is extended as follows.

$$\alpha, \beta \ ::= \ \ldots \ \mid \ \alpha \overset{\varphi}{\Rightarrow} [\beta] \qquad \rho \ ::= \ \alpha \ \mid \ \mathsf{Ref}(\alpha) \qquad \delta \ ::= \ \rho \ \mid \ [\rho]$$

We call a type of the form $[\alpha]$ or $[\mathsf{Ref}(\alpha)]$, an *opaque type*. A type which is not opaque is called *open type* or *non-opaque type*. Note the target of an arrow type can be opaque: this is because a function can generate, upon invocation, a closure which has a local

Fig. 5 Typing Rules with Opaque Types.

$$[Abs]\ \frac{\Gamma, x:\alpha \vdash M : [\beta], \varphi}{\Gamma \vdash \lambda x^{\alpha}.M : \alpha \overset{\varphi}{\Rightarrow} [\beta]} \qquad [App]\ \frac{\Gamma \vdash V : \alpha \overset{\varphi}{\Rightarrow} [\beta] \quad \Gamma \vdash W : \alpha}{\Gamma \vdash VW : [\beta], \varphi}$$

$$[Opaque\text{-}P]\ \frac{\Gamma \vdash M : \alpha, \varphi}{\Gamma \vdash M : [\alpha], \varphi} \qquad [Opaque\text{-}E]\ \frac{\Gamma, x:\alpha \vdash M : \delta, \varphi}{\Gamma, x:[\alpha] \vdash M : \delta, \varphi}$$

$$[LetN]\ \frac{\Gamma \vdash N : [\alpha], \varphi_2 \quad \Gamma, x:\alpha \vdash M : [\beta], \varphi_1}{\Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : [\beta], \varphi_1 \cup \varphi_2}$$

$$[New]\ \frac{\Gamma \vdash V : \alpha \quad \Gamma, x : \mathsf{Ref}(\alpha) \vdash N : [\beta], \varphi \quad x \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \mathtt{new\ var}\ x := V\ \mathtt{in}\ N : [\beta \setminus x], \varphi \setminus x}$$

$$[Co\text{-}New]\ \frac{\Gamma^{\text{-}x}, \vec{z}:[\vec{\tau}], x:\mathsf{Ref}(\alpha) \vdash M : \delta^{\text{-}x}, \varphi^{\text{-}x}}{\Gamma, \vec{z}:[\vec{\tau} \setminus x] \vdash M : \delta, \varphi}$$

state. Since the logical treatment of the original arrow type, $\alpha \overset{\varphi}{\Rightarrow} \beta$, is subsumed by that of the new arrow type, hereafter we only consider the latter.

While we may use the same typing for programs as given in §4.1, we introduce a refined typing for programs which makes the subsequent technical development more transparent. The typing rules overwrite/increment those of Figure 1 and Figure 4 with those in Figure 5. [*Opaque-P,E*] turn open types to opaque types in a program/environment. [*Co-New*] is dual to [*New*], hiding a reference in the environment. [*LetN*] refines the original [*Let*]. Note the type of x is open in the second premise.

The typing for terms, Definition 5, remains identical (which among others means no terms are typed with opaque types). For the typing of formulae, we add to Definition 6 the clauses for an evaluation formula with an opaque target type and an opening formula. Below the codomain of Γ is the extended set of types, similarly henceforth.

Definition 26. (well-typedness of formulae, local state) The relation $\Gamma \vdash A$ is generated by the same clauses as in Definition 6 together with the following two added clauses.

- $\Gamma \vdash \{C\}\ e_1 \bullet e_2 \searrow z\ \{C'\}$ if $\Gamma \vdash e_1 : \alpha \overset{\varphi}{\Rightarrow} [\beta]$ and $\Gamma \vdash e_2 : \alpha$ and, moreover, $\Gamma \vdash C$ and $\Gamma, z : [\beta] \vdash C'$.
- $\Gamma \vdash (\nu\vec{x})y_1..y_i..y_n$ in A if, for some $\vec{\rho}$ and $\vec{\alpha}$, we have $\Gamma \vdash y_i : [\rho_i/\vec{x}]$ $(1 \leq i \leq n)$ and $\Gamma/\vec{y},\ \vec{y}:\vec{\rho},\ \vec{x}:\mathsf{Ref}(\vec{\alpha}) \vdash A$.

The first clause in Definition 26 is the refined version of the original clause for evaluation formulae, treating the case when a target type is opaque. The second clause says that, for opening a vector of names with a vector of local reference names, (1) they should be opaquely typed now, and (2) once opened, these names as well as the local reference names should be assigned open types.

Judgements have the identical shape as before, typing programs and assertions with opaque typing and stipulating auxiliary names should always be assigned open types.

[3] An alternative typing framework without a refined type structure is discussed in B.

4.4 Models with Local State

We again use the notion of semi-closed programs, which is defined precisely by the same clause. To wit: a program $\Gamma \vdash M : \alpha, \varphi$ is *semi-closed* (resp. *closed*) when $\mathsf{dom}(\Gamma) = \varphi$ (resp. $\mathsf{dom}(\Gamma) = \varphi = \emptyset$). Similarly $\Gamma \vdash M : \alpha$ is *semi-closed* (resp. *closed*) when $\mathsf{ran}(\Gamma)$ consist of only reference types. Note semi-closed programs should *not* own opaque types, which is a fundamental requirement.

We first define an *open model*, which is a pair (ξ, σ) as we have treated in §3.3. A *model* (ranged over by $\mathcal{M}, \ldots$) is a pair of (1) a finite set of names and (2) an open model, denoted $(\nu\vec{x})(\xi, \sigma)$, where $\{\vec{x}\} \subseteq \mathsf{dom}(\sigma)$ and which is considered up to the α-convertibility induced by the binding by $(\nu\vec{x})$. In a model $\mathcal{M} \stackrel{\text{def}}{=} (\nu\vec{x})(\xi, \sigma)$, the names $\vec{x}$ are its *hiding*, and $\mathsf{dom}(\xi) \cup (\mathsf{dom}(\sigma) \backslash \{\vec{x}\})$ is its *subjects*, the latter denoted $\mathsf{sbj}(\mathcal{M})$. We write (ξ, σ) for $(\nu\emptyset)(\xi, \sigma)$.

We type models starting from the typing for open models in §3.3 which is extended by the following rules.

$$\frac{\Gamma,\ x\!:\!\rho \vdash \mathcal{M}}{\Gamma,\ x\!:\![\rho] \vdash \mathcal{M}} \qquad \frac{\Gamma^{-x}, \vec{y}\!:\![\vec{\rho}], x\!:\![\mathsf{Ref}\,(\alpha)] \vdash \mathcal{M} \quad x \in \mathsf{sbj}(\mathcal{M}) \cap \mathsf{fn}(\rho_i)}{\Gamma,\ \vec{y}\!:\![\vec{\rho}\backslash x] \vdash (\nu x)\mathcal{M}}$$

In the second rule, $(\nu x)\mathcal{M}$ is the model which adds x to the hiding of $\mathcal{M}$ (note x is free in $\mathcal{M}$ by the condition x being a subject in $\mathcal{M}$). Both rules are understood as the typing rules for programs in Figure 5. We use the following conventions/notations.

Notation 27.

1. Given $\mathcal{M} = (\nu\vec{x})(\xi, \sigma)$, we write $\mathcal{M}(y) = V$ when $\xi(y) = V$ and $\mathsf{fv}(V) \cap \{\vec{x}\} = \emptyset$ (such y is called *open*).
2. If $\vec{y}$ is open in $\mathcal{M}$, we write $\mathcal{M} \setminus \vec{y}$ for the result of taking off the $\vec{y}$-part from $\mathcal{M}$.
3. $\mathcal{M}_1 \cdot \mathcal{M}_2$ is the component-wise union of $\mathcal{M}_1$ and $\mathcal{M}_2$, stating at the same time their subjects are disjoint.

Example 28. We illustrate the last two clauses by examples. First, for $\mathcal{M} \setminus xy$, we have, assuming $\{\vec{z}\} \cap \mathsf{fv}(V, W) = \emptyset$:

$$(\nu\vec{z})(\xi \cdot x : V, \sigma \cdot y \mapsto W) \setminus xy = (\nu\vec{z})(\xi, \sigma) \tag{4.4}$$

For the union, we have:

$$(\nu\vec{z}_1)(\xi_1, \sigma_1) \cdot (\nu\vec{z}_2)(\xi_2, \sigma_2) = (\nu\vec{z}_1\vec{z}_2)(\xi_1 \cup \xi_2, \sigma_1 \cup \sigma_2) \tag{4.5}$$

where we set $\vec{z}_1 \cap \vec{z}_2 = \emptyset$ (note that, in forming the union in (4.5), we need to make the hidings disjoint by appropriate α-conversion).

The interpretation of terms is precisely as in Definition 13. For the satisfaction relation, we take I to be the union of an open model and a map from free type variables to closed types. The satisfaction relation is, as before, from typed models to formulae of the same type under I, where the latter is assumed to use semi-closed values which have open types (in correspondence with the convention all auxiliary names should be typed with open types). The definition of $\mathcal{M} \models^I C$ augments the clauses in Definition 15 with three additional clauses. In the third clause below, we define $\Gamma \vdash \mathcal{M}_1 \cong \mathcal{M}_2$ by turning each $\mathcal{M}_i$ into an n-tuple of values with a hiding and applying $\cong$.

Definition 29. (satisfaction for local state) Given $\Gamma \vdash \mathcal{M}\cdot I$ and $\Gamma \vdash C$, the relation $\mathcal{M}^{\Gamma} \vdash^{I} C$ (read: $\mathcal{M}$ *satisfies C under a typing* Γ *and an interpretation* I) is generated from the clauses of Definition 15 (whose $\mathcal{M}^{\Gamma}$ is regarded as a typed model with local state) combined with the following additional clauses.

$$\begin{array}{rcl}
\mathcal{M}^{\Gamma}\models^{I}\{C\}e_1\bullet e_2\searrow x\{C'\} & \text{if} & \forall\sigma.\,(\,\Gamma\restriction\varphi\vdash\sigma\ \wedge\ (\mathcal{M}\setminus\varphi)\cdot\sigma\models^{I} C\\
 & & \supset\ \exists V,\sigma'.\ (([\![e_1]\!]\mathcal{M}\cdot I)([\![e_2]\!]\mathcal{M}\cdot I),\ \sigma)\Downarrow(\nu\vec{y})(V,\sigma')\\
 & & \text{such that } (\mathcal{M}\setminus\varphi)\cdot(\nu\vec{y})(x\!:\!V,\sigma')\models^{I}C'\,)\\
\mathcal{M}^{\Gamma}\models^{I}(\nu\vec{x})\vec{u}\ \mathsf{in}\ C & \text{if} & \mathcal{M}=(\nu\vec{x})\mathcal{M}_0 \text{ and } \mathcal{M}_0^{\Gamma_0}\models^{I} C \text{ where (1) } \Gamma_0\backslash\vec{x}=\Gamma,\\
 & & \text{(2) } \Gamma_0(\vec{x}\vec{u}) \text{ open, and (3) } y\notin\{\vec{x}\vec{u}\}\supset\mathsf{fn}(\Gamma_0(y))\cap\{\vec{x}\}=\emptyset.\\
\mathcal{M}^{\Gamma}\models^{I}C & \text{if} & \mathcal{M}_0^{\Gamma}\models^{I}C\ \wedge\ \Gamma\vdash\mathcal{M}_0\cong\mathcal{M}
\end{array}$$

where, in the first clause, we assume $\Gamma \vdash e_1 : \alpha \overset{\varphi}{\Rightarrow} [\beta]$.

Remark 30. (satisfaction for local state)

1. The first added clause simply augments the corresponding definition in Definition 15 with a local state in the result of evaluation.
2. The second added clause defines the semantics of an opening formula. It says that, for $\mathcal{M}$ to satisfy $(\nu\vec{x})\vec{u}$ in C under the typing Γ and an interpretation of auxiliary names I, $\mathcal{M}$ should have the form where, after taking off $\vec{x}$, both $\vec{x}$ and $\vec{u}$ become open and satisfy C, as well as no other names using $\vec{x}$ as their effect names.
3. In the third clause, the closure under $\cong$ is necessary even though we are interpreting the equality over terms via $\cong$. This is because two programs which have different syntactic shape (especially in the number, types and content of hidden names) can be semantically equivalent: we need to use this equality if we wish to have a desired (and semantically correct) assertion to a program whose syntactic structure does not conform to the assertion. As a simple example, consider the following two programs:

$$M_0 \stackrel{\text{def}}{=} \mathtt{new\ var}\ x := 0\ \mathtt{in}\ \lambda().x := !x+1;\ !x M_1 \stackrel{\text{def}}{=} \mathtt{new\ var}\ x := 1\ \mathtt{in}\ \lambda().x := !x+1;(!x-1) \tag{4.6}$$

Since the visible behaviour of M_0 and M_1 coincide, we have $M_1 \cong M_2$. Thus we wish to have the same set of predicates to hold for M_1 and M_2. For example, an assertion which follows the syntactic structure of M_0, $(\nu u)x$ in $!x = 0 \wedge \forall i^{\mathsf{Nat}}.\{!x = i\}u\bullet()\searrow i+1\{!x = i+1\}$, may as well be satisfied by M_1. This is made possible by the third clause above.

In (2) below, we assume the standard bound name convention, which in particular guarantees $\{\vec{y}\}\cap\mathsf{fn}(\Gamma) = \emptyset$.

Definition 31. (semantics of judgement, local state)

1. $A \models M^{\Gamma;\delta} :_u B$ iff, for each I and ξ, whenever $\xi \models^{I} A$, we have $M I_t \xi \Downarrow V$ such that $u\!:\!V^{u:\delta} \models^{I} B$.
2. $\models \{C\} M^{\Gamma;\delta;\varphi} :_u \{C'\}$ iff, for each I and $\mathcal{M} \stackrel{\text{def}}{=} (\nu\vec{y})(\xi,\sigma)$ such that $\Gamma \vdash \mathcal{M}$, whenever $\mathcal{M}^{\Gamma} \models^{I} C$, we have $(\nu\vec{y})(M I_t\xi,\sigma) \Downarrow (\nu\vec{z})(V,\sigma')$ such that $(\nu\vec{z})(u\!:\!V,\ \sigma')^{\Gamma,\ u:\delta} \models^{I} C'$.

Remark 32. The second clause of Definition 31 refines the corresponding clause of Definition 17 by adding a hidden store, both in the initial state and in the final state.

4.5 Some Laws for Local State

Before moving into the proof rules for local state, we list a few basic logical laws which hold in models. They are often useful in proving various properties of programs with local state. Some of them are reminiscent of the structural rules of the π-calculus [36]. The first one is about a trivial opening (that which has an empty hiding). Below $\equiv$ is used meta-logically.

$$\mathcal{M}^{\Gamma,\vec{x}:[\vec{\rho}]} \models^{I} (\nu\varepsilon)\vec{x} \;\mathsf{in}\; C \quad \equiv \quad \mathcal{M}^{\Gamma,\vec{x}:\vec{\rho}} \models^{I} C \tag{4.7}$$

The equivalence (4.7) is immediate from the definition of the satisfaction of an opening formula (the second clause in Definition 29). Because of (4.7), we often regard $(\nu\varepsilon)\vec{x}$ in C and C (the latter under a typing in which $\vec{x}$ are given open types) as being equivalent under implicit type conversion. So we may infer:

$$\frac{\{C\}M^{\Gamma;\alpha,\varphi} :_u \{C'\}}{\{C\}M^{\Gamma;[\alpha],\varphi} :_u \{(\nu\varepsilon)u \;\mathsf{in}\; C'\}} \qquad \frac{\{C\}M^{\Gamma,x:\rho;\delta,\varphi} :_u \{C'\}}{\{(\nu\varepsilon)x \;\mathsf{in}\; C\}M^{\Gamma,x:[\rho];\delta,\varphi} :_u \{C'\}} \tag{4.8}$$

which is type-wise correct since we can always promote open types to opaque types (cf. Figure 5). On the other hand, the following inference is type-wise problematic, at least with the present typing rules for programs.

$$\frac{\{C\}M^{\Gamma;[\alpha],\varphi} :_u \{(\nu\varepsilon)u \;\mathsf{in}\; C'\}}{\{C\}M^{\Gamma;\alpha,\varphi} :_u \{C'\}} \tag{4.9}$$

This conversion is problematic if M can only have an opaque type because of its syntactic shape, even though it is semantically correct. In practice, however, we may as well consider C' (where u may occur open) above as an abbreviation for $(\nu\varepsilon)u$ in C'. Such an informal treatment can retain accuracy if the underlying formal representation is not forgotten (a ML-like editor may help in such situations).

The following simple equivalence does not need type conversion.

$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\varepsilon \;\mathsf{in}\; C \quad \equiv \quad \mathcal{M}^{\Gamma} \models^{I} C \qquad (\{\vec{x}\} \cap \mathsf{fn}(C) = \emptyset) \tag{4.10}$$

Again the equivalence is immediate from Definition 29. We also have, assuming the standard bound name convention:

$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y} \;\mathsf{in}\; ((\nu\vec{u})\vec{v} \;\mathsf{in}\; C) \quad \supset \quad \mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x}\vec{u})\vec{y}\vec{v} \;\mathsf{in}\; C \tag{4.11}$$

The converse also holds if types annotating $\vec{v}$ (resp. $\vec{u}$) in C, if any, do not contain any x_i (resp. y_j) among their effect names. We move to more non-trivial laws.

$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y} \;\mathsf{in}\; (C_1 \vee C_2) \quad \equiv \quad \mathcal{M}^{\Gamma} \models^{I} ((\nu\vec{x})\vec{y} \;\mathsf{in}\; C_1) \vee ((\nu\vec{x})\vec{y} \;\mathsf{in}\; C_2) \tag{4.12}$$
$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y} \;\mathsf{in}\; (\exists z.\, C) \quad \equiv \quad \mathcal{M}^{\Gamma} \models^{I} \exists z.\, (\nu\vec{x})\vec{y} \;\mathsf{in}\; C \tag{4.13}$$
$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y} \;\mathsf{in}\; \neg C \quad \supset \quad \mathcal{M}^{\Gamma} \models^{I} \neg\, (\nu\vec{x})\vec{y} \;\mathsf{in}\; C \tag{4.14}$$

We leave the verification of these laws to the reader (one point which may deserve a remark is, in $(\nu\vec{x})\vec{y}$ in C, if x_i occurs as an effect of the type annotating y_j in C, then we

Fig. 6 Proof Rules for Local State: Additional Rules

$$[New]\ \frac{\{C^{-x}\}\ V :_v \{C_1\} \quad \{C_1[!x/v]\}\ N :_u \{C' \setminus \nu x\}}{\{C\}\ \mathtt{new\ var}\ x := V\ \mathtt{in}\ N :_u \{C'\}}$$

$$[CoNew]\ \frac{\{C \setminus \nu x\}\ M :_u \{C' \setminus \nu x\}}{\{C\}\ M :_u \{C'\}}$$

$$[LetN]\ \frac{\{C\}\ M :_x \{(\nu \vec{y})x\vec{z}\ \mathsf{in}\ C_1\} \quad \{C_1\}\ N :_u \{C' \setminus \nu \vec{y}\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C'\}}$$

can uniquely determine which it is even under α-conversion: and if not, this name has no consequence in validity). From (4.12) and (4.14), we arrive at:

$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y}\ \mathsf{in}\ (C_1 \wedge C_2) \quad \supset \quad \mathcal{M}^{\Gamma} \models^{I} ((\nu\vec{x})\vec{y}\ \mathsf{in}\ C_1) \wedge ((\nu\vec{x})\vec{y}\ \mathsf{in}\ C_2) \qquad (4.15)$$

The converse does not hold (as a simple example, take $\mathcal{M} \stackrel{\text{def}}{=} u{:}V$, $\Gamma = u{:}\alpha$ and consider an assertion "$(\nu x)\varepsilon\ \mathsf{in}\ !x = 1 \wedge (\nu x)\varepsilon\ \mathsf{in}\ !x = 2$"). We however can have:

$$\mathcal{M}^{\Gamma} \models^{I} (\nu\vec{x})\vec{y}\ \mathsf{in}\ (C_1 \wedge C_2) \quad \equiv \quad \mathcal{M}^{\Gamma} \models^{I} ((\nu\vec{x})\vec{y}\ \mathsf{in}\ C_1) \wedge C_2 \quad (x_i, y_j \notin \mathsf{fv}(C_2)) \quad (4.16)$$

From these laws we can further infer other useful rules as needed, such as distribution over entailment.

4.6 Proof Rules

The additional proof rules for local state are listed in Figure 6, corresponding to the last three typing rules in Figure 5. [*New*] introduces a local state to a program, [*Co-New*] introduces a local state to an environment, and [*LetN*] refines [*Let*] by considering the case when the result of evaluation has a local state. Other rules coincide with the original proof rules in Figure 2 and Figure 3. Before illustrating the three newly introduced proof rules, we present brief observations on remaining rules.

1. In [*Var*] of Figure 2, it is essential that the type of x in the environment is open (as being demanded by its typing rule in Figure 1). In fact, we may as well assume all types (of the environment and of the program) used in the proof rules for values, Figure 2, are open. This does not lose generality while
2. We preserve the original [*Let*] for treating the case when the types of programs are open (though it is in fact subsumed by [*LetN*]). The proof rules of abstraction and application for arrow types with opaque targets are identical with those for the original arrow types, as given in Figure 3.
3. The proof rule corresponding to [*Opaque-P,E*] in Figure 5 are already given in (4.8), §4.5, which, in practice, may as well be treated as logical equivalence under implicit type conversion. cf. (4.7), §4.5.

We now discuss the new rules in Figure 6. Each rule uses an operation which deletes a new name from the hidden names of an opening formulae.

Notation 33. (unhiding) The *unhiding of x in C*, written $C\backslash\nu x$, which demands C has shape $(\nu\vec{a}\,x\,\vec{b})\vec{y}$ in C_0, is defined as: $((\nu\vec{z}\,x\,\vec{w})\vec{y}\text{ in }C_0)\setminus\nu x \stackrel{\text{def}}{=} (\nu\vec{z}\vec{w})\vec{y}\text{ in }C_0$.

Note x in C is a bound occurrence in $C\backslash\nu x$. We fix the shape of C in $C\backslash\nu x$ so that it is guarded, i.e. it starts with an opening formula whose hidden names include x. This may not be restrictive through the use of the laws for opening formulae discussed in §4.5. For example, by (4.12) in §4.5, we have the following equivalence:

$$(C_1 \vee C_2)\backslash\nu x \qquad \equiv \qquad (C_1\backslash\nu x) \vee (C_2\backslash\nu x)$$

This and other transformations may allow us to turn the shape of formulae into guarded ones in most practical cases. We now informally illustrate each rule one by one.

[*New*] may be considered as a combination of [*Assign*] and [*Let*] in Figure 3, integrated with name hiding. The rule reads:

> *Assume V (named v) under assumption C satisfies C_1; and assume the evaluation of N (named u) under the assumption $C_1[!x/v]$ (which means V is stored in x) satisfies $C' \setminus \nu x$ (which means x may be used for specifying N); Then* `new var` $x := V$ `in` *N, named u, satisfies C' with x now hidden, starting from C.*

Note the premise demands x should not occur in C, in accordance with the corresponding typing rule in Figure 5.

[*CoNew*] adds hiding in the pre/post conditions, thus allowing us to transform an assertion derived from the reasoning with unhidden variables to the one with a hidden variable. The rule assumes the typing constraints as given in [*CoNew*] of Figure 5, which is omitted in the rule. It reads:

> *Assume, with a reference x exposed, M named u starts evaluation from an initial state $C\backslash\nu x$ to reach $C'\backslash\nu x$, where the typing (cf. Figure 5) demands x is not referred to from within the program. Then M can also start evaluation from a state of the environment C, with now x hidden, to reach C', again with x hidden.*

Note the underlying typing plays an essential role. This rule (together with [*LetN*]) is significant since the reasoning which uses resources with local state often includes a hypothetical reasoning assuming the resources in the environment are open, followed by a closing of the resources. An example of such reasoning is discussed in §4.7.

[*LetN*] allows the result of evaluating which may contain a local state to be plugged into an assumed open behaviour. The rule is best understood by first referring to the following evaluation rule for `let`, derivable from the reduction rules in §4.1.

$$\frac{(M,\,\sigma) \Downarrow (\nu\,\vec{y})(V,\,\sigma')}{(\texttt{let}\ x = M\ \texttt{in}\ N,\,\sigma) \longrightarrow (\nu\,\vec{y})(N[V/x],\,\sigma')} \tag{4.17}$$

With this rule in mind, the proof rule reads:

Assume the evaluation of M (named x) starting from an initial state C converges and satisfies C_1, where x shares local state $\vec{y}$ with $\vec{z}$; assume further that the evaluation of N (named u) starting from C_1 leads to C' where x is open and $\vec{y}$ are exposed in C and C'. Then, starting from C, the program $\mathtt{let}\ x = M\ \mathtt{in}\ N$ *named u will eventually converge and satisfy C' with $\vec{y}$ now hidden.*

In the light of (4.17), the second premise of [*LetN*], $\{C_1\}\ M :_u \{C' \setminus \nu\vec{y}\}$, corresponds to the open configuration $(M[V/x],\ \sigma')$. In the conclusion, we close it with the hiding of $\vec{y}$, obtaining an assertion on $(\nu\,\vec{y})(N[V/x],\ \sigma')$. Note also, if we have $\vec{y} = \varepsilon$, the rule coincides with [*Let*] in Figure 3.

We shall prove the following result in §4.8, after illustrating the proof rules with several reasoning examples.

Theorem 34. (soundness of proof system, local state)

1. $A \vdash M^{\Gamma;\delta} :_u B$ *implies* $A \models M^{\Gamma;\rho} :_u B$.
2. $\vdash \{C\}\, M^{\Gamma;\delta;\varphi} :_u \{C'\}$ *implies* $\models \{C\}\, M^{\Gamma;\rho;\varphi} :_u \{C'\}$.

4.7 Examples of Reasoning with Local State

A Program with Local Shared Variables We first infer the following program given in § 4.2.

$$N \stackrel{\text{def}}{=} a := M;\ b :=!a;\ \mathtt{let}\ z = (!a)()\ \mathtt{in}\ \mathtt{let}\ z' = (!b)()\ \mathtt{in}\ z+z'$$

with M given in (4.1) in § 4.2. Then we derive:

1.	$\{\mathsf{T}\}\,\lambda().(x :=!x+1; !x) :_u \{Inc(u,x)\}$	(Deref,Ass,Abs)
2.	$\{\mathsf{T}\}\,M :_u \{(\nu x)u \text{ in } (!x = 0 \wedge Inc(u,x))\}$	(New)
3.	$\{\mathsf{T}\}\,a := M\,\{(\nu x)a \text{ in } (!x = 0 \wedge Inc(!a,x))\}$	(LetN,Ass)
4.	$\{!x = 0 \wedge Inc(!a,x)\}\,b := !a\,\{!x = 0 \wedge Inc(!a,x) \wedge Inc(!b,x)\}$	(LetN,Ass,Deref)
5.	$\{!x = 0 \wedge Inc(!a,x) \wedge Inc(!b,x)\}$ $\mathtt{let}\ z = (!a)()\ \mathtt{in}\ \mathtt{let}\ z' = (!b)()\ \mathtt{in}\ z+z' :_u$ $\{u = 3 \wedge !x = 2 \wedge Inc(!a,x) \wedge Inc(!b,x)\}$	(LetN,Deref,App)

Applying the derived structural rule in § 4.3 to the precondition of Line 4, we have:

$$\begin{aligned}(!x = 0 \wedge Inc(!a,x)) &\equiv (\nu\varepsilon)a \text{ in } (!x = 0 \wedge Inc(!a,x)) \\ &\equiv (\nu x)a \text{ in } (!x = 0 \wedge Inc(!a,x)) \setminus \nu x\end{aligned}$$

Similarly, in the postcondition of Line 5, we derive:

$$\begin{aligned}&(!x = 2 \wedge Inc(!a,x) \wedge Inc(!b,x)) \\ &\equiv (\nu x)ab \text{ in } (!x = 0 \wedge Inc(!a,x) \wedge Inc(!b,x)) \setminus \nu x\end{aligned}$$

Then an application of [*CoNew*] gives us:

$$\{(\nu x)a \text{ in } (!x = 0 \wedge Inc(!a,x))\} b := !a; \mathtt{let}\ z = \ldots\ \mathtt{in}\ z + z' :_u$$
$$\{(\nu x)ab \text{ in } (u = 3 \wedge !x = 2 \wedge Inc(!a,x) \wedge Inc(!b,x))\}$$

Finally by *[Seq]*, we obtain:

$$\{\mathsf{T}\} N :_u \{(\nu x)ab \text{ in } (u = 3 \wedge !x = 2 \wedge Inc(!a,x) \wedge Inc(!b,x))\}$$

Destructive List Update with Local Variables Next we present an example of specifying a polymorphic algorithm that computes the length of arbitrary lists using local state. Lists have the type $\mathsf{List}_\mathrm{X} \stackrel{\text{def}}{=} \mu \mathrm{Y}.(\mathsf{Unit} + \mathrm{X} \times \mathrm{Y})$. The algorithm is

$$\mathsf{len} \stackrel{\text{def}}{=} \Lambda \mathrm{X}.\lambda x^{\mathsf{List}_\mathrm{X}}.\mathtt{new}\ \mathtt{var}\ n := 0\ \mathtt{in}\ ((\mathsf{len}_\mathrm{X} x); !n)$$

where

$$\mathsf{len}_\mathrm{X} \stackrel{\text{def}}{=} \mu f^{\mathsf{List}_\mathrm{X} \stackrel{n}{\Rightarrow} \mathsf{Unit}}.\lambda l^{\mathsf{List}_\mathrm{X}}.\mathtt{case}\ l\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i \in \{1,2\}}$$
$$M_2 \stackrel{\text{def}}{=} n := !n + 1; g(\pi_2(x_2)) \qquad M_1 \stackrel{\text{def}}{=} ()$$

This procedure creates a hidden variable n which stores the length of the list processed so far. The list is recursively traversed. At each node the hidden variable is incremented and at the end the stored value is returned. The algorithm meets the following specification.

$$\mathsf{T} \vdash \mathsf{len} :_u \forall \mathrm{X}.\forall l^{\mathsf{List}_\mathrm{X}}.\forall n.(\mathsf{Length}_\mathrm{X}(l,n) \supset u \bullet [\mathrm{X}] \bullet l = n)$$

$\mathsf{Length}_\mathrm{X}(l,0)$ iff $l = \mathtt{in}_1(())$ and $\mathsf{Length}_\mathrm{X}(l,n+1)$ iff $\exists a,l'.(l = \mathtt{in}_2(\langle a,l' \rangle) \wedge \mathsf{Length}_\mathrm{X}(l',n))$, omitting some type information. Clearly, for infinite lists, or for lists containing infinite elements, the specification is vacuously true and hence uninteresting. This specification says: assuming a polymorphic list has length n, applying the type and the list to u would result in the length of the list. It is noteworthy that this specification is purely functional, despite crucial use of local state.

Destructive Tree Update with Local Variables We show an example which use binary trees. We first encode data types for trees using products and sums.

$$\mathsf{tree}_\alpha = \mu \mathrm{X}.(\mathsf{Unit} + (\alpha \times (\mathrm{X} \times \mathrm{X}))) \qquad \mathsf{tree} = \forall \mathrm{X}.\mathsf{tree}_\mathrm{X}$$

The empty tree is represented by $\mathsf{leaf} \stackrel{\text{def}}{=} \mathtt{in}_1(())$ while the tree carrying the datum a of type α in its root node and having left subtree l and right subtree, r is $\langle l,a,r \rangle \stackrel{\text{def}}{=} \mathtt{in}_2(\langle a, \langle l,r \rangle \rangle)$. By $\mathsf{tree}_\alpha \approx \mathsf{Unit} + (\alpha \times (\mathsf{tree}_\alpha \times \mathsf{tree}_\alpha))$ this is well-typed. Considering $\mu \mathrm{X}.\mathtt{in}_2(\langle a, \langle \mathrm{X},\mathrm{X} \rangle \rangle)$ one sees that trees can be infinite. Many tree algorithms are generic and the depth-first tree visitor is a popular example of this. It receives a type α, a function f of type $\alpha \stackrel{\varphi}{\Rightarrow} \mathsf{Unit}$ and a tree of type α. Then f is applied to each node in t in depth-first order. The algorithm df_φ, defined next, is generic in the type of tree-nodes but not in the effect of the function applied to each node. This is because our language does not offer effect polymorphism.

$$\mathsf{df}_{\alpha,\varphi} = \mu g^{\mathsf{tree}_\alpha \stackrel{\varphi}{\Rightarrow} \mathsf{Unit}}.\lambda f^{\alpha \stackrel{\varphi}{\Rightarrow} \mathsf{Unit}}.\lambda t^{\mathsf{tree}_\alpha}.M$$
$$M = \mathtt{case}\ t\ \mathtt{of}\ (\mathsf{leaf} \triangleright \mathtt{skip} \mid \langle l,a,r \rangle \triangleright g(l); f(a); g(r))$$
$$\mathsf{df}_\varphi = \Lambda \mathrm{X}.\mathsf{df}_{\mathrm{X},\varphi}$$

Fig. 7 Proof rules for trees.

$$[Leaf]\ \frac{-}{A[\mathsf{leaf}/u]\ \vdash \mathsf{leaf} :_u A}$$

$$[Node]\ \frac{A\ \vdash V_i :_{u_i} B_i\ (i=1,2,3) \quad (B_1 \wedge B_2 \wedge B_3) \supset B[\langle u_1,u_2,u_3\rangle/u]}{A\ \vdash \langle V_1,V_2,V_3\rangle :_u B}$$

$$[Case]\ \frac{\{C\}\ V :_v \{A\} \quad \begin{array}{c}\{C \wedge A[\mathsf{leaf}/v]\}\ M_1 :_u \{C'\}\\ \{C \wedge A[\langle l,a,r\rangle/v]\}\ M_2 :_u \{C'\}\end{array}}{\{C\}\ \mathtt{case}\ V\ \mathtt{of}\ (\mathsf{leaf} \triangleright M_1 \mid \langle l,a,r\rangle \triangleright M_2) :_u \{C'\}}$$

The case statement is syntactic sugar for

$$\mathtt{case}\ t\ \mathtt{of}\ \{\mathtt{in}_i(y_i).M_i\}_{i\in\{1,2\}}$$

where y_1,y_2 are fresh, $M_1 = \mathsf{leaf}$ and

$$M_2 = (\lambda l.\lambda a.\lambda r.N)(\pi_1(\pi_2(y_2)))(\pi_1(y_2))(\pi_2(\pi_2(y_2))).$$

Figure 7 proposes three derived rules for reasoning about trees. Next we give a specification for df_φ. We begin by defining a predicate for the number of nodes in the tree.

$$\begin{aligned}
\mathsf{size}(t,0) &\equiv t = \mathsf{leaf}\\
\mathsf{size}(t,n+1) &\equiv \exists l,a,r,m_l,m_r.t = \langle l,a,r\rangle\\
&\quad \wedge\ \mathsf{size}(l,m_l)\\
&\quad \wedge \mathsf{size}(r,m_r) \wedge m_l + m_r = n\\
\mathsf{finite}(t) &\equiv \exists n.\mathsf{size}(t,n)
\end{aligned}$$

Now we can specify the algorithm. Essentially all we guarantee is that f is applied to exactly all nodes in depth-first order and the effects of these applications are properly chained.

$$\begin{aligned}
\mathsf{DF}_{\varphi u} &\equiv \forall \mathrm{X}.\forall f^{\mathrm{X}\overset{\varphi}{\Rightarrow}\mathsf{Unit}}.\forall t^{\mathrm{X}}.A \text{ with } A \overset{\text{def}}{\equiv} \mathsf{finite}(t) \supset (\mathsf{DF}^{\mathsf{l}}{}_{\varphi \mathrm{X} uft} \wedge \mathsf{DF}^{\mathsf{n}}{}_{\varphi \mathrm{X} uft})\\
\mathsf{DF}^{\mathsf{l}}{}_{\varphi \mathrm{X} uft} &\equiv t = \mathsf{leaf} \supset \{C\}\ u \bullet \mathrm{X} \bullet f \bullet t \{C\}\\
\mathsf{DF}^{\mathsf{n}}{}_{\varphi \mathrm{X} uft} &\equiv \exists l,a,r.(t = \langle l,a,r\rangle \supset B_\varphi(\mathrm{X},u,f,t,l,a,r)\\
B_{\varphi \mathrm{X} uftlar} &\equiv \left(\begin{array}{c}\{C\}\ u \bullet \mathrm{X} \bullet f \bullet l \{C_1\}\\ \wedge\\ \{C_1\}\ f \bullet a\ \{C_2\}\\ \wedge\\ \{C_2\}\ u \bullet \mathrm{X} \bullet f \bullet r \{C'\}\end{array}\right) \supset \{C\}\ u \bullet \mathrm{X} \bullet f \bullet t \{C'\}
\end{aligned}$$

Now we can assert

$$\mathsf{T}\ \vdash \mathsf{df}_\varphi :_u \mathsf{DF}_{\varphi u}.$$

Counting Nodes by Depth-First Searching In itself df_φ does not cause effects, apart from what the function applied to each node does. Now we implement a node-counting algorithm that increments a local variable every time a tree-node is being accessed.

$$\begin{aligned}
\mathsf{df}' &\stackrel{\text{def}}{=} \texttt{new var}\ n := 0\ \texttt{in}\ (M; !n)\\
M &\stackrel{\text{def}}{=} \texttt{let}\ counter = \Lambda \mathrm{X}.\lambda x^{\mathrm{X}}.n := 1+!n\ \texttt{in}\ N\\
N &\stackrel{\text{def}}{=} \mathsf{df}_{\{n\}}\ [\mathrm{X}]\ counter
\end{aligned}$$

Then df' we can derive the following judgement.

$$\mathsf{T} \vdash \mathsf{df}' :_u \forall \mathrm{X}.\forall t^{\mathsf{treex}}.\forall n.(\mathsf{size}(t,n) \supset u \bullet [\mathrm{X}] \bullet t = n)$$

Once more we have a purely functional specification for an imperative algorithm, because the non-functional behaviour is not externally visible.

As such, our logic can offer a compact specification and a simple proof to infer the complex polymorphic shared data structures with local and global destructive updates; [23, 25] list further examples.

4.8 Soundness with Local State and Further Results

In this subsection we establish Theorem 34. Since the proofs of the soundness of the rules inherited from the logic with global state closely follow those given in §3.6, we leave their proofs to Appendix A, focussing on the newly added rules in Figure 6. We use the following lemmas, which are the counterpart of Lemmas in §3.6.

Lemma 35. *Let $\Gamma,\Theta \vdash C$ and assume given $\Gamma \vdash (\nu\vec{y})(\xi,\sigma)$, $\Theta \vdash I$.*

1. (thinning) *Assume $x \notin \mathsf{fv}(\Gamma) \cup \mathsf{fv}(\Theta)$. Then*

$$(\nu\vec{y})(\xi, \sigma\cdot x \mapsto V) \models^I C \quad \textit{implies} \quad (\nu\vec{y}\setminus x)(\xi,\sigma) \models^I C$$

2. (weakening) *For all $\Gamma' \vdash (\nu\vec{y})(\xi',\sigma')$ and $\Theta' \vdash I$ such that $\mathsf{dom}(\Gamma)\cap\mathsf{dom}(\Gamma') = \mathsf{dom}(\Theta)\cap\mathsf{dom}(\Theta') = \{\vec{x}\}\cap(\mathsf{dom}(\Gamma')\cup\mathsf{dom}(\Theta')\cup\{\vec{y}\}) = \{\vec{y}\}\cap(\mathsf{dom}(\Gamma)\cup\mathsf{dom}(\Theta)\cup\{\vec{x}\}) = \emptyset$, we have:*

$$(\nu\vec{x})(\xi,\sigma) \models^I A \textit{ implies } (\nu\vec{x}\vec{y})(\xi\cdot\xi',\sigma\cdot\sigma') \models^{I\cdot I'} C$$

PROOF: Direct from the definition of the satisfaction.□

Lemma 36. (substitution) *Assume $x \notin \{\vec{y}\}\cup\mathsf{fv}(e)\cup\mathsf{dom}(\mathcal{M})$ and $\{\vec{y}\}\cap(\mathsf{fv}(e)\cup\mathsf{dom}(\mathcal{M})) = \emptyset$. If $\mathcal{M} = (\nu\vec{y})(\xi,\sigma)$ then*

$$\mathcal{M} \models^I C[e/x] \quad \textit{iff} \quad (\nu\vec{y})(\xi\cdot x : [\![e]\!]\mathcal{M}\cdot I, \sigma) \models^I C.$$

PROOF: Mechanical from the clauses for defining the satisfaction.□

The following property of hiding in models and in formulae is important for reasoning about rules for local state.

Lemma 37. *Suppose $C \setminus \nu x$ is defined. Then*

1. $\mathcal{M} \models^I C \setminus \nu x$ *implies* $(\nu x)\mathcal{M} \models^I C$.
2. $\mathcal{M} \models^I C$ *implies there exists* $\mathcal{M}'$ *such that* $\mathcal{M}' \models C \setminus \nu x$ *and* $(\nu x)\mathcal{M}' \cong \mathcal{M}$.

PROOF: (1) Let $C \stackrel{\text{def}}{=} (\nu x\vec{y})\vec{z}$ in C'. Then by assumption, we have $\mathcal{M} \models^I (\nu\vec{y})\vec{z}$ in C', which implies $\mathcal{M}' \models^I C'$ with $\mathcal{M} \cong (\nu\vec{y})\mathcal{M}'$ and $\mathcal{M}' \stackrel{\text{def}}{=} \xi \cdot \vec{z} : \vec{W}\vec{y} \mapsto \vec{V} \cdot x \mapsto V' \cdot \sigma$ for some $\xi, \vec{W}, \vec{V}, V'$ and σ. This implies $(\nu\vec{y}x)\mathcal{M}' \models^T C$. Hence $(\nu x)\mathcal{M} \models^T C$, as required.

(2) If $\mathcal{M} \models^I (\nu x\vec{y})\vec{z}$ in C', then $\mathcal{M} \cong (\nu x\vec{y})\mathcal{M}'$ and $\mathcal{M}' \models^I C'$ such that $\mathcal{M}' \stackrel{\text{def}}{=} \xi \cdot \vec{z} : \vec{W}\vec{y} \mapsto \vec{V} \cdot x \mapsto V' \cdot \sigma$ for some $\xi, \vec{W}, \vec{V}, V'$ and σ. This implies $(\nu\vec{y})\mathcal{M}' \models^I (\nu\vec{y})\vec{z}$ in $C' \stackrel{\text{def}}{=} C \setminus \nu x$ where $(\nu x\vec{y})\mathcal{M}_1 \cong \mathcal{M}$, as required. □

The following result is worth stating because of the closure of the satisfaction under $\cong$.

Proposition 38. $\mathcal{M}^\Gamma \models^I A \wedge B$ *iff* $\mathcal{M}^\Gamma \models^I A$ *and* $\mathcal{M}^\Gamma \models^I B$. $\mathcal{M}^\Gamma \models^I \neg A$ *iff* $\mathcal{M}^\Gamma \models^I A$ *does not hold. Similarly for other connectives/quantifiers.*

PROOF: For the conjunction, the "if" direction is by definition. For the "only if" direction, if $\mathcal{M}^\Gamma$ satisfies $A \wedge B$ then some $\mathcal{M}_0^\Gamma$ which is congruent to $\mathcal{M}$ satisfies both A and B, hence $\mathcal{M}^\Gamma$ satisfies both A and B. For the negation, again only the "only if" direction is non-trivial. If both $\mathcal{M}^\Gamma \models^I \neg A$ and $\mathcal{M}^\Gamma \models^I A$ hold, then from the former there is some $\mathcal{M}_0^\Gamma$ such that (1) it is congruent to $\mathcal{M}$ and (2) it is not the case that $\mathcal{M}_0^\Gamma \models^I A$. Since $\mathcal{M}^\Gamma \models^I A$ by assumption, (1) means $\mathcal{M}_0^\Gamma \models^I A$ by assumption, contradicting (2). Other cases are essentially the same.□

We can now move to the proof of Theorem. For [*New*], let $\mathcal{M} = (\nu\vec{y})(\xi, \sigma^{-x})$. Clearly

$$(\nu\vec{y})(\texttt{new var}\ x := V\xi\ \texttt{in}\ N\xi) \to (\nu\vec{y}x)(N\xi, \sigma \cdot x \mapsto V\xi) \Downarrow (\nu\vec{z})(W, \sigma')$$

is the only way to converge to a value. Here we prove:

$$(\nu\vec{y})(u : W, \sigma') \models^I C'$$

By the first premise $\{C\}V :_v \{C_1\}$, we have:

$$(\nu\vec{y})(\xi \cdot v : V\xi, \sigma) \models^I C_1$$

By the (IH) on the second premise and Lemma 36, we have:

$$(\nu\vec{y})(\xi \cdot v : V\xi, \sigma) \models^I C_1 \Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi) \models^I C_1[!x/v] \tag{4.18}$$

Then (4.18) gives

$$\begin{aligned} (\nu\vec{y})(N\xi, \sigma \cdot x \mapsto V\xi) \Downarrow (\nu\vec{z} \setminus x)(W, \sigma') &\Rightarrow (\nu\vec{y} \setminus x)(\xi \cdot u : W, \sigma') \models^I C' \setminus \nu x \quad (4.19)\\ &\Rightarrow (\nu\vec{y})(\xi \cdot u : W, \sigma') \models^I C' \quad (4.20) \end{aligned}$$

(4.19) is by the second premise and the (IH), while (4.20) is from Lemma 37.

The next rule is [*CoNew*]. With $\mathcal{M} = (\nu\vec{y})(\xi, \sigma^{\cdot x})$, we derive

$$
\begin{aligned}
(\nu\vec{y})(M\xi,\sigma) \Downarrow (\nu\vec{z})(V,\sigma') &\Rightarrow (\nu\vec{y}\setminus x)(M\xi,\sigma) \Downarrow (\nu\vec{z}\setminus x)(V,\sigma') && (4.21)\\
&\Rightarrow (\nu\vec{z}\setminus x)(\xi\cdot u : V,\sigma') \models^{I} C'\setminus \nu x && (4.22)\\
&\Rightarrow (\nu\vec{z})(\xi\cdot u : V,\sigma') \models^{I} C' && (4.23)\\
&\Rightarrow (\nu\vec{z})(u : V,\sigma') \models^{I} C' && (4.24)
\end{aligned}
$$

(4.21) is by Lemma 36, (4.22) follows by (IH), and (4.23) is a consequence of Lemma 37. (4.24) is by thinning.

We now prove [*LetN*]. As usual we assume with $\mathcal{M} = (\nu\vec{y})(\xi,\sigma)$. By determinacy or reductions here is only the following potential reduction sequence towards a value.

$$
\begin{aligned}
(\nu\vec{y})(M\xi,\sigma) \rightarrow^{*} (\nu\vec{y}\vec{a})(W\xi,\sigma'). \;\Rightarrow\;& (\nu\vec{y})((\mathtt{let}\; x = M \;\mathtt{in}\; N)\xi,\sigma)\\
\rightarrow^{*}\;& (\nu\vec{y}\vec{a})((\mathtt{let}\; x = W \;\mathtt{in}\; N)\xi,\sigma')\\
\rightarrow\;& (\nu\vec{y}\vec{a})((N[W/x])\xi,\sigma') \Downarrow (\nu\vec{b})(V,\sigma'')
\end{aligned}
$$

With this assumption, we prove:

$$(\nu\vec{b})(x : V,\sigma'') \models^{I} C'$$

By (IH), we have:

$$
\begin{aligned}
(\nu\vec{y}\vec{a})(\xi\cdot x : W\xi,\sigma') \models^{I} (\nu\vec{a})x\vec{b} \;\mathsf{in}\; C_1 &\Rightarrow (\nu\vec{y})(\xi\cdot x : W\xi,\sigma') \models^{I} C_1 && (4.25)\\
&\Rightarrow (\nu\vec{b}\setminus\vec{a})(\xi\cdot x : V,\sigma') \models^{I} C'\setminus\nu\vec{a} && (4.26)\\
&\Rightarrow (\nu\vec{b})(\xi\cdot x : V,\sigma'') \models^{I} C' && (4.27)
\end{aligned}
$$

Here (4.25) and (4.27) are a consequence of Lemma 37 while (4.26) comes from the second premise and the (IH). Finally thinning gives us the required result.

Further Results. We conclude this section with basic properties of validity in the logic. We leave their detailed proofs to elsewhere.

First, the following holds since, essentially, the satisfaction is closed under $\cong$.

Proposition 39. *$\{C\}\, M_1^{\Gamma;\rho;\varphi} :_u \{C'\}$ and $\Gamma \vdash M_1 \cong M_2 : \alpha,\varphi$ imply $\{C\}\, M_2^{\Gamma;\rho;\varphi} :_u \{C'\}$.*

We also have the converse of Proposition 39, which holds because all finite sub-behaviours of a typed program can be captured by assertions, cf. [23, §3].

Proposition 40. *If $\Gamma \vdash M_1 \not\cong M_2 : \alpha,\varphi$, then $\{C\}\, M_i :_u \{C'\}$ but not $\{C\}\, M_j :_u \{C'\}$ with $i \neq j$ for some C and C'.*

Define $\Gamma \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha,\varphi$ to mean $\{\langle C,C'\rangle \mid \{C\}\, M_1 :_u \{C'\}\} = \{\langle C,C'\rangle \mid \{C\}\, M_2 :_u \{C'\}\}$. From Propositions 39 and 40 we conclude:

Corollary 41. (equational characterisation) $\Gamma \vdash M_1 \cong M_2 : \alpha,\varphi$ *iff* $\Gamma \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha,\varphi$.

Corollary 41 does not necessarily say that assertions in the present logic are expressive. But it does say they collectively capture all and only observable properties of programs.

5 Information Flow Analysis in Program Logic

This section offers a brief account of an experiment which uses the compositional logic discussed in the previous sections as a descriptive basis for program analysis, taking secrecy analysis [24] as an example. Our investigation centres on *noninterference*, the very property secrecy analysis aims to guarantee. Noninterference, or secrecy, means that a difference in a high-level (private) behaviour is never transmitted to a difference in a low-level (public) behaviour. We take the following three steps to achieve the main non-interference theorem.

Step 1: Introduction of a refinement of total logic called *generalised total logic*, which can make assertions about both divergence and termination.
Step 2: Formulation of the *NI-predicate* which characterises the noninterference property as assertions.
Step 3: Demonstration of the *soundness* of securely typed programs, by proving securely typed programs satisfy the predicate defined in Step 2.

The key point of Step 1 is to explicitly discuss termination and divergence, which is crucial for secrecy and dependency analysis [3, 57]. Combining Step 2 and 3, we can prove noninterference of securely typed programs. The main appeal of this three-step methods is that, thanks to the clean stratification of our logic, only very small, incremental changes of NI-predicates are necessary when we add or refine program constructs. Step 2 is independent from individual secrecy typings, while Step 3 is verified by simple induction. § 5.1, § 5.2 and §5.3 develop Step 1, 2 and 3, respectively, and § 5.4 discusses extensions.

We work under the following convention in the present section.

Convention 42. Throughout the present section, we stipulate: (1) We use the language without recursive types or polymorphism as our base language; and (2) We assume that effects in an arrow type are annotated with their types which conform to the given environment. Informally we may often still write effects without type annotations.

Regarding (1), the restricted syntax is all we need in the applications treated in this section. We shall discuss how the recursive types and polymorphisms may be incorporated at the end of the section. Regarding (2), explicitly typed effects are to make circularity in types explicit so that an implicit recursion induces a partial behaviour (such as $x := \lambda().\ (!x)()\ ;\ (!x)()$). We give one example.

Example 43. As an example of explicitly typed effects, we have the following typing.

$$x : \mathsf{Ref}(\mathsf{Nat}), u : \mathsf{Unit} \overset{x:\mathsf{Ref}(\mathsf{Nat})}{\Rightarrow} \mathsf{Unit} \vdash \lambda().x := 0$$

Note the type of an effect x conforms to the type which x is given in the environment. We can then check the assignment $x := \lambda().\ (!x)()$ is untypable without recursive types: if x is to be given a type $\mathsf{Ref}(\alpha)$, then α should be equal to $\mathsf{Unit} \overset{x:\mathsf{Ref}(\alpha)}{\Rightarrow} \mathsf{Unit}$.

Fig. 8 Typing with Pointed Types: Added Rules for Global State

$$[Rec]\ \frac{\Gamma, x : \alpha \overset{\varphi}{\Rightarrow} \delta \vdash \lambda y^{\alpha}.M : \alpha \overset{\varphi}{\Rightarrow} \delta \quad \alpha \overset{\varphi}{\Rightarrow} \delta \text{ pointed}}{\Gamma \vdash \mu x^{\alpha \overset{\varphi}{\Rightarrow} \delta}.\lambda y^{\alpha}.M : \alpha \overset{\varphi}{\Rightarrow} \delta}$$

$$[Lift]\ \frac{\Gamma \vdash M : \alpha}{\Gamma \vdash M : \llcorner\alpha\lrcorner} \qquad [Seq]\ \frac{\Gamma \vdash N : \llcorner\alpha\lrcorner \ \ \Gamma, x : \alpha \vdash M : \beta \ \ \beta \text{ partial}}{\Gamma \vdash \mathtt{seq}\, x^{\alpha} = N \,\mathtt{in}\, M : \beta}$$

5.1 Total Correctness with Partiality

Pointed Types We first refine the type structure by introducing pointed types [26] which distinguish totality and partiality at the level of types, following [3, 24]. This refinement is not necessary for the purpose of having logical description of noninterference properties: however we use it in the secrecy typing. *From this subsection to §5.4, we exclusively work with the language without local state.*

The only syntactic change is that `let` is split into a total version (still called `let`) and its potentially partial counterpart, written $\mathtt{seq}\, x^{\alpha} = M \,\mathtt{in}\, N$. The latter's reduction rule is identical with the old `let`. To the syntax of types we add:

$$\alpha, \beta ::= \ldots \mid \llcorner\alpha\lrcorner$$

We only work with the set of well-formed types, defined below.

Definition 44.

1. (partial, total and pointed types) A type of the form $\llcorner\alpha\lrcorner$ is *partial*. A type which is not partial is *total*. A type is *pointed* [3, 26] if it has shape $\alpha \overset{\varphi}{\Rightarrow} \beta$ with β partial.
2. (well-formed types) The set of well-formed types are inductively generated by the following clauses.
   - Atomic types are well-formed.
   - $\alpha \overset{\varphi}{\Rightarrow} \beta$ is well-formed if α and β are well-formed and, moreover, α is total.
   - $\llcorner\alpha\lrcorner$ is well-formed if α is total and well-formed.
   - $\alpha \times \beta$ is well-formed if α and β are well-formed and total.
   - $\alpha + \beta$ is well-formed if α and β are well-formed and total.

Convention 45. From now on we only treat well-formed types.

Remark 46. (pointed types) In pure functional types, any type of the form $\alpha \Rightarrow \beta$ such that β itself is (inductively) pointed can also be called pointed. We can also add other forms of types. such as $\alpha \times \beta$ with α and β pointed, among pointed types. In essence, a pointed type is a total type which has the least element in it.

Typing with pointed types use the same judgements, stipulating that, in $\Gamma \vdash M : \alpha$ and $\Gamma \vdash M : \alpha, \varphi$, we restrict the co-domain of Γ to be total types and reference types. The typing rules are Figure 1 without [Rec], augmented with the rules in Figure 8 (which refine [*Rec*] and add rules for partial types). We observe:

Proposition 47.

1. (subject reduction) *If* $\Gamma \vdash M : \alpha, \varphi$ *with pointed types and* $M \longrightarrow M'$*, then* $\Gamma \vdash M' : \alpha, \varphi$*. Similarly if* $\Gamma \vdash M : \alpha$ *with pointed types and* $M \longrightarrow M'$*, then* $\Gamma \vdash M' : \alpha$*.*
2. (totality) *If* $\Gamma \vdash M : \alpha, \varphi$ *is semi-closed (i.e.* $\mathsf{cod}(\Gamma)$ *are reference types) and* α *is total, then for each* $\Gamma \vdash \sigma$ *such that* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma)$*, we have* $(M, \sigma) \Downarrow$*.*

The proof of (1) is standard. For (2) we observe invocation of recursion is not possible by syntax.

Generalised Total Correctness We refine the logic so that we can assert on both total and partial computation. First we extend the grammar of terms.

$$e ::= \ldots \mid \llcorner e \lrcorner \mid \bot^{\llcorner \alpha \lrcorner}$$

For typing of terms, $\llcorner e \lrcorner$ (which denotes a convergent program of partial type) is typed with $\llcorner \alpha \lrcorner$ if e is typed with α. $\bot^{\llcorner \alpha \lrcorner}$, which denotes a divergent program, has always a partial type. The following notations are often useful.

Notation 48. $e \Downarrow$ denotes $e \neq \bot$. $C \Downarrow_x$ denotes $C \supset x \Downarrow$.

For formulae, we add $\{C\} e \bullet e' \Uparrow$ which stands for divergent procedure invocation.[4] The semantics of judgements changes when programs have partial type. Assuming M has a partial type, $\{C\} M :_u \{C'\}$ now says,

> *if M starts execution under an environment and a state satisfying C, then the result of evaluation satisfies C′ – which may indicate M diverges.*

Similarly for $A \models M :_u B$. Semantically, terms of total types are interpreted as values, while the anchor of a partial type is either a value or a diverging program. The rest is unchanged, except that we don't demand convergence at partial types in sequents or in the evaluation formulae.

This logic subsumes the original logic since an assertion $\{C\} M :_u \{C'\}$ in the original system (which, as far as C is satisfiable, guarantees a termination of M) can be translated into $\{C\} M :_u \{C' \wedge u \Downarrow\}$ in the generalised logic. It also encompasses partial correctness in the standard sense of Hoare Logic since we can now assert, for example, "this program either calculates the doubling function or diverges," For details see [23, §4–§7]. In the present study, this allows us to describe how different effects may lead to difference in termination behaviour.

The proof rules ensuring generalised total correctness for programs with global state follow Figure 3 and Figure 2 except we add the rule for [*Seq*] and add an additional rule for recursion. The new rule for recursion makes it possible to infer a safety property for recursive behaviours. The rule uses the notion of *admissibility* [14] with respect to the termination ordering [55]. Essentially, admissibility means a property which is downward closed — if a program satisfies it, then a less defined program satisfies it. Further we demand if a chain of programs satisfy the property, then its lub (if it exists as a program) also satisfies the property. Below Ω^{α} denotes the least terminating closed program in α (if any).

[4] This predicate is implicitly present in the original logic as the negation of an evaluation formula.

Fig. 9 Refined Proof Rules for Generalised Total Correctness

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

$$[Seq]\ \frac{\begin{array}{c}\{C\}\ M :_m \{\ (\nu\vec{y})m\vec{z} \ \mathsf{in}\ C_1[\llcorner m \lrcorner/x]\ \}\\ \{C_1\}\ N :_u \{C' \backslash \nu\vec{y}\} \quad C' \Downarrow_u \supset C_1 \Downarrow_m\end{array}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C'\}}$$

Definition 49. (admissibility)

1. (termination ordering) $M_1 \sqsubseteq M_2$ with $\vdash M_{1,2} \triangleright \alpha$ when $C[M_1] \Downarrow$ implies $C[M_2] \Downarrow$ for each $C[\cdot]^{\llcorner \mathsf{Unit} \lrcorner}_{\alpha}$.
2. (approximation) A sequence $\{V_i\}_{i \in \mathsf{Nat}}$ *approximate* V_ω if: (2-a) $V_i \sqsubseteq V_{i+1} \sqsubseteq V_\omega$ for each i, and (2-b) $C[V_\omega]$ is a lub of $\{C[V_i]\}_{i \in \mathbb{N}}$ for each well-typed $C[\ \cdot\ ]$.
3. (admissibility) A formula A is *admissible at* u^α when (i) $\models \Omega :_u A$ and (ii) for a sequence $\{V_i\}_{i \in \mathsf{Nat}}$ approximating V_ω, if $\models V_i :_u A$ for each i then $\models V_\omega :_u A$ where $\sqsubseteq$ and approximation are defined as follows.

Remark 50. (admissibility) In Definition 49 (2-b), we close the ordering by each context so that a pathological lub (cf.[55, §4]) is excluded (actually the only approximating chains we use are those of the shape $\{V_i\}_{i \in \mathbb{N}}$ approximating $\mu x.V$ for some $x : \alpha \vdash V : \alpha$, given inductively by $V_0 = \Omega$ and $V_{i+1} = V[V_i/x]$).

The proof rules are given in Figure 9 (which are combined with the rules in Figure 3 and Figure 2). [*Seq*] ensures convergence of the argument N when we wish to guarantee convergence of the whole term. [*Rec*-$\Uparrow$] uses admissibility. The rule may not be sound if B is not admissible as an example below demonstrates. The condition $A \supset \exists x.B$ ensures A is stronger than B in its constraint on auxiliary names (among others we can avoid the case when, for example, $B = \mathsf{F}$, by this condition). We observe:

Proposition 51. *Let* $\Gamma \vdash M : \llcorner \alpha \lrcorner, \varphi$ *be semi-closed and* $\Gamma \vdash \sigma$ *such that* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma)$. *Further assume C is satisfiable. Then:* (1) $\{C\}M :_u \{u \Downarrow\}$ *implies* $M \Downarrow$. (2) $\{C\}M :_u \{u = \bot\}$ *implies* $M \Uparrow$.

Remark 52. (the need for admissibility) We show an example which shows the need for admissibility. Consider a program:

$$M \stackrel{\text{def}}{=} \mu x^{\mathsf{Unit} \Rightarrow \llcorner \mathsf{Unit} \lrcorner}.\lambda().x()$$

where $x()$ is a shorthand for $\mathtt{seq}\ y = x()\ \mathtt{in}\ y$ and an assertion

$$B(u) \stackrel{\text{def}}{=} u \bullet () \Downarrow.$$

Then it is surely true that $B(x) \models \lambda().x() :_u B(u)$, but $\mathsf{T} \models M :_u B(u)$ does not hold.

5.2 Predicates for Noninterference

We demonstrate how our logic can clarify noninterference properties incrementally for different language constructs starting from pure monomorphic functions (i.e. those without reference types, polymorphism or recursive types), later refining the framework with treatment of global and local state. In both secrecy typing and in formulation of a predicate for noninterference, this class of programs is the simplest to treat, while offering a basis on which treatment of more complex cases can be given. In particular, it is worth noting that the clauses for defining predicates for noninterference do *not* differ among different classes except for the case of arrow types.

Convention 53. In this subsection, we exclusively work with programs with pointed types but without references, both in syntax and type structures.

Let $\mathcal{L}$ be a non-trivial complete lattice of secrecy levels (higher means more secret/private), ranged over by $s, s', \ldots$. We write H and L for the top and bottom elements in $\mathcal{L}$. First, secrecy-annotated types are generated by the following grammar.

$$\alpha \quad ::= \quad \mathsf{Bool}_s \quad | \quad \mathsf{Nat}_s \quad | \quad \alpha \Rightarrow \beta \quad | \quad \alpha \times \beta \quad | \quad \alpha +_s \beta \quad | \quad \llcorner \alpha \lrcorner_s$$

Among them, we only use well-formed types in the sense of of Definition 44. Secrecy annotations indicate the level at which information is communicated. For example, Nat_H is a natural number which has a high secrecy level (i.e. which is the most secret). An intuition is that a datum of type Nat_s can only be observed by a principal which acts at the level s or higher. Secrecy intuitively means that differences at high-level data, or actions based on high-level data, are never transmitted (become observable) as differences at low-level data.

Some programs with secrecy-annotated types are safe in this sense, while others are not, even if they are typable in the non-secrecy typing. For example

$$\vdash \lambda x^{\mathsf{Bool}_\mathrm{H}}.\mathtt{if}\ x\ \mathtt{then}\ 0\ \mathtt{else}\ 1 : \mathsf{Bool}_\mathrm{H} \Rightarrow \mathsf{Nat}_\mathrm{L} \qquad (5.1)$$

is not secure: it receives high-level information and emits its content as a low-level value via the conditional judgement.

We start by asking why (5.1) is not secure. One possible answer is that it transforms what cannot be distinguished at low-level (two high-level booleans) into what can be (low-level numerals). In other words, *it transforms indistinguishability to distinguishability*. Note these two notions, “distinguishability” and “indistinguishability”, are seen from a specific secrecy level, in this case the low-level. If, on the other hand, such undue transformations never take place in a program, then that program must surely be secure. This understanding leads to the following *“make-no-difference”* predicate, $\mathtt{mnd}(e, e' : \beta, s)$, which intuitively asserts:

Seen from level s, e and e′, both being of type β*, are indistinguishable.*

A central point of this predicate is that the assertion makes sense for complex data types (and later even with side effects). For example, two data of an arrow type $\alpha \Rightarrow \beta$ make no difference from level s if, whenever two data of type α make no difference from s, then the result of applying these data again do not make difference at level s.

Fig. 10 "make-no-difference"-predicate and NI-predicate

$$\mathtt{mnd}(e, e' : \mathsf{Unit}, s) \stackrel{\text{def}}{=} \mathsf{T}$$

$$\mathtt{mnd}(e, e' : \mathsf{Nat}_{s'}, s) \stackrel{\text{def}}{=} s' \sqsubseteq s \supset e = e'$$

$$\mathtt{mnd}(e, e' : \alpha \Rightarrow \beta, s)$$
$$\stackrel{\text{def}}{=} \forall x^{\alpha}, y^{\alpha}. ((\mathtt{NI}(x : \alpha) \wedge \mathtt{NI}(y : \alpha) \wedge \mathtt{mnd}(x, y : \alpha, s)) \supset \mathtt{mnd}(e \bullet x, e' \bullet y : \beta, s)))$$

$$\mathtt{mnd}(e, e' : \alpha_1 +_{s'} \alpha_2, s)$$
$$\stackrel{\text{def}}{=} s' \sqsubseteq s \supset \bigwedge_{i=1,2} \forall x^{\alpha_i}. \exists y^{\alpha_i}. (\ e = \mathtt{in}_i(x) \ \supset \ e' = \mathtt{in}_i(y) \wedge \mathtt{mnd}(x, y : \alpha_i, s)\ )$$

$$\mathtt{mnd}(e, e' : \alpha_1 \times \alpha_2, s) \stackrel{\text{def}}{=} \bigwedge_{i=1,2} \mathtt{mnd}(\pi_i(e), \pi_i(e') : \alpha_i, s)$$

$$\mathtt{mnd}(e, e' : \llcorner \alpha \lrcorner_{s'}, s)$$
$$\stackrel{\text{def}}{=} (s' \sqsubseteq s \supset e \Downarrow \equiv e' \Downarrow) \wedge \forall x^{\alpha}, y^{\alpha}. ((e = \llcorner x \lrcorner \wedge e' = \llcorner y \lrcorner) \supset \mathtt{mnd}(x, y : \alpha, s))$$

$$\mathtt{NI}(e : \beta) \stackrel{\text{def}}{=} \forall s. \mathtt{mnd}(e, e : \beta, s)$$

The predicate $\mathtt{mnd}(e, e' : \beta, s)$ is defined by induction on types, and is given in Figure 10. In Figure 10, we write e.g. $s \sqsubseteq s'$ in an assertion, internalising $\mathcal{L}$ in the logic. We can read these clauses thus:

- Two closed values of unit types are always the same, so whichever level you are at, they make no difference.
- Two numerals of level s' make no difference from level s iff either $s' \not\sqsubseteq s$ (s' is not viewable from s) or if they are identical. Similarly for $\mathsf{Bool}_{s'}$ (which is omitted), $\alpha +_{s'} \beta$ and $\llcorner \alpha \lrcorner_{s'}$.
- The most complex case is the arrow type. It reads: for e and e' of type $\alpha \Rightarrow \beta$ to be s-indistinguishable, we demand that, for any two s-indistinguishable x and y of type α which moreover satisfy the NI-predicate (given below), the result of feeding them to e and e' is again s-indistinguishable. The clause for products is naturally defined by the conjunction.

As a further remark on arrow types, observe that the clause demands the arguments to be "non-interfering" (using the noninterference predicates given in the last line). This is not to make the predicate too strict. Without this condition, an innocuous program such as $M \stackrel{\text{def}}{=} \lambda x.x : (\mathsf{Bool}_{\mathrm{H}} \Rightarrow \mathsf{Bool}_{\mathrm{L}}) \Rightarrow (\mathsf{Bool}_{\mathrm{H}} \Rightarrow \mathsf{Bool}_{\mathrm{L}})$ would always make a difference (consider feeding M with an argument $\lambda y.y : \mathsf{Bool}_{\mathrm{H}} \Rightarrow \mathsf{Bool}_{\mathrm{L}}$). Practically this condition makes sense since we may as well assume all concerned programs behave securely — if a procedure is not secure, a program which calls it cannot stay secure.

Finally, the noninterference predicate, written $\mathtt{NI}(e : \alpha)$, is defined as in the last clause of Figure 10. The predicate $\mathtt{NI}(e : \alpha)$ literally says that e does not make any difference from itself at any level. More intuitively, it says that, at any secrecy level, e does not transform what are indistinguishable to what are distinguishable (since if it does at level s, two copies of e would make a difference at s).

A formalisation of noninterference for open programs follow.

Definition 54. (noninterference predicate for open programs) Write $\mathtt{mnd}'(x, y : \alpha, s)$ for the conjunction of $\mathtt{NI}(x : \alpha)$, $\mathtt{NI}(y : \alpha)$ and $\mathtt{mnd}(x, y : \alpha, s)$. Then $\Gamma \models_{\mathsf{sec}} M : \beta$ is the following property (for simplicity we present the case $\Gamma = x : \alpha$):

For each $s \in \mathcal{L}$, *if* $\mathsf{T} \models V, W :_{v,w} \mathtt{mnd}'(v, w : \alpha, s)$, *then* $\mathsf{T} \models M[V/x], M[W/x] :_{v,w} \mathtt{mnd}(v, w : \beta, s)$.

where V and W are closed. $A \models V, W :_{v,w} B$ stands for $A \models \langle V, W \rangle :_u B[\pi_1(u)\pi_2(u)/vw]$.

Remark 55. (noninterference predicate for open programs)

1. In Definition 54, $\Gamma \models_{\mathsf{sec}} M : \beta$ is defined over arbitrary well-typed programs with security annotation, *not* over programs typed by the secrecy typing which is introduced later. Thus the definition delineates the noninterference property among the whole properties general programs may satisfy, not only as the property of securely typed ones.
2. In the context of (1), the above formalisation of noninterference is closer to the denotational/relational approach as given in [3, 6, 53] and the approach based on secrecy bisimulations [61] than those which are based on securely typed programs and their equational/operational properties.
3. At the same time, the formulation directly mentions behaviours of programs and environments as logical predicates. In particular, the requirement for non-interfering values in the environment (which is essentially about compositionality) gives a direct link to those approaches based on secrecy typing.
4. The property formalised above cannot be written down as a judgement in the logic as far as an open program is the subject of a sequent. This is because, for formalising the notion of make-no-difference for open programs, we need to vary behaviours standing for free variables, which is impossible because of the semantics of judgements.
5. In spite of (4), these translations of secrecy typability are simply a different way of saying the same thing as the mnd-predicate for function types (assuming all free variables are λ-abstracted). Since any program can be transparently closed using a sequence of λ-abstractions without changing its essential meaning, we can in fact do without the translations, working directly with mnd-predicates and closed programs. Importantly this also enables us to appeal to purely syntactic proofs for noninterference through an alternative proof system, which will be discussed elsewhere.

The following is immediate from the definition.

Proposition 56. *Let* $\Gamma \models_{\mathsf{sec}} M : \llcorner \mathsf{Unit} \lrcorner_s$ *with* $\Gamma = \vec{x} : \vec{\alpha}$. *Then* $\mathsf{T} \models V_i, W_i :_{v_i, w_i} \mathtt{mnd}(v_i, w_i : \alpha_i, s)$ *implies* $M[\vec{V}/\vec{x}] \Downarrow$ *iff* $M[\vec{W}/\vec{x}] \Downarrow$.

5.3 Secrecy Typing and Noninterference

In this subsection, we first introduce the secrecy typing for call-by-value pure functions (as introduced in [24, §4.5]) together with associated notions, study their properties with respect to the MND-predicates and NI-predicates, and finally establish the soundness of the secrecy typing.

Fig. 11 Secrecy Typing Rules for Call-by-value DCC

$$[\mathit{Var}]\ \frac{x:\alpha \in \Gamma}{\Gamma \vdash_{\mathsf{sec}} x : (\alpha)_s} \qquad [\mathit{Op}]\ \frac{\Gamma \vdash_{\mathsf{sec}} M_i : \alpha_i \quad \mathsf{protect}(\beta) \sqsubseteq \mathsf{protect}(\alpha_i)}{\Gamma \vdash_{\mathsf{sec}} \mathsf{op}(\vec{M}) : \beta}$$

$$[\mathit{Abs}]\ \frac{\Gamma, x:\alpha \vdash_{\mathsf{sec}} M : \beta}{\Gamma \vdash_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \qquad [\mathit{App}]\ \frac{\Gamma \vdash V : \alpha \Rightarrow \beta \quad \Gamma \vdash_{\mathsf{sec}} W : \alpha}{\Gamma \vdash_{\mathsf{sec}} VW : \beta}$$

$$[\mathit{Rec}]\ \frac{\Gamma, x : \alpha \Rightarrow \beta \vdash_{\mathsf{sec}} \lambda y^{\alpha}.M : \alpha \Rightarrow \beta \quad \alpha \Rightarrow \beta \text{ pointed}}{\Gamma \vdash_{\mathsf{sec}} \mu x^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M : \alpha \Rightarrow \beta}$$

$$[\mathit{If}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \mathsf{Bool}_s \quad \Gamma \vdash_{\mathsf{sec}} N_{1,2} : \alpha \quad \boxed{s \sqsubseteq \mathsf{protect}(\alpha)}}{\Gamma \vdash_{\mathsf{sec}} \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha}$$

$$[\mathit{Inj}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha_i}{\Gamma \vdash_{\mathsf{sec}} \mathtt{in}_i(V) : \alpha_1 +_s \alpha_2}$$

$$[\mathit{Case}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha_1 +_s \alpha_2 \quad \Gamma, x_i : \alpha_i \vdash_{\mathsf{sec}} M_i : \beta \quad \boxed{s \sqsubseteq \mathsf{protect}(\beta)}}{\Gamma \vdash_{\mathsf{sec}} \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}} : \beta}$$

$$[\mathit{Pair}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V_i : \alpha_i\ (i = 1,2)}{\Gamma \vdash_{\mathsf{sec}} \langle V_1, V_2 \rangle : \alpha_1 \times \alpha_2} \qquad [\mathit{Proj}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha_1 \times \alpha_2}{\Gamma \vdash_{\mathsf{sec}} \pi_i(V) : \alpha_i\ (i = 1,2)}$$

$$[\mathit{Let}]\ \frac{\Gamma, x : \alpha \vdash_{\mathsf{sec}} M : \beta \quad \Gamma \vdash_{\mathsf{sec}} N : \alpha}{\Gamma \vdash_{\mathsf{sec}} \mathtt{let}\ x^{\alpha} = N\ \mathtt{in}\ M : \beta} \qquad [\mathit{Lift}]\ \frac{\Gamma \vdash_{\mathsf{sec}} M : \alpha}{\Gamma \vdash_{\mathsf{sec}} M : \llcorner\alpha\lrcorner_s}$$

$$[\mathit{Seq}]\ \frac{\Gamma \vdash_{\mathsf{sec}} N : \llcorner\alpha\lrcorner_s \quad \Gamma, x:\alpha \vdash_{\mathsf{sec}} M : \beta \quad \beta \text{ partial} \quad \boxed{s \sqsubseteq \mathsf{protect}(\beta)}}{\Gamma \vdash_{\mathsf{sec}} \mathtt{seq}\ x^{\alpha} = N\ \mathtt{in}\ M : \beta}$$

Secrecy Typing for Pure Functions We use the *protection level* [3], written $\mathsf{protect}(\alpha)$, which indicates the level at which data of that type may affect the environment [24].

$$\begin{aligned}
\mathsf{protect}(\mathsf{Unit}) &= \textsc{h} \\
\mathsf{protect}(\mathsf{Nat}_s) &= \mathsf{protect}(\alpha +_s \beta) \;=\; \mathsf{protect}(\llcorner\alpha\lrcorner_s) \;=\; s \\
\mathsf{protect}(\alpha \Rightarrow \beta) &= \mathsf{protect}(\beta) \\
\mathsf{protect}(\alpha \times \beta) &= \mathsf{protect}(\alpha) \sqcap \mathsf{protect}(\beta)
\end{aligned}$$

Further, α *raised at s*, written $(\alpha)_s$, which indicates the result of raising the outermost level of α, is given by: $(\mathsf{Unit})_s = \mathsf{Unit}$, $(\mathsf{Nat}_{s'})_s = \mathsf{Nat}_{s' \sqcup s}$, $(\alpha \Rightarrow \beta)_s = \alpha \Rightarrow (\beta)_s$, $(\alpha \times \beta)_s = (\alpha)_s \times (\beta)_s$, $(\alpha +_{s'} \beta)_s = \alpha +_{s' \sqcup s} \beta$ and $(\llcorner\alpha\lrcorner_{s'})_s = \llcorner\alpha\lrcorner_{s' \sqcup s}$.

The secrecy typing, with sequents $\Gamma \vdash_{\mathsf{sec}} M : \alpha$, replaces [*Var*, *Num*, *Bool*, *Op*, *If*, *Inj*, *Case*, *Lift*] and [*Seq*] with their secrecy annotated versions, leaving other rules as they are. For reference, we present the full rules in Figure 11. [*op*] is a rule schema to be instantiated into each concrete n-ary first-order operation (operations include constants when the arity is zero).

The marked part in each rule indicates the secrecy condition. Detailed illustration of each rule is given in [24, §3].

Soundness of Secrecy Typing For the proof of noninterference, we first clarify some of the basic properties of MND-predicates. First we start from some notations, which will be used throughout the noninterference proofs. Hereafter we assume all substitutions use closed values as their codomain. We recall $\Gamma \vdash \xi$ says ξ is typed under Γ.

Definition 57.

1. Let $\Gamma \vdash \xi$. Then we write $\Gamma \models_{\mathsf{sec}} \xi$ for $\models_{\mathsf{sec}} V : \alpha$ for each $\xi(x) = V$ with $\Gamma(x) = \alpha$.
2. $\mathtt{mnd}(M, N : \alpha, s)$ stands for $\models M, N :_{u,v} \mathtt{mnd}(u, v : \alpha, s)$ with M and N closed.
3. $\mathtt{mnd}(\xi_1, \xi_2 : \Gamma, s)$ means each pair of values in $\xi_{1,2}$ are thus related.

Lemma 58.

1. $\Gamma, x : \alpha \models_{\mathsf{sec}} M : \beta$ *iff* $\models_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \Rightarrow \beta$.
2. $\models_{\mathsf{sec}} M : \alpha$ *iff* $\models M :_u \mathtt{NI}(u : \alpha)$.
3. *If* $\Gamma \models_{\mathsf{sec}} M : \alpha$ *and* $\Gamma \models_{\mathsf{sec}} \xi$ *then* $\models M\xi :_u \mathtt{NI}(u : \alpha)$.

PROOF: All are direct from the definition. (3) uses (2).□

Notation 59. $x \bullet y_1..y_n$ stands for $(..(x \bullet y_1) \bullet y_2) \bullet \ldots \bullet y_n)$. $\beta_1..\beta_n \Rightarrow \alpha$ stands for $\beta_1 \Rightarrow (\beta_2 \Rightarrow ....(\beta_n \Rightarrow \alpha)..)$.

Lemma 60.

1. $\mathtt{NI}(u : \alpha)$ *is admissible at* u *for each* α.
2. *If* $\vec{y} : \vec{\beta}, x : \alpha \models_{\mathsf{sec}} M : \gamma$ *then* $\vec{y} : \vec{\beta}, z : \vec{\beta} \Rightarrow \alpha \models_{\mathsf{sec}} M[z \bullet \vec{y}/x] : \gamma$.
3. *If* $\models_{\mathsf{sec}} \mu z.\lambda\vec{y}.V[z \bullet \vec{y}/x] : \vec{\beta} \Rightarrow \alpha$ *then* $\models_{\mathsf{sec}} \lambda\vec{y}.\mu x.V : \vec{\beta} \Rightarrow \alpha$.

PROOF: For (1), we show the formula satisfies two conditions in Definition 49 (3).

- For the first condition, the least program Ω^{α} surely satisfies the predicate.
- For the second condition, we use induction on types to show $\mathtt{mnd}(\pi_1(u), \pi_1(u) : \alpha, s)$ has the approximation property for each α. Assume a chain $\{V_i\}$ (resp. $\{W_i\}$) approximates V_{ω} (resp. W_{ω}) and $\models V_i, W_i :_{v,w} \mathtt{mnd}(v, w : \alpha, s)$ for each i. The only interesting case is $\alpha = \beta \Rightarrow \llcorner\gamma\lrcorner$. Let $\models T, U :_{t,u} \mathtt{mnd}'(t, u : \beta, s)$. If $V_{\omega}T \Downarrow$ then by $V_{\omega}T$ being the lub of $\{V_iT\}_i$ we have $V_iT \Downarrow$ for some i hence $W_iU \Downarrow$. By induction we know $\models V_{\omega}T, W_{\omega}U :_{v,w} \mathtt{mnd}(v, w : \alpha, s)$. If $V_{\omega}T \Uparrow$ the result is immediate. The remaining cases are easy.

(2) is direct from the definition. (3) is standard [17, 54].□

Proposition 61.

1. $\mathsf{protect}((\alpha)_s) = \mathsf{protect}(\alpha) \sqcup s$.
2. $\forall u, v.\ (\mathtt{mnd}(u, v : \alpha, s) \supset \mathtt{mnd}(u, v : (\alpha)_{s'}, s))$ *for any* s'.
3. $\mathsf{protect}(\alpha) \not\sqsubseteq s \supset (\mathtt{NI}(u : \alpha) \wedge \mathtt{NI}(v : \alpha) \supset \mathtt{mnd}(u, v : \alpha, s))$.

PROOF: All are easy by induction on types.□

We are now ready to prove:

Theorem 62. (soundness) $\Gamma \vdash_{\mathsf{sec}} M : \alpha$ *implies* $\Gamma \models_{\mathsf{sec}} M : \alpha$.

PROOF: By rule induction on the secrecy typing rules in Figure 11. Below (IH) stands for "by induction hypothesis". (Def) stands for "by definition".

(Var) $\mathtt{mnd}'(\xi_1, \xi_2 : \Gamma, s)$

$$\begin{array}{ll} \supset \mathtt{mnd}'(x\xi_1, x\xi_2 : \alpha, s) & \text{(Definition)} \\ \supset \mathtt{mnd}'(x\xi_1, x\xi_2 : (\alpha)_{s'}, s) & \text{(Prop. 61 (3)).} \end{array}$$

(Op) Let T and T' be atomic types of level s'. Let op be the operation of type $\vec{T} \Rightarrow T'$. It suffices to show:

$$\mathtt{mnd}'(\vec{x}, \vec{y} : \vec{T}, s) \supset \mathtt{mnd}'(\mathsf{op}(\vec{x}), \mathsf{op}(\vec{y}) : T', s)$$

If $s' \sqsubseteq s$ then $\mathtt{mnd}'(\vec{x}, \vec{y} : T, s)$ means $\vec{x} = \vec{y}$ hence done. If $s' \not\sqsubseteq s$ then $\mathtt{mnd}'(\mathsf{op}(\vec{x}), \mathsf{op}(\vec{y}) : T', s)$ by Proposition 61 (2).

(Abs) Direct from Lemma 58 (1).

(App) $\mathtt{mnd}'(\xi_1, \xi_2 : \Gamma, s)$

$$\begin{array}{ll} \supset \mathtt{mnd}(V\xi_1, V\xi_2 : \alpha \Rightarrow \beta, s) \ \wedge & \\ \quad \mathtt{mnd}'(W\xi_1, W\xi_2 : \alpha, s) & \text{(IH, Lem. 58 (2))} \\ \supset \mathtt{mnd}((VW)\xi_1, (VW)\xi_2 : \beta, s) & \text{(Def).} \end{array}$$

In detail, the second inference uses the entailment:

$$\begin{array}{l} \mathtt{mnd}'(v, v' : \alpha \Rightarrow \beta, s) \wedge \mathtt{mnd}'(w, w' : \alpha, s) \\ \quad \equiv \quad \forall x^{\alpha}, y^{\alpha}.(\ \mathtt{mnd}'(x, y : \alpha, s) \supset \mathtt{mnd}(v \bullet x, v' \bullet y : \beta, s)\ ) \ \wedge \ \mathtt{mnd}'(w, w' : \alpha, s) \\ \quad \supset \quad \mathtt{mnd}(v \bullet w, v' \bullet w' : \beta, s). \end{array}$$

Note there is a close structural correspondence between this semantic part of the reasoning and the initial concrete part.

(Rec) We first claim, with α pointed:

Claim. $x : \alpha \models_{\mathsf{sec}} \lambda y.M : \alpha$ implies $\models_{\mathsf{sec}} \mu x.\lambda y.M : \alpha$.

The proof of the claim relies on the soundness of [Rec-$\Uparrow$].

$x : \alpha \models_{\mathsf{sec}} \lambda y.M : \alpha$

$$\begin{array}{ll} \supset \mathtt{NI}(x : \alpha) \models \lambda y.M :_u \mathtt{NI}(u : \alpha) & \text{(IH, Lem. 58 (3))} \\ \supset \models \mu x.\lambda y.M :_u \mathtt{NI}(u : \alpha)u & \text{(Lem. 60 (1), [Rec-}\Uparrow\text{])} \end{array}$$

as required (end of the proof of the Claim).

We can now infer as follows, setting $V \stackrel{\text{def}}{=} \lambda y.M$.

$\vec{w} : \vec{\beta}, x : \alpha \models_{\mathsf{sec}} V : \alpha$

$$\begin{array}{ll} \supset \vec{w} : \vec{\beta}, z : \vec{\beta} \Rightarrow \alpha \models_{\mathsf{sec}} V[z \bullet \vec{w}/x] : \alpha & \text{(Lem. 60 (2))} \\ \supset z : \vec{\beta} \Rightarrow \alpha \models_{\mathsf{sec}} \lambda \vec{w}.(V[z \bullet \vec{w}/x]) : \vec{\beta} \Rightarrow \alpha & \text{(Lem. 58 (1))} \\ \supset \models_{\mathsf{sec}} \mu z.\lambda \vec{w}.(V[z \bullet \vec{w}/x]) : \vec{\beta} \Rightarrow \alpha & \text{(Claim above)} \\ \supset \models_{\mathsf{sec}} \lambda \vec{w}.\mu x.V : \vec{\beta} \Rightarrow \alpha & \text{(Lem. 60 (3))} \\ \supset \vec{w} : \vec{\beta} \models_{\mathsf{sec}} \mu x.V : \alpha & \text{(Lem. 60 (1)).} \end{array}$$

(If) Subsumed by **(Case)** below.

(Inj) It suffices to show, for each $i \in \{1,2\}$:

$$\mathtt{mnd}(x,y\!:\!\alpha_i,\, s) \;\supset\; \mathtt{mnd}(\mathtt{in}_i(x),\mathtt{in}_i(y)\!:\!\alpha_1\!+_{s'}\!\alpha_2,\, s)$$

which is immediate from the definition.

(Case) Consider the sum type $\gamma = \alpha_1 +_{s'} \alpha_2$ (note s' replaces s in the typing rule). Accordingly the secrecy condition becomes $s' \sqsubseteq \mathsf{protect}(\beta)$. We have two cases. Below let $N \stackrel{\text{def}}{=} \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i\in\{1,2\}}$. If $s' \sqsubseteq s$, we infer:

$$\begin{array}{ll}
\mathtt{mnd}'(\xi,\xi'\!:\!\Gamma,\, s) & \\
\supset \mathtt{mnd}'(V\xi, V\xi'\!:\!\gamma,\, s) & \text{(IH, Lem. 58 (3))} \\
\supset V\xi \cong \mathtt{in}_i(W_i) \,\wedge\, V\xi' \cong \mathtt{in}_i(W_i') \,\wedge\, \mathtt{mnd}'(W_i, W_i'\!:\!\alpha_i,\, s) & (s' \sqsubseteq s) \\
\supset \mathtt{mnd}(M_i\xi[W_i/x_i], M_i\xi'[W_i'/x_i]\!:\!\beta,\, s) & \text{(IH)}
\end{array}$$

With i being either 1 or 2, hence done. On the other hand, if $s' \not\sqsubseteq s$, then $\mathsf{protect}(\beta) \not\sqsubseteq s$ by the secrecy condition $s' \sqsubseteq \mathsf{protect}(\beta)$, hence by Proposition 61 (3), we have $\mathtt{mnd}(N\xi, N\xi'\!:\!\beta,\, s)$ regardless of ξ,ξ'.

(Pair, Proj) Because $\mathtt{mnd}'(x_1,x_1'\!:\!\alpha_1,\, s) \wedge \mathtt{mnd}'(x_2,x_2'\!:\!\alpha_2,\, s) \;\equiv\; \mathtt{mnd}'(\langle x_1,x_2\rangle, \langle x_1',x_2'\rangle\!:\!\alpha_1 \times \alpha_2,\, s)$ (which is immediate from the definition).

(Let) Subsumed by **(Seq)** below.

Note. In **(Lift)** and **(Seq)** below, we assume the mnd-predicate for partial types, $\mathtt{mnd}(e,e'\!:\!\llcorner\alpha\lrcorner_{s'},\, s)$, given as:

$$s' \sqsubseteq s \;\supset\; (e\Downarrow \equiv e'\Downarrow) \,\wedge\, \forall x^{\alpha}, y^{\alpha}.((e = \llcorner x \lrcorner \wedge e' = \llcorner y \lrcorner) \;\supset\; \mathtt{mnd}(x,y\!:\!\alpha,\, s))$$

(Lift) By $\mathtt{mnd}(x,y\!:\!\alpha,\, s) \supset \mathtt{mnd}(\llcorner x\lrcorner, \llcorner y\lrcorner\!:\!\llcorner\alpha\lrcorner_{s'},\, s)$.

(Seq) Let $L \stackrel{\text{def}}{=} \mathtt{seq}\ x = N\ \mathtt{in}\ M$ and $\gamma \stackrel{\text{def}}{=} \llcorner\alpha\lrcorner_{s'}$ (note s' replaces s in the typing rule). The secrecy condition becomes $s' \sqsubseteq \mathsf{protect}(\beta)$. First, if $s' \sqsubseteq s$, we infer:

$$\begin{array}{ll}
\mathtt{mnd}'(\xi,\xi'\!:\!\Gamma,\, s) & \\
\supset p\mathtt{mnd}(N\xi, N\xi'\!:\!\gamma,\, s) & \text{(IH, Lem. 58 (3))} \\
\supset \mathtt{mnd}(N\xi, N\xi'\!:\!\alpha,\, s) \,\vee\, (N\xi\Uparrow \wedge N\xi'\Uparrow) & \text{(Def)} \\
\supset \mathtt{mnd}(M\xi[N\xi/x], M\xi'[N\xi'/x]\!:\!\alpha,\, s) \,\vee\, (L\xi\Uparrow \wedge L\xi'\Uparrow) & \text{(IH)} \\
\supset \mathtt{mnd}(L\xi, L\xi'\!:\!\beta,\, s) & \text{(Def)}
\end{array}$$

If $s' \not\sqsubseteq s$, we have $\mathsf{protect}(\beta) \not\sqsubseteq s$ by the secrecy condition $s' \sqsubseteq \mathsf{protect}(\beta)$, hence by Prop. 61 (3) $\mathtt{mnd}(L\xi, L\xi'\!:\!\beta,\, s)$ regardless of ξ,ξ'. We have now exhausted all cases. $\square$

Note the theorem implies $\vdash_{\text{sec}} M : \alpha \supset \mathsf{T} \models M :_m \mathtt{NI}(m\!:\!\beta)$. Now let $\cong_s$ for each s be the maximum typed congruence on securely typed programs such that, whenever $\vdash M_{1,2} : \mathsf{Unit} +_s \mathsf{Unit}$, we have $M_1\Downarrow$ iff $M_2\Downarrow$. From Proposition 56 and Theorem 62 we conclude:

Corollary 63. (noninterference) *Let* $\Gamma \vdash_{\text{sec}} M : \llcorner\mathsf{Unit}\lrcorner_s$ *with* $\Gamma = \vec{x}:\vec{\alpha}$ *and let* $\vdash V_i \cong_s W_i : \alpha_i$ *for each i. Then we have* $M[\vec{V}/\vec{x}]\Downarrow$ *iff* $M[\vec{W}/\vec{x}]\Downarrow$.

The corollary implies a weaker form where α is replaced by ground types (such as Nat_s), which is often used in the literature.

5.4 Global State (1): Preparation

In this subsection and the next, we extend the technical development for the description and justification of secrecy typing in the preceding subsections to the language with global state. Since we have already introduced the refinement to the language and logic with global state with pointed types and generalised total correctness in §5.1, we only have to present the secrecy typing for global state and its representation (and justification) in the logic. Fortunately, these refinements can be done incrementally on the basis of our preceding development, with, among others, a very close parallel in proof methods. In the following technical development, we recall:

1. As stipulated in§5.1, effects on arrow types are given type annotations for the purpose of preserving totality in the face of reference agents (though for brevity we shall often treat effects of an arrow type as if they were a sequence of plain names).
2. $\xi, \xi', \ldots$ range over substitutions for semi-closed values while $\sigma, \sigma', \ldots$ range over stores, i.e. maps from reference typed names to semi-closed values. We continue to write $\mathtt{mnd}(\xi_1, \xi_2 : \Gamma, s)$ and $\mathtt{mnd}(\sigma_1, \sigma_2 : \Gamma, s)$ for the conjunction of component-wise mnd-relationship.

In addition to the purely functional secrecy-annotated types, we annotate a reference type with a secrecy level. Further we add the "read-only" reference type which is important for accurate flow analysis.

$$\rho \ ::= \ \ldots \mid \mathsf{Refr}_s(\alpha) \mid \mathsf{Ref}_s(\alpha)$$

Protection levels are also refine, which now include "contravariant protection levels" $\mathsf{protectE}(\tau)$. We also need a basic condition, the structural security (the significance and implication of this condition is discussed in derail in [24]).

Definition 64. (protection levels and structural security)

1. We set $\mathsf{protect}(\mathsf{Refr}_s(\alpha)) = \mathsf{protect}(\mathsf{Ref}_s(\alpha)) = \mathsf{protect}(\gamma \overset{\varphi}{\Rightarrow} \alpha) = \mathsf{protect}(\alpha)$.
2. $\mathsf{Refr}_s(\alpha)$ and $\mathsf{Ref}_s(\alpha)$ are *structurally secure* iff $s \sqsubseteq \mathsf{protect}(\alpha)$.
3. $\mathsf{protectE}(\rho)$ is given as follows.

$$\mathsf{protectE}(\rho) \ = \ \begin{cases} s & (\rho = \mathsf{Ref}_s(\alpha)) \\ \mathrm{H} & (\text{if else}) \end{cases}$$

Further $\mathsf{protectE}(\Gamma) = \sqcap\{\mathsf{protectE}(\alpha) \mid \alpha \in \mathsf{ran}(\Gamma)\}$.

Convention 65. We only consider structurally secure reference types from now on.

Remark 66. (protection levels) The definitions of the (covariant and contravariant) protection levels, $\mathsf{protect}(\alpha)$ and$\mathsf{protectE}(\alpha)$, are much simpler than what is presented in [24] (and which is reproduced in §5.6 later). In particular, we have no secrecy annotation on arrow types. This is because, when we are solely working in global state, effects given by a program are always visible, so that there is no need to record them separately. However if we are to use an explicit secrecy level rather than the present implicit one (in the terminology of [24]), We next refine the mnd-predicates. The only change we need is in arrow types. For readability we only consider a single reference/variable, omit type annotations and write $\mathtt{NI}(\vec{x})$ for the conjunction of the NI-predicates. Similarly for $\mathtt{mnd}(xv, x'v', s)$. We set, with $\gamma = \alpha \overset{\vec{y}}{\Rightarrow} \beta$:

$$\mathtt{mnd}(e, e' : \gamma, s) \stackrel{\text{def}}{=} \begin{cases} \Phi_{\Downarrow}(e, e' : \gamma, s) & (\beta \text{ total}) \\ \Phi_{\Downarrow}(e, e' : \gamma, s) \wedge \Phi_{\Uparrow}(e, e' : \gamma, s) & (\beta \text{ partial}) \end{cases}$$

where $\Phi_{\Downarrow}(e, e' : \alpha \overset{\vec{y}}{\Rightarrow} \beta, s)$ is given as:

$$\begin{array}{l} \forall x x' z z' \vec{v}\vec{v}' \vec{w}\vec{w}'.((\mathtt{NI}(xx'\vec{v}\vec{v}') \wedge \mathtt{mnd}(x\vec{v}, x'\vec{v}', s) \wedge \\ \quad \{!\vec{y} = \vec{v}\}\, e \bullet x \searrow u\, \{u = z \wedge !\vec{y} = \vec{w}\} \wedge \\ \quad \{!\vec{y} = \vec{v}'\}\, e' \bullet x' \searrow u\, \{u = z' \wedge !\vec{y} = \vec{w}'\}\ ) \supset \mathtt{mnd}(z\vec{w}, z'\vec{w}' : \beta\vec{\gamma}, s)\,) \end{array}$$

The assertion reads: *if the initial state and argument are s-indistinguishable, then the results should be s-indistinguishable.* In other words, it says that if the environments are extensionally s-related, then the results are again extensionally s-related. If β is partial, $\Phi_{\Uparrow}(e, e' : \alpha \overset{\vec{y}}{\Rightarrow} \beta, s)$, with β partial, is given as:

$$\begin{array}{l} \forall x x' \vec{v}\vec{v}'.\ ((\mathtt{NI}(xx'\vec{v}\vec{v}') \wedge \mathtt{mnd}(x\vec{v}, x'\vec{v}', s)) \quad \supset \\ \qquad\qquad \{!\vec{y} = \vec{v}\}\, e \bullet x \Uparrow\ \equiv\ \{!\vec{y} = \vec{v}'\}\, e' \bullet x' \Uparrow\,) \end{array}$$

We can now present a logical translation of the secrecy typing. The typing uses two sequents, $\Gamma \vdash_{\mathsf{sec}} M : \alpha$ and $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$. The former is translated precisely as before. For the latter, let $\Delta = \Gamma \restriction \mathsf{dom}_f(\Gamma)$ and $\Theta = \Gamma \restriction \varphi$. Then, if α is total, $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ is translated into:

> *For each s, whenever* $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ *and* $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$*, if* $\langle M\xi_1, \sigma_1\rangle \Downarrow \langle V_1, \sigma_1'\rangle$ *and* $\langle M\xi_2, \sigma_2\rangle \Downarrow \langle V_2, \sigma_2'\rangle$ *then* $\mathtt{mnd}'(V_1, V_2 : \alpha, s)$ *and* $\mathtt{mnd}'(\sigma_1', \sigma_2' : \Theta, s)$.

If β is partial, we translate $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ into the conjunction of the above statement and the following one:

> *For each s, whenever* $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ *and* $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$*, if* $\mathsf{protect}(\beta) \sqsubseteq s$*, then* $\langle M\xi_1, \sigma_1\rangle \Uparrow$ *iff* $\langle M\xi_2, \sigma_2\rangle \Uparrow$.

The translation simply increments the one for stateless behaviour with writing effects, saying: *if* the input vary up to s, *then* their effects, in convergence/divergence and the return value/variable content, should only vary up to s.

Remark 67. (translation of predicate in the case of explicit secrecy annotation) In the technical development in the work by Myers [39] (using Java) and Pottier and Simonet [47], (using ML), the writing level of a closure is recorded explicitly in the arrow type. It is notable that there is *no* change in the definition of MND-predicates: the explicit secrecy annotation offers another (and probably refined) way to ensure secrecy, but the guaranteed property itself remains identical, demonstrating the c generality of the logical description.

Fig. 12 Secrecy Typing Rules with Global State

$$[\mathit{If}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \mathsf{Bool}_s \quad \Gamma \vdash_{\mathsf{sec}} N_{1,2} : \alpha, \varphi \qquad s \sqsubseteq \mathsf{protect}(\alpha) \sqcap \mathsf{protectE}(\Gamma \upharpoonright \varphi)}{\Gamma \vdash_{\mathsf{sec}} \mathtt{if}\ V\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \alpha}$$

$$[\mathit{Case}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha_1 +_s \alpha_2 \quad \Gamma, x_i : \alpha_i \vdash_{\mathsf{sec}} M_i : \beta, \varphi \qquad s \sqsubseteq \mathsf{protect}(\beta) \sqcap \mathsf{protectE}(\Gamma \upharpoonright \varphi)}{\Gamma \vdash_{\mathsf{sec}} \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}} : \beta, \varphi}$$

$$[\mathit{Seq}]\ \frac{\Gamma \vdash_{\mathsf{sec}} N : \llcorner\alpha\lrcorner_s, \varphi_1 \quad \Gamma, x : \alpha \vdash_{\mathsf{sec}} M : \beta, \varphi_2 \quad \beta \text{ partial} \qquad s \sqsubseteq \mathsf{protect}(\beta) \sqcap \mathsf{protectE}(\Gamma \upharpoonright \varphi_2)}{\Gamma \vdash_{\mathsf{sec}} \mathtt{seq}\ x = N\ \mathtt{in}\ M : \beta, \varphi_1 \cup \varphi_2}$$

$$[\mathit{Deref}]\ \frac{\Gamma \vdash_{\mathsf{sec}} x : \mathsf{Refr}_s(\alpha)}{\Gamma \vdash_{\mathsf{sec}} !x : \alpha, \{x\}} \qquad [\mathit{Assign}]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha \quad \Gamma \vdash_{\mathsf{sec}} x : \mathsf{Ref}_s(\alpha)}{\Gamma \vdash_{\mathsf{sec}} x := V : \alpha, \{x\}}$$

$$[\mathit{Weak\text{-}write}]\ \frac{\Gamma, x : \mathsf{Refr}_s(\alpha) \vdash_{\mathsf{sec}} M : \alpha, \varphi}{\Gamma, x : \mathsf{Ref}_s(\alpha) \vdash_{\mathsf{sec}} M : \alpha, \varphi}$$

We present the secrecy typing for global state in Figure 12. (Var), (Op), (Abs), (App), (Rec), (Promote), (Inj), (Pair), (Proj), (Let) and (Weak) stay as in Figure 11. It is notable that totality ensured by Convention 42 (2) is crucial for secrecy typing (as is clear from the distinct treatment of MND-predicates for total and partial arrow types).[5]

We extend Lemmas 58 and 60, after a refinement of Definition 57.

Definition 68.

1. Let $\Gamma \vdash \xi$. Then we write $\Gamma \models_{\mathsf{sec}} \xi$ for $\models_{\mathsf{sec}} V : \alpha$ for each $\xi(x) = V$ with $\Gamma \vdash V : \alpha$. Let $\Gamma \vdash \sigma$. Then we write $\Gamma \models_{\mathsf{sec}} \sigma$ for $\models_{\mathsf{sec}} V : \alpha$ for each $\sigma(x) = V$ with $\Gamma \vdash V : \alpha$.
2. $\mathtt{mnd}(V, W : \alpha, s)$ stands for $\models V, W :_{u,v} \mathtt{mnd}(u, v : \alpha, s)$ with V and W closed.
3. $\mathtt{mnd}(\xi_1, \xi_2 : \Gamma, s)$ means each pair of values in $\xi_{1,2}$ are thus related, at the same time assuming $\mathsf{dom}(\xi_1) = \mathsf{dom}(\xi_2)$ and $\xi_{1,2}$ are under the same typing. Similarly for $\mathtt{mnd}(\sigma_1, \sigma_2 : \Gamma, s)$.
4. With $\Gamma \vdash M_{1,2}$ semi-closed, $\mathtt{mnd}(M_1, M_2 : \alpha, s)$ with α total stands for: whenever $\mathtt{mnd}(\sigma_1, \sigma_2 : \Gamma, s)$ and $(M_i, \sigma_i) \Downarrow (V_i, \sigma_i')$ $(i = 1, 2)$, we have $\mathtt{mnd}(V_1, V_2 : \alpha, s)$ and $\mathtt{mnd}(\sigma_1', \sigma_2' : \Gamma, s)$.
5. With $\Gamma \vdash M_{1,2}$ semi-closed, $\mathtt{mnd}(M_1, M_2 : \llcorner\alpha\lrcorner_{s'}, s)$ stands for: whenever $\mathtt{mnd}(\sigma_1, \sigma_2 : \Gamma, s)$, we have: (1) If $(M_i, \sigma_i) \Downarrow (V_i, \sigma_i')$ $(i = 1, 2)$ then $\mathtt{mnd}(V_1, V_2 : \alpha, s)$ and $\mathtt{mnd}(\sigma_1', \sigma_2' : \alpha, s)$; and (2) If $s' \sqsubseteq s$ then $(M_1, \sigma_1) \Uparrow$ iff $(M_2, \sigma_2) \Uparrow$.
6. Let $\mathsf{dom}(()\sigma_1) = \mathsf{dom}(()\sigma_2)$ and $\Gamma \vdash \sigma_{1,2}$. Then we write $\Gamma \vdash \sigma_1 =_s \sigma_2$ when for each reference whose level is the same as, or lower then, s, their content values coincide. Similarly we write $\Gamma \vdash \sigma_1 \cong_s \sigma_2$ if each variable

[5] In [24], the guarantee of totality is ensured by not allowing direct access to references in the body of abstraction. The present formulation, based on type annotation on effects, practically subsumes the older condition.

In (5, 6) below, $\beta_1..\beta_n \Rightarrow \alpha$ stands for $\beta_1 \stackrel{\emptyset}{\Rightarrow} (..\beta_n \stackrel{\emptyset}{\Rightarrow} \alpha)..)$.

Lemma 69. *Assume Θ (resp. Γ) uses reference types (resp. non-reference types) as its co-domain.*

1. $\Gamma, x:\alpha, \Theta \models_{\mathsf{sec}} M : \beta, \varphi$ *iff* $\Gamma, \Theta \models_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \stackrel{\varphi}{\Rightarrow} \beta$.
2. $\Theta \models_{\mathsf{sec}} M : \alpha$ *iff* $\Theta \models M :_u \mathtt{NI}(u:\alpha)$. *If* $\Theta \models_{\mathsf{sec}} M : \alpha, \varphi$, $\Theta \models_{\mathsf{sec}} \sigma :$ *and* $\langle M, \sigma\rangle \Downarrow \langle V, \sigma'\rangle$, *then* $\Theta \models V :_u \mathtt{NI}(u:\alpha)$.
3. *If* $\Gamma, \Theta \models_{\mathsf{sec}} M : \alpha$ *and* $\Gamma, \Theta \models_{\mathsf{sec}} \xi$ *then we have:* $\Theta \models M\xi :_u \mathtt{NI}(u:\alpha)$. *Similarly if* $\Gamma, \Theta \models_{\mathsf{sec}} M : \alpha, \varphi$, $\Gamma, \Theta \models_{\mathsf{sec}} \xi$, $\Theta \models_{\mathsf{sec}} \sigma$ *and* $\langle M\xi, \sigma\rangle \Downarrow \langle V, \sigma'\rangle$, *then we have:* $\Theta \models V :_u \mathtt{NI}(u:\alpha)$.
4. $\mathtt{NI}(u:\alpha)$ *is admissible at u for each* α.
5. *If* $\Theta, \vec{y}:\vec{\beta}, x:\alpha \models_{\mathsf{sec}} V : \gamma$ *then we have:* $\Theta, \vec{y}:\vec{\beta}, z:\vec{\beta} \Rightarrow \alpha \models_{\mathsf{sec}} V[z \bullet \vec{y}/x] : \gamma$.
6. $\models_{\mathsf{sec}} \mu z.\lambda\vec{y}.V[z \bullet \vec{y}/x] : \vec{\beta} \Rightarrow \alpha \ \supset\ \models_{\mathsf{sec}} \lambda\vec{y}.\mu x.V : \vec{\beta} \Rightarrow \alpha$.
7. *Let* $\Gamma \vdash M : \alpha, \varphi$ *be semi-closed and* $\Gamma \vdash \sigma$ *such that* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma)$. *Then* $\mathsf{protectE}(\Gamma \upharpoonright \varphi) \not\sqsubseteq s$ *and* $(M, \sigma) \Downarrow (V, \sigma')$ *imply* $\sigma =_s \sigma'$.

All proofs are easy. (5) is justified via the contextual congruence between two programs, using their process encodings and replication theorems. (7) is because such a process never writes on s-non-observable references.

We can now prove to the proof of soundness with global state.

5.5 Global State (2): Proof of Noninterference

Theorem 70. $\Gamma \vdash_{\mathsf{sec}} M : \alpha$ *implies* $\Gamma \models_{\mathsf{sec}} M : \alpha$. *Similarly* $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ *implies* $\Gamma \models_{\mathsf{sec}} M : \alpha, \varphi$.

PROOF: By rule induction. **(Var, Op, Abs, Rec, Pair, Proj, Inj, Lift)** precisely follow those in Theorem 3, using the corresponding statements in Lemma 69. Below metavariables for types etc. are from each secrecy typing rule. Throughout we let $\Delta \stackrel{\text{def}}{=} \Gamma \upharpoonright \mathsf{dom}_f(\Gamma)$ and $\Theta \stackrel{\text{def}}{=} \Gamma \upharpoonright \varphi$.

(App) Assume β is partial (which subsumes the total case).
$\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s) \wedge \mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$

$$\begin{array}{ll}
\supset \mathtt{mnd}(V\xi_1, V\xi_2 : \alpha \stackrel{\varphi}{\Rightarrow} \beta, s) \ \wedge\ \mathtt{mnd}'(W\xi_1, W\xi_2 : \alpha, s) & \text{(IH, Lem. 69 (2))} \\
\supset \wedge_{i=1,2} \langle (VW)\xi_i, \sigma_i\rangle \Uparrow \ \vee & \\
\quad (\ \wedge_{i=1,2} \langle (VW)\xi_i, \sigma_i\rangle \Downarrow \langle U_i, \sigma_i'\rangle \ \wedge & \\
\qquad \mathtt{mnd}(U_1, U_2 : \beta, s) \ \wedge\ \mathtt{mnd}(\sigma_1', \sigma_2' : \Theta, s)\ ) & \text{(Def).}
\end{array}$$

(If) Subsumed by **(Case)** below.

(Case) Let $N \stackrel{\text{def}}{=} \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}}$, $\gamma = \alpha_1 +_{s'} \alpha_2$ and β be partial (subsuming the total case). If $s' \sqsubseteq s$:

$$
\begin{array}{ll}
\mathtt{mnd}'(\xi_1,\xi_2:\Delta,s) \wedge \mathtt{mnd}'(\sigma_1,\sigma_2:\Theta,s) & \\
\supset \mathtt{mnd}'(V\sigma_1,V\sigma_2:\gamma,s) & quad(\text{IH, Lem. 58 (3)}) \\
\supset \wedge_{i=1,2}\, V\sigma_i \cong \mathtt{in}_j(W_i) \wedge & \\
\quad \mathtt{mnd}'(W_1,W_2:\alpha_j,s) & (s' \sqsubseteq s) \\
\supset (\wedge_{i=1,2}\, \langle M_j\sigma_i[W_i/x_j],\sigma_i\rangle \Downarrow \langle U_i,\sigma_i'\rangle \wedge & \\
\quad \mathtt{mnd}(U_1,U_2:\beta,s) \wedge \mathtt{mnd}(\sigma_{1j}',\sigma_{2j}':\Theta,s)) \vee & \\
\quad \wedge_{i=1,2}\, \langle M_j\sigma_i[W_i/x_j],\sigma_i\rangle \Uparrow & (\text{IH})
\end{array}
$$

with j being either 1 or 2. If $s' \not\sqsubseteq s$, we have $\mathsf{protect}(\beta) \not\sqsubseteq s$ and $\mathsf{protectE}(\Gamma \restriction \varphi) \not\sqsubseteq s$ from the secrecy condition, hence neither convergence/divergence nor, when both converge, any state change is s-distinguishable, the latter by Lemma 69 (7).

(Let) Subsumed by **(Seq)** below.

(Seq) Let $L \stackrel{\text{def}}{=} \mathtt{seq}\ x = N\ \mathtt{in}\ M$ and $\gamma \stackrel{\text{def}}{=} \llcorner\alpha\lrcorner_{s'}$ (note s' replaces s in the typing rule). The secrecy condition becomes $s' \sqsubseteq \mathsf{protect}(\beta)$. First, if $s' \sqsubseteq s$, we infer:

$$
\begin{array}{ll}
\mathtt{mnd}'(\xi_1,\xi_2:\Delta,s) \wedge \mathtt{mnd}'(\sigma_1,\sigma_2:\Theta,s) & \\
\supset \mathtt{mnd}(N\xi_1,N\xi_2:\gamma,s) & (\text{IH, Lem. 58 (3)}) \\
\supset \mathtt{mnd}(N\xi_1,N\xi_2:\alpha,s) \vee ((N\xi_1,\sigma_1)\Uparrow \wedge (N\xi_2,\sigma_1)\Uparrow) & (\text{Def}) \\
\supset (\wedge_{i=1,2}\langle M\xi_i[N\xi_i/x],\sigma_i\rangle \Downarrow \langle V_i,\sigma_i'\rangle \wedge & \\
\quad \mathtt{mnd}(V_1,V_2:\alpha,s) \wedge \mathtt{mnd}(\sigma_1',\sigma_2':\Delta,s)) \vee \wedge_{i=1,2} L\sigma_i \Uparrow & (\text{IH})
\end{array}
$$

The case $s' \not\sqsubseteq s$ is identical as **Case** above.

(Deref) Direct from the structural security.

(Assign) $\mathtt{mnd}'(\xi_1,\xi_2:\Delta,s) \wedge \mathtt{mnd}'(\sigma_1,\sigma_2:\Theta,s)$

$$
\begin{array}{ll}
\supset \mathtt{mnd}(V\xi_1,V\xi_2:\alpha,s) & (\text{IH}) \\
\supset \mathtt{mnd}(\sigma_1[x\mapsto V\xi_1],\sigma_2[x\mapsto V\xi_2]:\alpha,s) &
\end{array}
$$

(Weak, Weak-write) Each immediate from the induction hypothesis. We have now exhausted all cases. □

5.6 Local State (1): Preparation

The introduction of local state makes secrecy analysis subtle, since effects can now be on hidden state. For example, a procedure $\lambda f^{\mathsf{Unit}\Rightarrow\mathsf{Unit}}.f()$ may change a shared hidden low-level state which f has access to and which can later be used by another procedure. We follow the approach in [24] to treat this issue, annotating arrow types by the levels, so that the above procedure is typed as $(\mathsf{Unit} \Rightarrow_{\mathrm{L}} \mathsf{Unit}) \Rightarrow_{\mathrm{H}} \mathsf{Unit}$; then, by taking the level of types in contravariant positions into account, the protection level of this type becomes low.

The integration of pointed types and opaque types is done simply by adding $\llcorner\alpha\lrcorner$ to the grammar of types in §4.3 (which becomes $\llcorner\alpha\lrcorner_s$ for secrecy typing):

$$\alpha,\beta ::= \ \ldots \llcorner\alpha\lrcorner$$

Fig. 13 Secrecy Typing Rules with Local State (refined rules)

$$[Abs]\ \frac{\Gamma, x\!:\!\alpha \vdash_{\mathsf{sec}} M : \beta, \varphi \quad s \sqsubseteq \mathsf{protectE}(\Gamma \upharpoonright \varphi)}{\Gamma \vdash_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \overset{\varphi}{\Rightarrow}_s \beta}$$

The grammar for ρ and δ stays the same. The (non-secrecy) typing rules integrate Figure 5 and Figure 8. On this basis, the proof rules combine the proof rules for local state and those in Figure 9.

To incorporate local state into secrecy typing, we first refine the arrow type to record the secrecy level, adding:

$$\alpha \overset{\varphi}{\Rightarrow}_s \beta$$

The level s in $\alpha \overset{\varphi}{\Rightarrow}_s \beta$ says when this closure is invoked, it may affect the environment at the level as low as s. We also retain the purely function arrow type $\alpha \Rightarrow \beta$ for a technical convenience, for which the logic can assert using the application operator, $e \bullet e'$.[6]

The contravariant protection levels are now defined in their general form (the definition of $\mathsf{protect}(\alpha)$ stays the same except the refined arrow type, given below).

$$\begin{aligned}
\mathsf{protect}(\alpha \overset{\varphi}{\Rightarrow}_s \beta) &= \mathsf{protectE}(\alpha) \sqcap \mathsf{protect}(\beta)\\
\mathsf{protectE}(\alpha \overset{\varphi}{\Rightarrow}_s \beta) &= s\\
\mathsf{protectE}(\mathsf{Nat}_s) &= \mathsf{protectE}(\mathsf{Bool}_s) \;=\; \mathrm{H}\\
\mathsf{protectE}(\alpha \times \beta) &= \mathsf{protectE}(\alpha) \sqcap \mathsf{protectE}(\beta)\\
\mathsf{protectE}(\alpha +_s \beta) &= \mathsf{protectE}(\alpha) \sqcap \mathsf{protectE}(\beta)\\
\mathsf{protectE}(\llcorner \alpha \lrcorner_s) &= \mathsf{protectE}(\alpha)\\
\mathsf{protectE}(\mathsf{Ref}_s(\alpha)) &= s\\
\mathsf{protectE}(\mathsf{Refr}_s(\alpha)) &= \mathsf{protectE}(\alpha)
\end{aligned}$$

Remark 71. (conservativity) Regarding $\alpha \overset{\varphi}{\Rightarrow} \beta$ in the system with global state as $\alpha \overset{\varphi}{\Rightarrow}_{\mathrm{H}} \beta$ in the present system, the covariant/contravariant protection levels of each α in the original system coincide with those in the present system.

We again assume the structural security, which now says:

Definition 72. $\mathsf{Ref}_s(\alpha)$ is *structurally secure* when $s \sqsubseteq \mathsf{protect}(\alpha) \sqcap \mathsf{protectE}(\alpha)$. $\mathsf{Refr}_s(\alpha)$ is *structurally secure* when $s \sqsubseteq \mathsf{protect}(\alpha)$.

The notion conservatively extends the original one (i.e. $\mathsf{Ref}_s(\alpha)$ is structurally secure in the system with global state iff it is so in the system with local state). *From now on, we assume all reference types are structurally secure.*

[6] With global state, $\alpha \Rightarrow \beta$ has a faithful embedding as $\alpha \overset{\emptyset}{\Rightarrow} \beta$. However $\alpha \overset{\emptyset}{\Rightarrow}_s \beta$ in the present system may allow writing at a hidden name.

Under this refined notion of covariant/contravariant protection levels, the secrecy typing rules for existing programs stay precisely the same except, in (Abs), we record the secrecy level. We list the refined rule in Figure 13. Note however the ways to calculate protection levels in [*Case*], [*If*] and [*Seq*] are updated. The "new" is treated by the same rule as the non-secrecy typing.

$$[New]\ \frac{\Gamma \vdash_{\mathsf{sec}} V : \alpha \quad \Gamma, x : \mathsf{Ref}_s(\alpha) \vdash_{\mathsf{sec}} N : \beta, \varphi}{\Gamma \vdash_{\mathsf{sec}} \mathtt{new\ var}\ x := V\ \mathtt{in}\ N : \beta \setminus x, \varphi \setminus x}$$

We also use a rule for the purely functional abstraction, which infers $\Gamma \vdash M : \alpha \Rightarrow \beta$ from $\Gamma, x{:}\alpha \vdash M : \beta$ (note the latter has no effect).

We next augment the predicates for noninterference. Suppose we are given two evaluated programs, M_1 and M_2, named as u_1 and u_2. We first need to open these u_1 and u_2 using opening formulae, then relate u_1 and u_2 as well as a pair (or pairs) of local references.[7] If u_1 and u_2 have arrow types, relating u_1 and u_2 means they act in the same way as far as their local states, global states and input data are related. In the following technical development, the mnd-predicate $\mathtt{mnd}(u_1, u_2 : \alpha, s)$ assumes u_1 and u_2 are already opened, assuming the opening is given beforehand.

By assuming related behaviours are already opened, the only change is in the arrow type. We introduce several notations. Firstly, the predicate:

$$\{C\}e \bullet e' \searrow (\nu \vec{y}) z \{C'\} \tag{5.2}$$

stands for $\{C\}e \bullet e' \searrow z\{(\nu \vec{y}) z\vec{w} \text{ in } C'\}$ where $\vec{w}$ exhausts all primary names in C'. Secondly, we write

$$\{C\}e \bullet e' \Downarrow \tag{5.3}$$

for the negation of $\{C\}e \bullet e' \Uparrow$ when the result is a partial type: if not, (5.3) is set to be T.

We now give the predicate for arrow types, again dividing them into two, one for convergence and another for divergence. For legibility we present the case when there is a single shared (global) effect name, a single local effect name, and a single added local name ($\vec{y}$ in (5.3) above). By changing each single name to a vector of names, the definition easily extends to the general case.

1. $\Phi_{\Downarrow}(x_1 : \alpha \overset{y;l_1}{\Rightarrow}_{s'} \beta, x_2 : \alpha \overset{y;l_2}{\Rightarrow}_{s'} \beta, s)$ where β is either total or partial, which says two functions behave in the same way when they terminate.
2. $\Phi_{\Uparrow}(x_1 : \alpha \overset{y;l_1}{\Rightarrow}_{s'} \beta, x_2 : \alpha \overset{y;l_2}{\Rightarrow}_{s'} \beta, s)$ in which β is partial, which says two functions behave in the same way in divergence/convergence.

[7] Here it is assumed that the local references hidden by u_1 and those by u_2 are in precise correspondence with each other, in both number and types. This does not lose generality because of the the closure of semantics of formulae by $\cong$. In particular these local names may not directly correspond to the syntactic structure of programs.

First we define the former. Below let the type of $t_{1,2}$ (the content of local states) to be γ and let $s'' \stackrel{\text{def}}{=} \mathsf{protect}(\alpha \stackrel{y;l_1}{\Rightarrow}_{s'} \beta)$.

$$
\begin{array}{l}
\forall \vec{u}\vec{v}\vec{w}\vec{t}.\ (\ (\mathtt{mnd}'(u_1v_1, u_2v_2, s) \wedge \mathtt{mnd}'(t_1, t_2 : (\gamma)_{s''}, s) \wedge \\
\quad \bigwedge_{i=1,2}\{!y = u_i \wedge !l_i = t_i\} x_i \bullet v_i \Downarrow) \\
\qquad \supset\ \{!y = u_1 \wedge !l_1 = t_1\} x_1 \bullet v_1 \searrow (\nu z_1) w_1 \\
\qquad\qquad \{\ \{!y = u_2 \wedge !l_2 = t_2\} x_2 \bullet v_2 \searrow (\nu z_2) w_2 \\
\qquad\qquad\quad \{\mathtt{mnd}'(!z_1 !y_1 !l_1 w_1, !z_2 !y_2 !l_2 w_2, s)\}\ \}\ )
\end{array}
$$

Some observations on the above predicate follow.

Remark 73.

1. The mnd-predicate for $t_{1,2}$ is specifically treated since, as can be seen, their indistinguishability is treated by raising their level to s''. Since the datum of the level s'' does not affect below s'' or at incompatible levels, it is safe to consider the indistinguishability of hidden state up to this level. The same local inflation is done for all hidden names.
2. Either y (or its vector generalisation) is total or local, we demand the resulting states, as well as returned values, to be s-related. In addition, we require the new local states to be s-related. This is essential since these local values store the result of computation, to influence their external behaviour later.
3. We need nesting of evaluation formulae since we first need to open w_1 and w_2, as well as (possibly) y_1 and y_2, the latter now possibly holding, for example, these returned values.
4. Since types of hidden names are opaque, the shape of the predicate cannot be uniquely determine by a given arrow type. However it suffices that a closed program satisfies any predicate of the shape given below.

We next move to the second predicate, which is for divergence, written $\Phi_{\Uparrow}(x_1 : \alpha \stackrel{y;l_1}{\Rightarrow}_{s'} \beta, x_2 : \alpha \stackrel{y;l_2}{\Rightarrow}_{s'} \beta, s)$.

$$
\begin{array}{l}
\forall \vec{t}\vec{u}\vec{v}.\ (\ \mathtt{mnd}'(t_1u_1v_1, t_2u_2v_2, s)\ \wedge\ s' \sqsubseteq s \\
\qquad \supset\ (\{!y = u_1 \wedge !l_1 = t_1\} x_1 \bullet v_1 \Uparrow \\
\qquad\qquad \equiv\ \{!y = u_2 \wedge !l_2 = t_2\} x_2 \bullet v_2 \Uparrow)\ )
\end{array}
$$

which says the convergence behaviours should coincide as far as the information level of the convergence (s' in the above case) is visible from the given secrecy level (s in the above case).

Finally we present the translation of the typability in secrecy typing. The only essential change is the need to prefix each mnd-predicates with opening formulae. We introduce a convenient notation. Assuming $M_i \stackrel{\text{def}}{=} \mathtt{new\ var}\ \vec{y}_i := \vec{W}_i\ \mathtt{in}\ V_i\ (i = 1,2)$ (closed programs of the same type), we write $\mathtt{mnd}(M_1, M_2 : \alpha, s)$ for $\models M_1, M_2 :_{x_1,x_2} A$ where:

$$A \stackrel{\text{def}}{=} (\nu\,\vec{y}_1) x_1 \text{ in } (\nu\,\vec{y}_2) x_2 \text{ in } \mathtt{mnd}(!\vec{y}_1 x_1, !\vec{y}_2 x_2, s) \tag{5.4}$$

with $!\vec{x} \stackrel{\text{def}}{=} !x_1..!x_n$ when $\vec{x} = x_1..x_n$. We further extend this predicate to a pair of the configurations which are results of converged evaluations, $C_i \stackrel{\text{def}}{=} (\nu\,\vec{y}_i)(V_i, \sigma_i)\ (i = 1,2)$, in the obvious way, writing $\mathtt{mnd}(C_1, C_2 : \alpha, \Theta, s)$ where α is the type of the value part while Θ is the typing of the exposed reference names.[8] Now let $\Delta = \Gamma \upharpoonright \mathsf{dom}_f(\Gamma)$. First, the sequent $\Gamma \vdash_{\mathsf{sec}} M : \alpha$ is translated as follows.

> *For each s, if we have* $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ *then* $\mathtt{mnd}'(M\xi_1, M\xi_2 : \alpha, s)$.

Next, the sequent $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ with α being total, is translated as follows. Let $\Theta = \Gamma \upharpoonright \varphi$.

> *For each s, if we have* $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ *and* $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$*, then* $\langle M\xi_1, \sigma_1 \rangle \Downarrow C_1$ *and* $\langle M\xi_2, \sigma_2 \rangle \Downarrow C_2$*, then* $\mathtt{mnd}(C_1, C_2 : \alpha, \Theta, s)$.

If β is partial, we translate $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ into the conjunction of the above statement and the following one:

> *For each s, whenever* $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ *and* $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$*, if* $\mathsf{protect}(\beta) \sqsubseteq s$*, then* $\langle M\xi_1, \sigma_1 \rangle \Uparrow$ *iff* $\langle M\xi_2, \sigma_2 \rangle \Uparrow$.

Note the translation simply increments the one for global state, adding one opening formula at the initial state and another at the final state. We also note, in these cases, raising to $\mathsf{protect}(\alpha) \cap \mathsf{protectE}(\Gamma \upharpoonright \varphi)$ is not necessary because hidden states are already s-related.

We restate the basic lemma. Below, as before, we write $\alpha_1..\alpha_n \Rightarrow \beta$ for $\alpha_1 \Rightarrow (\alpha_2 \Rightarrow ..(\alpha_n \Rightarrow \beta)..)$. While we believe (6) comes from categorical studies such as [17, 54], it is also a consequence of the equality over the translation of concerned programs into the π-calculus (which essentially only relies on the replication theorems as found in [41]).

Lemma 74. *Assume Θ (resp. Γ) uses only reference types (resp. non-reference types) as its co-domain.*

(1) $\Gamma, x : \alpha, \Theta \models_{\mathsf{sec}} M : \beta, \varphi$ *iff* $\Gamma, \Theta \models_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \overset{\varphi}{\Rightarrow}_s \beta$*. Similarly,* $\Gamma, x : \alpha, \Theta \models_{\mathsf{sec}} M : \beta$ *iff* $\Gamma, \Theta \models_{\mathsf{sec}} \lambda x^{\alpha}.M : \alpha \Rightarrow \beta$.
(2) $\Theta \models_{\mathsf{sec}} M : \alpha$ *iff* $\Theta \models M :_u \mathtt{NI}(u : \alpha)$*. If* $\Theta \models_{\mathsf{sec}} M : \alpha, \varphi$*,* $\Theta \models_{\mathsf{sec}} \sigma :$ *and* $\langle M, \sigma \rangle \Downarrow C$*, then* $\Theta \models C :_u \mathtt{NI}(u : \alpha)$.
(3) If $\Gamma, \Theta \models_{\mathsf{sec}} M : \alpha$ *and* $\Gamma, \Theta \models_{\mathsf{sec}} \xi$ *then we have:* $\Theta \models M\xi :_u \mathtt{NI}(u : \alpha)$*. Similarly if* $\Gamma, \Theta \models_{\mathsf{sec}} M : \alpha, \varphi$*,* $\Gamma, \Theta \models_{\mathsf{sec}} \xi$*,* $\Theta \models_{\mathsf{sec}} \sigma$ *and* $\langle M\xi, \sigma \rangle \Downarrow C$*, then we have:* $\Theta \models C :_u \mathtt{NI}(u : \alpha)$.
(4) $\mathtt{NI}(u : \alpha)$ *is admissible at u for each* α.
(5) If $\Theta, \vec{y} : \vec{\beta}, x : \alpha \models_{\mathsf{sec}} V : \gamma$ *then we have:* $\Theta, \vec{y} : \vec{\beta}, z : \vec{\beta} \Rightarrow \alpha \models_{\mathsf{sec}} V[z \bullet \vec{y}/x] : \gamma$.
(6) $\models_{\mathsf{sec}} \mu z.\lambda \vec{y}.V[z \bullet \vec{y}/x] : \vec{\beta} \overset{\varphi}{\Rightarrow}_s \alpha \supset \models_{\mathsf{sec}} \lambda \vec{y}.\mu x.V : \vec{\beta} \overset{\varphi}{\Rightarrow}_s \alpha$.

Proposition 75. *1. Let Θ has only reference types as its codomain,* $\Theta \models_{\mathsf{sec}} M :_{1,2} : \alpha, \varphi$ *and* $\Theta' = \Theta \upharpoonright \varphi$*. Further assume* $\mathsf{protect}(\alpha) \sqcap \mathsf{protectE}(\Theta') \not\sqsubseteq s$*. Then* $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta', s)$ *and* $(M_i, \sigma_i) \Downarrow C_i$ $(i = 1, 2)$ *imply* $\mathtt{mnd}'(C_1, C_2 : \alpha, \Theta', s)$.

[8] Formally, given $(\nu \vec{y})(V, \sigma \cdot \sigma')$ where $\mathsf{dom}(\sigma) = \{\vec{y}\}$ and $\mathsf{dom}(\sigma') = \{\vec{w}\}$, we regard this term as $\mathtt{new\ var}\ \vec{y} := \sigma(\vec{y})\ \mathtt{in}\ (V, \sigma'(\vec{w}))$ where $(V, \sigma'(\vec{w}))$ is an n-ary pairing. This recovers the shape needed for defining the formula in (5.4).

2. *If we have* $\mathtt{mnd}(\mathcal{C}_1, \mathcal{C}_2 : \alpha, \Theta \cdot x : \mathsf{Ref}_s(\beta), s)$*, then we also have* $\mathtt{mnd}((\nu x)\mathcal{C}_1, (\nu x)\mathcal{C}_2 : \alpha/x, \Theta/x, s)$.

PROOF: For (1), because $\mathsf{protect}(\alpha) \sqcap \mathsf{protectE}(\Theta') \not\sqsubseteq s$ means each of $M_{1,2}$ never writes to a reference visible from s directly or indirectly (including hidden states), we know all references visible from s are never written. Hence, for $\mathcal{C}_1$ and $\mathcal{C}_2$ named as x_1 and x_2, we can open x_1 and x_2 enumerating all their hidden references and s-relate them (which are s-invisible because they are raised up to $\mathsf{protect}(\alpha)$), as well as the return values themselves (for he latter formally by induction on types). (2) is immediate from the form of the mnd-predicate. $\square$

5.7 Local State (2): Proof of Noninterference

Theorem 76. $\Gamma \vdash_{\mathsf{sec}} M : \alpha$ *implies* $\Gamma \models_{\mathsf{sec}} M : \alpha$*. Similarly* $\Gamma \vdash_{\mathsf{sec}} M : \alpha, \varphi$ *implies* $\Gamma \models_{\mathsf{sec}} M : \alpha, \varphi$.

PROOF: (outline) By rule induction. All cases are essentially the same as in the case of global state except "hidden secrecy levels" (recorded in arrow types) are taken into consideration. We show the reasoning for two typical cases.

(Case) Let $N \stackrel{\text{def}}{=} \mathtt{case}\ V\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i\in\{1,2\}}$ which is typed as β under Γ. Also let $\gamma = \alpha_1 +_{s'} \alpha_2$. For simplicity below we assume β is total (the partial case is dealt with by incorporating divergence in the now familiar way). First, if $s' \sqsubseteq s$:

$$\begin{array}{lr}
\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s) \ \wedge\ \mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s) & \\
\ \supset \mathtt{mnd}'(V\sigma_1, V\sigma_2 : \gamma, s) \qquad \text{(IH, Lem. 58 (3))} & \\
\ \supset \wedge_{i=1,2}\, V\sigma_i \cong \mathtt{in}_j(W_i) \ \wedge & \\
\qquad \mathtt{mnd}'(W_1, W_2 : \alpha_j, s) & (s' \sqsubseteq s) \\
\ \supset (\wedge_{i=1,2}\, \langle M_j\sigma_i[W_i/x_j], \sigma_i\rangle \Downarrow \mathcal{C}_i \ \wedge & \\
\qquad \mathtt{mnd}(\mathcal{C}_1, \mathcal{C}_2 : \beta, \Theta, s) & \text{(IH)}
\end{array}$$

with j being either 1 or 2. On the other hand, if $s' \not\sqsubseteq s$, we have (a) $\mathsf{protect}(\beta) \not\sqsubseteq s$ and (b) $\mathsf{protectE}(\Gamma \upharpoonright \varphi) \not\sqsubseteq s$, by the secrecy condition. From (a) and (b) we can use Proposition 75 (1), hence as required.

(New) Assume $\Gamma \models_{\mathsf{sec}} V : \alpha$ and $\Gamma, x : \mathsf{Ref}_s(\alpha) \models_{\mathsf{sec}} N :: \beta, \varphi$. For simplicity let α be total. We show $\Gamma \backslash x \models_{\mathsf{sec}} \mathtt{new\ var}\ x := V\ \mathtt{in}\ N : \beta\backslash x, \varphi\backslash x$. Let $\Delta = \Gamma \upharpoonright \mathsf{dom}_f(\Gamma)$ and $\Theta = \Gamma \upharpoonright \varphi$. Further let $\mathtt{mnd}'(\xi_1, \xi_2 : \Delta, s)$ and $\mathtt{mnd}'(\sigma_1, \sigma_2 : \Theta, s)$. Now set $M \stackrel{\text{def}}{=} \mathtt{new\ var}\ x := V\ \mathtt{in}\ N$. Then evaluating $(\nu x)(M\xi_i, \sigma_i \cdot x \mapsto V)$ $(i = 1,2)$ is the same thing as evaluating $(M\xi_i, \sigma_i \cdot x \mapsto V)$ $(i = 1,2)$. Since we have $\mathtt{mnd}'(V, V : \alpha, s)$ for each s by the first assumption, if $(M\xi_i, \sigma_i \cdot x \mapsto V) \Downarrow \mathcal{C}_i$ $(i = 1,2)$ then we have $\mathtt{mnd}(\mathcal{C}_1, \mathcal{C}_2 : \alpha, \Theta,)$. By Proposition 75 (2) we are done. $\square$ The secrecy typing with local state in [24] (restricted to sequential programs) obeys this discipline. The use of logical methods (in particular Proposition 56) also allows us to verify untypable programs are safe. For example, we can infer a program "$\mathtt{if}\ !y^{\mathrm{H}}\ \mathtt{then}\ y := 1^{\mathrm{H}}; x := 2^{\mathrm{L}}\ \mathtt{else}\ x := 2^{\mathrm{L}}$" is safe and satisfies $\models_{\mathsf{sec}}$, which is not generally typable in the systems of [24, 47, 62].

5.8 Extensions and Refinement

The representation of the secrecy property using logical predicates and associated non-interference proofs can be refined in several directions. First of all, as we noted in Remark 55, §5.4, we believe all proofs including those for local state can be carried out syntactically, using an alternative proof system mentioned in Remark 55. While this does not still preclude the use of validity in the proofs, this will lead to a further simplification of the non-interference proofs.

As further extensions and refinements, we may consider:

(A) Treatment of convergence.
(B) Extensions to second-order polymorphism and recursive types.
(C) Treatment of secrecy policies with explicitly secrecy levels [39, 47, 63] as well as other secrecy policies [3].

Of these, (A) indicates the choice to disregard convergence/divergence as information emittance. By the use of pointed types in our technical development, however, the presented framework has already incorporated this refinement: just by raising the level of pointed types to the (unreachably) high level, we can always disregard convergence.

For (B), we believe extensions to second-order polymorphism, including existential and universal types, can be soundly done. This is mainly because we are working with second-order polymorphisms at the level of syntax (a recent method by Vouillon and Mellies may be usable to give a behavioural representation of these types). For recursive types, we cannot use the inductive construction as we have developed so far. Can we logically represent a non-interference property for a program with recursive types finitely? This may be possible for a concrete datum, at least if it does not contain destructive update. Another method is introduction of recursion at the level of formulae, as in modal μ-calculi.

For (C), we believe the policy with local state as discussed in the previous section is in fact obeyed by the type disciplines in [39, 47, 63]. The dependency core calculus [3] has introduced a way to allow globally secure flow while violating local safety. This can be easily incorporated as a refinement of our predicates, by raising the levels of contravariant types as well as reference types up to the level of the protection level of the type of a datum to be produced (just as the local inflation of hidden state in the interpretation of arrow types in E.5 above).

6 Discussion

This paper presented a compositional program logic for higher-order procedures with local state, with an application to secrecy analysis. The logic is stratified on the basis of the one for pure polymorphic functions [25]. This stratification allows a clean and general logical articulation of programs combining higher-order functions and local state which, to our knowledge, has not been known before. In the following, we discuss related work and further topics.

Further discussions on related work are found in [23, 25].

Equational Logics Equational logics for the λ-calculi have been studied since the classical work by Curry and Church. LCF [13] 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 [4, 46] or using logical relations [2, 43, 44, 60]. 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, on the other.

Hoare Logics and Names A systematic use of names is the basis of the proposed assertional method. Early work by Kowaltowski [30] introduced naming into Hoare Logic in order to treat return values of expressions. In his logic, each assertion has access to a default name which denotes the result of the current expression. Von Oheimb [58] partly uses Kowaltowski's method for formulating a Hoare logic for a subset of Java. His proof system is complete with respect to partial correctness of programs. Other related work includes a logic for objects with subtyping by Leavens and Baker [32], which is a basis of JML [1], an integrated specification/verification framework for Java. program logic for a variant of the object calculus. Their work focusses on specific features of object-oriented computation. For example, field update is specified using a pair of predicates reminiscent of hooked and unhooked variables in VDM; an object creation (allocation) is specified using a similar predicates. These predicates treat some aspects of higher-order operations. They also use the idea of naming a return value. The present work aims to offer a uniform account of some of the related features including higher-order computation, polymorphism, side effects and local state, which may not be found in these precursors. An extension of the presented method to a logic for a simple object-oriented language is reported in [23, §12].

Assertions for Procedures and Shared Data Structures Dynamic Logic [16], introduced by Pratt [48] and studied by Harel [15] and others, 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. The nesting of predicates on programs in Dynamic Logic is analogous to the nesting of predicates on names which denote programs in the present logic, even though existing versions of dynamic logics do not (intend to) treat such features as higher-order procedures and local state. Assertions in the present logic talk about behaviours of programs not by having them in formulae but by naming them. Our approach may give a more abstract perspective on models of the logic (since the underlying universe of behaviours can be stipulated independently from a specific programming language), even though further studies are needed for comparing those two approaches.

Reynolds, O'Hearn, Bornat and their colleagues [7, 40, 51, 52] study extensions of Hoare's Logic, aiming to offer effective reasoning methods about low-level operations such as garbage collection. A central idea of their approach is to explicitly assume dynamically allocated regions of memory cells in the universe of discourse and to assert on them using a conjunction which at the same time assumes disjointness of two regions (called *separated conjunction*). Their focus on this specific form of conjunction partly comes from the observation that some forms of separated use of memory regions are often found in practical programs. The present work aims to offer an integrated basis for logically articulating central features of high-level programming constructs found in major programming languages, including standard (monomorphic and polymorphic) data types, higher-order procedures and local state, which differs from their work. For treating shared mutable data structures, the present logic takes a different approach from that by Reynolds, O'Hearn and others: we use classical conjunction, but tries to achieve clean structured reasoning using a predicate which indicates explicit sharing (we note our approach allows us to have an arguably more general form of separation embodied by, among others, the weakening rule in Figure 3). How these two approaches compare in practical reasoning is an interesting subject for further studies.

A logic for higher-order procedures and imperative variables, called a specification logic, is developed by Reynolds for Idealised Algol [49]. Apart from the shape of assertions and target languages, the use of compositional proof rules in the present logic is a major difference.

Semantics of Local State A body of work closely related with the treatment of local state in the present logic is the semantic studies of higher-order procedures with local variables, cf. [34, 45]. This has been a challenging topic because it defies the traditional approach to semantics of programs based on CPOs. In this context, the present work intends to offer one possible approach to this problem from a viewpoint of compositional program logics, offering new reasoning principles and analytical tool. As an example, take the following two programs treated by Meyer and Sieber in [34, Example 7], which are observationally equivalent but hard to equate in CPO-based models.

$$\mathtt{let}\ f = (\mathtt{new\ var}\ x := 0\ \mathtt{in}\ \lambda().x := !x+1)\ \mathtt{in}\ pf$$
$$\mathtt{let}\ f = (\mathtt{new\ var}\ x := 0\ \mathtt{in}\ \lambda().x := !x+2)\ \mathtt{in}\ pf.$$

In our logic, the proof rules in Figure 6 indicate that we can only have logically equivalent assertions for them regardless of properties of a program which stands for p, intuitively because an assertion for that program, say $A \vdash P :_p B$, cannot mention x, hence cannot use any property of f except its termination. The interplay between behaviours is given a structural understanding using proof rules.

The present logic uses the observation congruence in its semantics of assertions (cf. Section 4.4). This is essential for abstract reasoning, and conservatively extends our approaches in [22, 25]. Nevertheless it incurs a new level of complexity in calculation of validity. The study of useful calculation methods is an important topic for further studies.

Secure Information Flow in Logics Joshi and Leino [29] develop relational semantics based on weakest preconditions for secrecy analysis. Their proposal is general enough to reason about most language features semantically, but they lack an inference system. Darvas, Hähnle and Sands [10] use a version of Dynamic Logic to analyse secure information flow. Their target language is a first-order language without polymorphism and local variables. Benton [6] studies an equational logic for reasoning about various program properties. Among others he translates a form of Volpano-Smith's secure typing system for a first-order imperative language into relational predicates and shows its correctness using logical relations. More recently, Amtoft and Banerjee [5] presented a compositional static analysis for secrecy of first-order imperative programs which is based on a Hoare-style sequent. The sequent uses, instead of general assertions, a relational predicate which represents the lack of in(ter)dependency between a pair of variables. They have shown the soundness of the analysis with respect to an applied form of the standard relational model for Hoare logic. One of the merits of their approach is the generation of a counterexample when a flow safety fails, using what corresponds to Floyd's verification condition generation.

The main difference of the present work from these preceding studies is the use of a general assertion method with clear semantic status as a tool for logical analysis. The logical language which can discuss general properties of programs allows us to precisely pinpoint and write down the noninterference properties of interest, for pure higher-order functions, for functions with global state, and for those with local state. We have demonstrated the use of this descriptive power for justification of a secrecy typing. The experience encourages us to seek further possibilities in the use of compositional program logics as an integrated formal foundation for software engineering and architecture. As a concrete topic, it would be interesting to describe refined security policies such as degradation [39, 63] and those involving resource access control [27] using extensions of the present logic. Another challenging topic is an integrated security analysis with a logical basis applicable to large programs.

Among the proof methods for noninterference, ours is related to the use of logical relations in DCC [3] and an operational technique by Pottier and Simonet [47]. In comparison, ours is based on explicit and transparent description of noninterference, and may arguably be more modular (and, as a result, shorter as far as the part of the proof on noninterference goes). A shortcoming of our approach is that we need a good compositional logic for a target language to start with.

Extensions Although not discussed in this paper for space sake, the presented methods are extensible to diverse forms of typed behaviour, such as call-by-name evaluation

(§ 7 in [23]), a traditional higher-order procedural languages (§ 10.7 in [23]) and stateful objects (§ 12 in [23]). Different notions of correctness, including partial correctness (§ 7 in [23]) can also be expressed. We have also omitted treatment of references carried as parameters of procedures or as content of other references. While this can be incorporated by encoding into the present logic, a direct treatment needs another level of stratification [23, §11], which will be discussed elsewhere. From a more practical viewpoint, a significant challenge is to obtain tractable methods for calculating validity in the logic besides that of number-theoretic facts.

The origin of the assertions and judgements introduced in the present work is the logic for typed π-calculi [22, 23], where linear types lead to a compositional process logic. The known precise embeddings of high-level languages into these typed π-calculi can be used to determine the shape of name-based logics like those presented here for the embedded languages. Once found, they can be embedded back with precision into the originating process logics. [21, 23] discuss process logics and their relationship to the program logics in detail. [22] also summarises the background of this approach.

References

1. The java modeling language (jml) home page. http://www.jmlspecs.org/.
2. Martín Abadi. ⊤⊤-closed relations and admissibility. *MSCS*, 10(3), 2000.
3. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proc. 26th POPL*. ACM Press, 1999.
4. Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *TCS*, 121(1-2), 1993.
5. Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *Symposium on Static Analysis 2004*, LNCS. Springer Verlag, 2004.
6. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. POPL*, 2004.
7. Richard Bornat. Proving pointer programs in hoare logic. In *Proc. Conf. on Mathematics of Program Construction*, LNCS. Springer-Verlag, 2000.
8. Luís Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part I). *Inf. and Comp.*, 186(2):194–235, 2003.
9. Ole-Johan Dahl and Tony Hoare. Hierarchical program structure. *Structured Programming*, pages 175–220, 1972.
10. Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In *Proc. Workshop on Issues in the Theory of Security*, 2003.
11. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7), 1977.
12. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
13. Mike Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh LCF. LNCS 78, Springer Verlag, 78.
14. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
15. David Harel. Proving the correctness of regular deterministic programs. *TCS*, 12(16), 1980.
16. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logigc*. The MIT Press, 2000.
17. Masahito Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. Distinguished Dissertation Series. Springer-Verlag, 1999.
18. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
19. Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
20. Kohei Honda. A theory of types for the π-calculus. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2001.
21. Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, November 2003. Typescript, 50 pages.
22. Kohei Honda. From process logic to program logic. In *Proc. ICFP'04*, 2004. A short version of Sections 1–3 and 6 in [23].
23. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
24. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL'02*, pages 81–92. ACM Press, 2002. Full version available at www.doc.ic.ac.uk/˜yoshida, Revised in June 2004, 84 pages.
25. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*, 2004. To appear.
26. Brian T. Howard. Inductive, coinductive, and pointed types. In *Proc.* 1^{st} *int. conf. on Functional programming*. ACM Press, 1996.
27. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proc. 29th POPL*, 2002.

28. Alan Jeffrey and Julian Rathke. Towards a theory of bisimulation for local names. In *Proc. LICS'99*, pages 56–66. IEEE Computer Society Press, 1999.
29. Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3), 2000.
30. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
31. Butler W. Lampson. A note on the confinement problem. *Comm. ACM*, 16(10), 1973.
32. 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.
33. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
34. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.
35. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
36. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
37. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
38. Eugenio Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
39. Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proc. 26th POPL*, 1999.
40. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
41. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
42. Benjamin C. Pierce. *Type Systems and Programming Languages*. MIT Press, 2002.
43. Andy M. Pitts. Existential Types: Logical Relations and Operational Equivalence. In *Proc. ICALP'98*, number 1443 in LNCS, 1998.
44. Andy M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10, 2000.
45. Andy M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.
46. Gordin Plotkin and Martín Abadi. A logic for parametric polymorphism. In *Proc. LICS'98*, 1998.
47. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Prog. Lang. and Sys.*, 25(1):117–158, January 2003.
48. 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.
49. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
50. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
51. John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, 2000.
52. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
53. Andrei Sabelfield and David Sand. A per model of secure information flow in sequential programs. In *Proc. of ESOP'99*, number 1576 in LNCS. Springer, 1999.
54. Alex Simpson and Gordon Plotkin. Complete Axioms for Categorical Fixed-Ppoint Operators. In *Proc. 15th LiCS*, 2000.
55. Scott Smith. From Operational to Denotational Semantics. In *Proc. Conference on Mathematical Foundations of Programming Language Semantics*, volume 598 of *LNCS*, 1992.

56. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. LICS'92*, pages 162–173, 1992.
57. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2,3):167–187, 1996.
58. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.
59. J. Vouillon and P. Mellies. Semantic types: A fresh look at the ideal model for types. In *Proc. POPL'04*, pages 52–63, 2004.
60. Philip Wadler. Theorems for free. In *Proc. Functional Programming Languages and Computer Architecture*, 1989.
61. Nobuko Yoshida, Kohei Honda, and Martin Berger. Linearity and bisimulation. In *FoSSaCs02*, volume 2303 of *LNCS*, pages 417–433. Springer, 2002.
62. Steve Zdancewic and Andrew C. Myers. Secure information flow and cps. In *Proc. 10th ESOP*. Springer-Verlag, 2001.
63. Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

A Remaining Proofs for Theorem 34

In this appendix we list those proofs for Theorem 34 which are omitted from §4.8. The proofs for the rules in Figure 2 are identical with those in §3.6, hence omitted. We need a couple of lemmas.

Lemma 77. (substitution) *Assume* $x \notin \{\vec{y}\} \cup \mathsf{fv}(e) \cup \mathsf{dom}(\mathcal{M})$ *and* $\{\vec{y}\} \cap (\mathsf{fv}(e) \cup \mathsf{dom}(\mathcal{M})) = \emptyset$. *If* $\mathcal{M} = (\nu\vec{y})(\xi, \sigma)$ *then*

$$\mathcal{M} \models^I C[e/x] \quad \textit{iff} \quad (\nu\vec{y})(\xi \cdot x : [\![e]\!]\mathcal{M} \cdot I, \sigma) \models^I C.$$

Proof. Immediate.

Lemma 78. *Suppose* $C \setminus \nu x$ *is defined. Then*

1. $\mathcal{M} \models^I C \setminus \nu x$ *implies* $(\nu x)\mathcal{M} \models^I C$.
2. $\mathcal{M} \models^I C$ *implies there exists* $\mathcal{M}'$ *such that* $\mathcal{M}' \models C \setminus \nu x$ *and* $(\nu x)\mathcal{M}' \cong \mathcal{M}$.

PROOF: (1) Let $C \stackrel{\text{def}}{=} (\nu x\vec{y})\vec{z}$ in C'. Then by assumption, we have $\mathcal{M} \models^I (\nu\vec{y})\vec{z}$ in C', which implies $\mathcal{M}' \models^I C'$ with $\mathcal{M} \cong (\nu\vec{y})\mathcal{M}'$ and $\mathcal{M}' \stackrel{\text{def}}{=} \xi \cdot \vec{z} : \vec{W}\vec{y} \mapsto \vec{V} \cdot x \mapsto V' \cdot \sigma$ for some $\xi, \vec{W}, \vec{V}, V'$ and σ. This implies $(\nu\vec{y}x)\mathcal{M}' \models^T C$. Hence $(\nu x)\mathcal{M} \models^T C$, as required.

(2) If $\mathcal{M} \models^I (\nu x\vec{y})\vec{z}$ in C', then $\mathcal{M} \cong (\nu x\vec{y})\mathcal{M}'$ and $\mathcal{M}' \models^I C'$ such that $\mathcal{M}' \stackrel{\text{def}}{=} \xi \cdot \vec{z} : \vec{W}\vec{y} \mapsto \vec{V} \cdot x \mapsto V' \cdot \sigma$ for some $\xi, \vec{W}, \vec{V}, V'$ and σ. This implies $(\nu\vec{y})\mathcal{M}' \models^I (\nu\vec{y})\vec{z}$ in $C' \stackrel{\text{def}}{=} C \setminus \nu x$ where $(\nu x\vec{y})\mathcal{M}_1 \cong \mathcal{M}$, as required. □

First we treat [*Abs*]. Assume $\mathcal{M} \models^I C^{\text{-}x} \wedge A^{\text{-}x} \wedge A_1^x$ with $\mathcal{M} = (\nu\vec{y})(\xi, \sigma)$. Then we prove:

$$u : (\lambda x.M)\xi \models^I B$$

with $\forall x.\,(A_1 \supset \{C\}\, u \bullet x \searrow m\, \{C'\}) \supset B$. By (IH) and noting $x \notin \mathsf{fv}(A) \cup \mathsf{fv}(C)$, we know, under the assumption $\xi \models^I A \wedge x : V\xi \models^I A_1 \wedge \sigma \models^I C$, we have:

$$(\nu\vec{y})(M[V/x]\xi, \sigma) \Downarrow (\nu\vec{z})(W, \sigma') \text{ s.t. } (\nu\vec{z})(m : W, \sigma') \models^I C'$$

Note that $(\nu\vec{y})((\lambda x.M)V)\xi, \sigma) \Downarrow (\nu\vec{z})(W, \sigma')$. From this and by definition of the satisfaction, under the assumption $\xi \models^I A$, we have:

$$x : V\xi \models^I A_1 \supset \mathcal{M} \cdot x : V\xi \cdot u : \lambda x.M\xi \models^I \{C\}\, u \bullet x \searrow m\, \{C'\}$$

Then by definition of the satisfaction, we have:

$$\mathcal{M} \cdot u : \lambda x.M\xi \models^I \forall x.(A_1 \supset \{C\}\, u \bullet x \searrow m\, \{C'\})$$

Since u is the sole primary name, the thinning gives us:

$$u : \lambda x.M\xi \models^I \forall x.(A_1 \supset \{C\}\, u \bullet x \searrow m\, \{C'\})$$

Finally by $\forall x.(A_1 \supset \{C\}\, u \bullet x \searrow m\, \{C'\}) \supset B$, we have the required result.

Next we prove [*App*]. We assume $\Gamma \vdash V : \alpha \overset{\varphi}{\Rightarrow} \beta$ and $\Gamma \vdash W : \alpha$. Assume

$$\mathcal{M} = (\nu\vec{y})(\xi, \sigma_\varphi \cdot \sigma)$$

where $\mathsf{dom}(\sigma_\varphi) = \varphi$. Then if

$$(\nu\vec{y})((VW)\xi, \sigma_\varphi \cdot \sigma) \Downarrow (\nu\vec{z})(V', \sigma'_\varphi \cdot \sigma)$$

(where the substitution ξ makes VW appropriately semi-closed) then we have to show

$$(\nu\vec{y})(u : V', \sigma'_\varphi, \sigma) \models^I C'$$

By the (IH) and weakening, we have

$$(\nu\vec{y})(\xi \cdot v : V\xi, \sigma_\varphi \cdot \sigma) \models^I C_1$$
$$(\nu\vec{y})(\xi \cdot w : W\xi, \sigma_\varphi \cdot \sigma) \models^I C_2$$

so the definition of the satisfaction of $\wedge$ delivers

$$\mathcal{M}' \overset{\text{def}}{=} (\nu\vec{y})(\xi \cdot v : V\xi \cdot w : W\xi, \sigma_\varphi \cdot \sigma) \models^I C_1 \wedge C_2$$

By the other premise then

$$\mathcal{M}' \models^I \{C\}\, v \bullet w \searrow u\, \{C'\}$$

Clearly $[\![v]\!]\mathcal{M}' \cdot I = V\xi$ and $[\![w]\!]\mathcal{M}' \cdot I = W\xi$. Then by the definition of the satisfaction (noting $\mathcal{M} \setminus \varphi = (\nu\vec{y})(\xi, \sigma)$), we must have

$$(\nu\vec{z})(\xi, v : V\xi \cdot w : W\xi \cdot u : V', \sigma'_\varphi \cdot \sigma) \models^I C'$$

Finally by thinning, we have

$$(\nu\vec{y})(u : V', \sigma'_\varphi \cdot \sigma) \models^I C'$$

because neither v nor w can occur in C'.

[*If*] and [*Case*] are mechanical and similar to the proofs of (1).

Next case is [*Assign*]. We'll prove it for the following slightly more general form of the rule.

$$\frac{\{C\}\ V :_v \{A\} \quad (C[i/x] \wedge A[!x/v]) \supset C'[()/u] \quad i \text{ fresh}}{\{C\}\ x := V :_u \{C'\}}$$

With $\mathcal{M} = (\nu\vec{y})(\xi, \sigma \cdot x \mapsto W)$, assume $\mathcal{M} \models^I C$. Then we prove:

$$(\nu\vec{y})(\xi \cdot u : (), x \mapsto V\xi \cdot \sigma) \models^I C'$$

We derive:

$$\begin{aligned}
&\mathcal{M} \models^I C \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto W \cdot i \mapsto W) \models^I C && \text{(A.1)} \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot i \mapsto W) \models^I C[i/x] && \text{(A.2)} \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^I C[i/x] && \text{(A.3)}
\end{aligned}$$

Here (3.12) and (3.13) are weakening, which is applicable as i, is fresh in the former and x in the latter application; (3.14) is Lemma 77. By (IH) and $\mathcal{M} \models^I C$ we get

$$\begin{aligned}
&(\nu\vec{y})(\xi, \sigma \cdot v \mapsto V\xi \cdot x \mapsto W) \models^I A \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi) \models^I A[!x/v] && \text{(A.4)} \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^I A[!x/v] && \text{(A.5)} \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^I C[i/x] \wedge A && \text{(A.6)} \\
&\Rightarrow (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^I C'[()/u] && \text{(A.7)} \\
&\Rightarrow (\nu\vec{y})(\xi \cdot u : (), \sigma \cdot x \mapsto V\xi \cdot i \mapsto W) \models^I C' && \text{(A.8)} \\
&\Rightarrow (\nu\vec{y})(u : (), \sigma \cdot x \mapsto V\xi) \models^I C' && \text{(A.9)}
\end{aligned}$$

Here (3.15) is Lemma 77. (3.16) is weakening. (3.17) uses (3.14). (3.18) comes from the second premise $(C[i/x] \wedge A[!x/v]) \supset C'[()/u]$. Then (3.19) is by the substitution lemma. The last inference is an application of thinning, which is permissible as i does not occur in C'.

For [*Deref*] assume $\mathcal{M} = (\nu\vec{y})(\xi, \sigma \cdot x \mapsto V)$. and

$$(\nu\,\vec{y})(!x, \sigma \cdot x \mapsto V) \Downarrow (\nu\,\vec{y})(V, \sigma \cdot x \mapsto V)$$

Then we prove:

$$(\nu\vec{y})(u : V, \sigma \cdot x \mapsto V) \models^I C'$$

We observe:

$$
\begin{aligned}
&\mathcal{M} \models^I C \\
&\Rightarrow \mathcal{M} \models^I C'[!x/u] && \text{(A.10)}\\
&\Rightarrow (\nu\vec{y})(\xi\cdot u : [\![!x]\!]\mathcal{M}\cdot I, \sigma\cdot x\mapsto V) \models^I C' && \text{(A.11)}\\
&\Rightarrow (\nu\vec{y})(\xi\cdot u : V, \sigma\cdot x\mapsto V) \models^I C'
\end{aligned}
$$

Here (3.21) follows from the premise of the rule, (3.22) is a direct consequence of Lemma 24 because u does not appear in ξ, V and σ. The final equivalence comes from evaluating $[\![!x]\!]\mathcal{M}\cdot I = V$. Then thinning derives the required result.

[*Unpack*] is similar to [*T-Pack*], hence we omit.

[*Promte*] is obvious by (IH) and the definition of the satisfaction.

For [*Weak*] let $\mathcal{M} = (\nu\vec{y})(\xi, \sigma)$. Then $(\nu\vec{y})(M\xi, \sigma) \Downarrow (\nu\vec{z})(V, \sigma')$ together with the (IH) implies

$$(\nu\vec{z})(\xi\cdot m : V, \sigma') \models^I C'. \tag{A.12}$$

Under the assumption $\varphi \cap \varphi' = \emptyset$ and $\mathcal{M} \models^I C \wedge C_1$, We have to prove

$$(\nu\vec{z})(\xi\cdot m : V, \sigma') \models^I C' \wedge C_1$$

Define $\sigma_\varphi \stackrel{\text{def}}{=} \{\sigma(w) \mid w \in \varphi\}$. Then we have:

$$
\begin{aligned}
\mathcal{M} \models^I C \wedge C_1 &\Rightarrow \mathcal{M} \models^I C_1 && \text{(A.13)}\\
&\Rightarrow (\nu\vec{y}\cap\varphi')(\xi, \sigma|_{\varphi'}) \models^I C_1 && \text{(A.14)}\\
&\Rightarrow (\nu\vec{z}\cap\varphi')(\xi, \sigma'|_{\varphi'}) \models^I C_1 && \text{(A.15)}\\
&\Rightarrow (\nu\vec{z})(\xi, \sigma') \models^I C_1 && \text{(A.16)}\\
&\Rightarrow (\nu\vec{z})(\xi\cdot m : V, \sigma') \models^I C_1 && \text{(A.17)}
\end{aligned}
$$

where $\sigma|_{\varphi'} = \sigma'|_{\varphi'}$ and $\vec{y}\cap\varphi' = \vec{z}\cap\varphi'$. Hence we have (3.26). Then weakening gives (3.27). Now by the definition of $\wedge$, we derive the required result.

The rules [*Consequence*], [$Aux_\forall$] and [$Aux_\exists$] are immediate from the definition of the satisfaction relation and the laws of first-order logic.

B Satisfaction Relation without Opaque Types

This subsection outlines an alternative formulation of the models which use a less strict notion of typing. The typing in particular does not use opaque types.

Assertions and Models without Opaque Types From now on we use the syntax of types introduced in Section 2, i.e. those without opaque types.

The typing for assertions is just as in §4.3, except we ignore opaqueness. The typing for models is given accordingly.

$$\frac{\Gamma, x : \mathsf{Ref}(\alpha) \vdash \mathcal{M} \quad x \text{ is a subject in } \mathcal{M}}{\Gamma/x \vdash (\nu x)\mathcal{M}}$$

where $(\nu x)\mathcal{M}$ is the model which adds x to the hiding of $\mathcal{M}$ (note x is free in $\mathcal{M}$ by the condition x being a subject in $\mathcal{M}$).

We use the following conventions/notations.

- Given $\mathcal{M} = (\nu\vec{x})(\xi,\sigma)$, we write $\mathcal{M}(y) = V$ when $\xi(y) = V$ and $\mathsf{fv}(V) \cap \{\vec{x}\} = \emptyset$ (such y is called *open*).
- If $\vec{y}$ is open in $\mathcal{M}$, we write $\mathcal{M} \setminus \vec{y}$ for the result of taking off the $\vec{y}$-part from $\mathcal{M}$.
- $\mathcal{M}_1 \cdot \mathcal{M}_2$ is the component-wise union of $\mathcal{M}_1$ and $\mathcal{M}_2$, stating at the same time their subjects are disjoint.

We illustrate the last two clauses by examples. First, for $\mathcal{M} \setminus xy$, we have, assuming $\{\vec{z}\} \cap \mathsf{fv}(V,W) = \emptyset$:

$$(\nu\vec{z})(\xi \cdot x : V, \sigma \cdot y \mapsto W) \setminus xy = (\nu\vec{z})(\xi,\sigma)$$

For the union, we have:

$$(\nu\vec{z}_1)(\xi_1,\sigma_1) \cdot (\nu\vec{z}_2)(\xi_2,\sigma_2) = (\nu\vec{z}_1\vec{z}_2)(\xi_1 \cup \xi_2, \sigma_1 \cup \sigma_2)$$

where we set $\vec{z}_1 \cap \vec{z}_2 = \emptyset$ (note that, in forming the union, we always need to make the hidings disjoint by appropriate α-conversion).

Satisfaction We first define interpretation of terms. As before, we assume I stand for an open model combined with a finite map from type variables to closed types. In such cases, we write I_a for an interpretation of the auxiliary names (an open model) and I_t for that of the free type variables in I (so that we have $I_a \cup I_y = I$).

Let $\Gamma \vdash \mathcal{M}$. Further let I be such that $\Theta \vdash I_a$ with $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Theta) = \emptyset$. Then the *interpretation* $[\![e]\!]\mathcal{M}\cdot I$ is is given as:

$$\begin{aligned}
[\![x^{\alpha}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} V \quad \text{(either } \mathcal{M}(x) = V \text{ or } I(x) = V\text{)}\\
[\![!x]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} V \quad \text{(either } \mathcal{M}(x) = V \text{ or } I(x) = V\text{)}\\
[\![x^{\mathsf{Ref}(\alpha)}]\!]\mathcal{M}\cdot I &\stackrel{\text{def}}{=} x\\
[\![\mathbf{c}]\!]\mathcal{M}\cdot I &= \mathtt{c}\\
[\![\mathbf{op}(\vec{e})]\!]\mathcal{M}\cdot I &= \mathbf{op}([\![\vec{e}]\!]\mathcal{M}\cdot I)\\
[\![\langle e,e'\rangle]\!]\mathcal{M}\cdot I &= \langle [\![e]\!]\mathcal{M}\cdot I, [\![e']\!]\mathcal{M}\cdot I\rangle\\
[\![\pi_i(e)]\!]\mathcal{M}\cdot I &= \pi_i([\![e]\!]\mathcal{M}\cdot I)\\
[\![\mathsf{inj}^i_{\alpha+\beta}(e)]\!]\mathcal{M}\cdot I &= \mathtt{in}_i([\![e]\!]\mathcal{M}\cdot I)\\
[\![e \bullet [\beta]]\!]\mathcal{M}\cdot I &= ([\![e]\!]\mathcal{M}\cdot I)([\![\beta]\!]I_t)\\
[\![\langle\beta,e\rangle^{\exists \mathrm{X}.\alpha}]\!]\mathcal{M}\cdot I &= \mathtt{pack}\,\langle [\![\beta]\!]I_t, [\![e]\!]\mathcal{M}\cdot I\rangle\,\mathtt{as}\,\exists \mathrm{X}.\alpha
\end{aligned}$$

Some remarks are due:

- In the last two clauses, we evaluate non-reference types by mapping free type variables to closed types, using the type map part of I.
- In the first two clauses, we require x, if it is primary, to be open in the given model. If not, the interpretation is undefined. Through a more informative typing on formulae and on models (which distinguish open names and opaque names), we can prevent such undefined interpretations. We use the present construction partly for the sake of simplicity and partly because the resulting satisfaction relation does not differ substantially (see Remark 79 for further discussions).

Next, the satisfaction relation $\mathcal{M} \models^{I} A$ is generated following the standard (classical) first-order logic, cf. [25, § 3.3], except for evaluation and opening formulae and except we interpret equality as the observational congruence $\cong$. Below we list the full definition (some of the clauses appear in Section 5 in the main paper). Let $\Gamma \vdash \mathcal{M}$ and $\Gamma \vdash C$. Then $\mathcal{M}^{\Gamma} \models^{I} C^{\Gamma}$, or often simply $\mathcal{M} \models^{I} C$, is generated by the following clauses. Below connectives on the right-hand side are used semantically. In the fourth line, $[\![\alpha]\!]$ denotes the set of all semi-closed values of type α (which are simply constants if α is a base type).

$$\begin{array}{lll}
\mathcal{M} \models^{I} e_1 = e_2 & \text{if} & [\![e_1]\!]_{\mathcal{M}\cdot I} \cong [\![e_2]\!]_{\mathcal{M}\cdot I} \quad ([\![e_i]\!]_{\mathcal{M}\cdot I}\,\text{defined}) \\
\mathcal{M} \models^{I} C_1 \wedge C_2 & \text{if} & (\mathcal{M} \models^{I} C_1) \wedge (\mathcal{M} \models^{I} C_2) \\
\mathcal{M} \models^{I} \neg C & \text{if} & \neg\,(\mathcal{M} \models^{I} C) \\
\mathcal{M} \models^{I} \forall x^{\alpha}.C & \text{if} & \forall V \in [\![\alpha]\!].\ \mathcal{M} \models^{I\cdot x:V} C, \\
\mathcal{M} \models^{I} \forall \mathrm{X}.C & \text{if} & \forall \alpha\ \text{closed}.\ \mathcal{M} \models^{I\cdot x:\alpha} C \\
\mathcal{M} \models^{I} C & \text{if} & \exists \mathcal{M}'.\,(\mathcal{M}' \models^{I} C \wedge \mathcal{M}' \cong \mathcal{M}).
\end{array}$$

The remaining logical connectives are through de Morgan duality. For evaluation/opening formulae, we set their satisfaction as follows.

- Let $\Gamma \vdash \mathcal{M}$ and $\Gamma \vdash \{C\}e_1 \bullet e_2 \searrow x\{C'\}$. Further assume $\Gamma \vdash e_1 : \alpha \overset{\varphi}{\Rightarrow} \beta$. Then $\mathcal{M} \models^{I} \{C\}e_1 \bullet e_2 \searrow x\{C'\}$ if $[\![e_i]\!]_{\mathcal{M}\cdot I}$ is well-defined, whose value we write V_i $(i = 1,2)$; and, for each σ such that $\Gamma \vdash \sigma$, $\varphi \subseteq \mathsf{dom}(\sigma)$ and $(\mathcal{M} \setminus \varphi) \cdot \sigma \models^{I} C$, we have $(V_1V_2, \sigma) \Downarrow (\nu \vec{y})(V, \sigma')$ such that $(\mathcal{M} \setminus \varphi)\cdot(\nu \vec{y})(x\!:\!V, \sigma') \models^{I} C'$.
- $\mathcal{M} \models^{I} (\nu \vec{x})\vec{u}$ in A if $\mathcal{M}' \models^{I} A$ and $\mathcal{M} = (\nu \vec{x})\mathcal{M}'$ where $\vec{u}$ and $\vec{x}$ are open in $\mathcal{M}'$ and no name in $\vec{x}$ occurs in $\mathcal{M}'$ except in $\mathcal{M}'(\vec{u}\vec{x})$.

We can check such properties as: $\mathcal{M} \models^{I} \neg\neg C$ iff $\mathcal{M} \models^{I} C$; $\mathcal{M} \models^{I} C_1 \wedge C_2$ iff both $\mathcal{M} \models^{I} C_1$ and $\mathcal{M} \models^{I} C_2$; similarly for two kinds of universal quantifications; and standard laws of the logic with equality (cf. Remark 3 (2) below).

The satisfaction relation defined above has two closely related features which are worth discussing. First, the relation is defined by finding a model modulo $\cong$ which satisfies the given formula as follows.

$$\mathcal{M} \models^{I} A \quad \text{if} \quad \mathcal{M}' \models^{I} A \ \wedge\ \mathcal{M}' \cong \mathcal{M}$$

(the use of this rule is discussed in [23, §11]). Second, when interpreting expressions, it always demands they are defined. We illustrate these points using examples in the following remark.

Remark 79. (satisfaction)

1. (undefinedness) Let $W \stackrel{\text{def}}{=} \lambda().(y := !y+1; !y)$ (which increments y and return its new value) and let
$$\mathcal{M} \stackrel{\text{def}}{=} (\nu y)(u : W\ v : W,\ y \mapsto 0)$$
Then we have:
$$\mathcal{M} \models \neg\ (u^{\mathsf{Unit}\Rightarrow\mathsf{Nat}} = v^{\mathsf{Unit}\Rightarrow\mathsf{Nat}}).$$
Note the hidden shared state between u and v in $\mathcal{M}$ cannot be erased by $\cong$ since it is semantically crucial for the behaviours assigned to these two free names. Thus the inequality is reached *not* because we can compare u and v and find out that they are distinct behaviours: rather it is because we cannot interpret u and v in the given model. To understand the semantics of this satisfaction, we may rewrite the above judgement using the following equivalent assertion.
$$\mathcal{M} \models \neg\,(\ (\nu\varepsilon)uv \text{ in } u = v\ ).$$
The assertion "$(\nu\varepsilon)uv \text{ in } u = v$" says, from the satisfaction for opening formulae given above:
Neither u nor v has any local state, shared or non-shared; and they have precisely the same behaviours up to $\cong$.
But u and v in $\mathcal{M}$ do have local state (which cannot be eliminated through the use of $\cong$), so the formula cannot be satisfied, hence the negation. Thus "undefinedness up to $\cong$" has a very clear meaning: it says that we cannot get rid of a local state semantically,
2. (extensional equality) Slightly changing the above model, let
$$\mathcal{M}' \stackrel{\text{def}}{=} (\nu yz)(u : W,\ v : W[z/y],\ y \mapsto 0,\ z \mapsto 0)$$
one may wonder why we do not allow:
$$\mathcal{M}' \models u = v$$
which surely seems correct in the present semantics: $\mathcal{M}'$ in fact contains two distinct closed programs which have precisely the same behaviour. The central reason why this is not allowed in the present logic is because we demand the semantics of assertions to satisfy the following standard substitutivity law for the equality (cf. [33, §8]; $A(x,x)$ below denotes a formula with two holes).
$$(A(x,x) \wedge x = y) \supset A(x,y)$$
This law would not hold if we allowed the equation between u and v and define any non-trivial predicates on their behaviour — for example, invoking u twice is different from invoking u then invoking v. By allowing terms to denote only purely functional behaviours (in addition to constant values), we maintain the substitutivity of equals for equals in the standard sense.