Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control

T. Crolard[a,1], E. Polonowski[a,1]

[a]*LACL, Université Paris-Est, 61 avenue du Général de Gaulle, 94010 Créteil Cedex, France*

Abstract

We derive a Floyd-Hoare logic for non-local jumps and mutable higher-order procedural variables from a formulæ-as-types notion of control for classical logic. A key contribution of this work is the design of an imperative dependent type system for Hoare triples, which corresponds to classical logic, but where the famous *consequence rule* is admissible. Moreover, we prove that this system is complete for a reasonable notion of validity for Hoare judgments.

Key words: Floyd-Hoare logic, higher-order procedure, jump, callcc, continuation, monad.

1 Introduction

Floyd-Hoare logics for non-local jumps are notoriously difficult to obtain, especially in the presence of procedures and local mutable variables [86]. This should be contrasted to the fact that deriving a Floyd-Hoare logic for a simple imperative language with jumps is quite natural, as first noticed by Jensen in [46] and developed in [2]. This follows directly from a surprising remark: the standard continuation semantics can be seen as a generalization of Dijkstra's predicate transformers. Unfortunately, this idea does not seem to extend easily to procedures and local variables, which require to model somehow the stack, and thus complicate significantly the continuation semantics [80].

On the other hand, in his seminal series of papers [52, 55, 53], Landin proposed a direct translation (as opposed to a continuation-passing style translation) of an idealized Algol into the λ-calculus. This direct translation required to extend the λ-calculus with a new operator **J** in order to handle non-local jumps in Algol. The **J** operator, which was described in detail in [54] (see also [87] for an introduction), was the first control operator in functional languages (such as the famous **call/cc** of Scheme [48] or Standard ML of New Jersey [37]). Those operators have been subsequently explored thoroughly for instance by Reynolds [79], Felleisen [29] and Danvy [25, 26].

A type system for control operators which extends the so-called Curry-Howard correspondence [23, 45] to classical logic first appeared in Griffin's pioneering work [36], and was immediately generalized to first-order dependent types (and Peano's arithmetic) by Murthy in his thesis [66]. The following years, this extension of the formulas-as-types paradigm to classical logic has then been studied by several researchers, for instance in [3, 77, 27, 50, 71] and many others since.

It is thus tempting to revisit Landin's work in the light of the computational interpretation of classical logic. Indeed, although it is difficult to define a sound program logic for an imperative language with procedures and non-local jumps [69], adding first-order dependent types to such an imperative language, and translating type derivations into proof derivations appears more tractable. The difficult to obtain program logic is then mechanically derived. Moreover, this logic permits by construction to deal elegantly with *mutable* higher-order procedural variables (whereas in [86] procedural variables are immutable).

In [19], Chapter 3 (also available as [21]), we followed this path and the resulting framework could be qualified as a *classical imperative type theory*. This framework may also be seen as an attempt to bridge the gap between conventional program logics for imperative languages and type theories. As expected, we thus obtained a program logic for mutable higher-order procedural variables and non-local jumps (which is, as far as we know, the first such program logic). However, we are more interested in the other side of the bridge: in this framework, the programmer can provide an imperative program as the computational content of a classical proof and rely on the program logic to prove that this program realizes its specification. More importantly, a

[1] *Email addresses:* `crolard@u-pec.fr` (T. Crolard), `polonowski@u-pec.fr` (E. Polonowski)

complete proof-term of the specification can be built by combining the program with the proof of its correcteness. This framework is thus particularly appealing for studying the computational content of classical proofs from a programming perspective.

Let us be more specific about this classical imperative type theory. First, the imperative language (called $\textsc{Loop}^\omega$), which was originally defined by the authors in [22], is essentially an extension of Meyer and Ritchie's $\textsc{Loop}$ language [61] with higher-order procedural variables. $\textsc{Loop}^\omega$ is a genuine imperative language as opposed to functional languages with imperative features. However, $\textsc{Loop}^\omega$ is a "pure" imperative language: side-effects and aliasing are forbidden. These restrictions enable simple location-free operational semantics [28]. Moreover, the type system relies on the distinction between mutable and read-only variables to prevent procedure bodies to refer to non-local mutable variables. This property is crucial to guarantee that fix-points cannot be encoded using procedural variables. Since there is no recursivity and no unbounded loop construct in $\textsc{Loop}^\omega$, one can prove that all $\textsc{Loop}^\omega$ programs terminate. Although $\textsc{Loop}^\omega$ is somehow restricted as a programming language, these restrictions are very similar to what is actually implemented in the industry (such a toolset for verifying formal specifications of critical real-time systems is described in [4]).

By construction, $\textsc{Loop}^\omega$ is also an imperative counterpart of Gödel System T [34] (the expressive power of system T is attained thanks to mutable higher-order procedural variables). In other words, $\textsc{Loop}^\omega$ programs are imperative proof-terms of Heyting arithmetic. Programs with non-local jumps provide thus imperative proof-terms of Peano arithmetic (that is, an imperative counterpart to the extension of System T with control operators as described in [66]). Note that we shall use instead a formulation of Heyting arithmetic which was proposed by Leivant [56, 57] (and rediscovered independently by Krivine and Parigot in the second-order framework [51]). The main advantage of this variant is that it requires no encoding in formulas (with Gödel numbers) to reason about functional programs. Moreover it can be extended to any other algebraic data-types (such as lists or trees).

The original design of the *classical imperative type theory* described in [21] may be outlined as follows:

- We first defined an imperative dependent type system **ID** for $\textsc{Loop}^\omega$, by translation into a functional dependent type system **FD** (which is actually Leivant's formulation of Heyting arithmetic $\mathbf{IT}(\mathbb{N})$ [57]).
- We then extended $\textsc{Loop}^\omega$ with non-local jumps and raised its type system to $\mathbf{ID}^c$. The semantics of $\mathbf{ID}^c$ was given by translation into $\mathbf{FD}^c$ (which corresponds to Peano arithmetic), and $\mathbf{FD}^c$ is itself defined by $\neg\neg$-translation into **FD**.

Although the resulting classical imperative type theory does provide a program logic and has been successfully applied to verify non-trivial examples (including an encoding of delimited control operators **shift**/**reset** [20]), it is not as convenient as a Floyd-Hoare logic. The main reason is that Floyd-Hoare logics usually contain rules such as the famous *rule of consequence* which allows you to strengthen a pre-condition or weaken a post-condition. Such a rule is especially useful in practice since it enables the generation of proof-obligations (also called *verification conditions* [35]). The sub-task consisting in checking the validity of mathematical facts can thus be isolated from the task of program proving, whereas both activities are usually interwoven in type theory.

In this paper, we propose to consider a similar rule for the classical imperative type theory. The main difficulty comes from the fact that, in type theories, formulas and types are unified and the programmer has thus to provide proof-terms also for logical assertions. In a constructive type theory, there is a simple solution: some formulas have no computational content (we shall call them *irrelevant* as in [7], they are sometimes called *data-mute* [57] or *self-realizing* [89]). Thus, if we assume that assertions are irrelevant formulas then we can derive a *rule of consequence* for assertions. Besides, since any formula is classically equivalent to some irrelevant formula, this assumption is actually not a restriction (on the proviso that specifications are understood classically).

Unfortunately, in a classical type theory there is no obvious notion of irrelevant formula (this question is studied in details in [7, 8] and also in [59, 60]) and the above trick is no longer available. To address this issue, we reconsider here the way our *classical imperative type theory* was designed: instead of translating $\mathbf{ID}^c$ into $\mathbf{FD}^c$ (where we cannot erase proof-terms any more), we translate $\mathbf{ID}^c$ directly into **FD** while ensuring that assertions are not $\neg\neg$-translated. Proofs of assertions can thus still be erased, while retaining the full expressive power of classical logic. At the term level, it means that commands, sequences of commands and procedures are CPS-translated into non-erasable functional terms, while functional proof-terms are not translated (and can thus be erased).

In this revisited classical imperative type theory, the consequence rule is admissible. To be more specific, let us show informally what a rule of $\mathbf{ID}^c$ looks like. The syntax of imperative types, including higher-order procedure types and first-class labels, is the following (where φ ranges over functional dependent types):

$$\sigma, \tau ::= \varphi \mid \mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\tau}; \exists\vec{\jmath}\,\mathbf{out}\ \vec{\sigma}) \mid \mathbf{label}\ \exists\vec{\jmath}.\vec{\sigma}$$

Typing judgments of $\mathbf{ID}^c$ have the form $\Gamma; \Omega \vdash e\colon \sigma$ if e is an expression and $\Gamma; \Omega \vdash s \rhd \Omega'$ if s is a sequence, where environments Γ and Ω corresponds respectively to immutable and mutable variables. Note that our type system is *pseudo-dynamic* in the sense that the type of mutable variables can change in a sequence and the new (possibly existentially quantified) types are given by Ω' (as in [90]). Indeed, for instance, after the assignment $x := 0$, the type of x is $\mathbf{nat}(0)$. The type of x is thus changed by this assignment whenever the former value of x is different from 0. Moreover, the type of x before the assignment does not matter: there is no need to even require that x be a natural number. As an example, here is the typing rule of the assignment:

$$\frac{\Gamma; \Omega, y\colon \sigma \vdash e\colon \tau}{\Gamma; \Omega, y\colon \sigma \vdash y := e \rhd \Omega, y\colon \tau}$$

Let us now sketch how to simulate Hoare judgments in $\mathbf{ID}^c$. Indeed, let us take a global mutable variable, called *assert*, and let us assume that this global variable is simulated in the usual *state-passing style* (the variable is passed as an explicit **in** and **out** parameter to each procedure call). Consequently, any sequence shall be typed with a judgment of the form $\Gamma; \Omega, \mathit{assert}\colon \varphi \vdash s \rhd \Omega', \mathit{assert}\colon \psi$. If we now introduce the usual Hoare notation for triples (which hides the name of global variable *assert*), we obtain judgments of the form $\Gamma; \Omega\{\varphi\} \vdash s \rhd \Omega'\{\psi\}$. Rules very similar to Hoare rules are thus obtained: for instance, the type of *assert* corresponds to the invariant in a loop, and to the type of *pre* and *post* conditions in a procedure type. However, one rule which is not directly obtained that way is the *consequence rule*:

$$\frac{\Gamma, \Omega \vdash \varphi' \Rightarrow \varphi \qquad \Gamma; \Omega\{\varphi\} \vdash s \rhd \Omega'\{\psi\} \qquad \Gamma, \Omega \vdash \psi \Rightarrow \psi'}{\Gamma; \Omega\{\varphi'\} \vdash s \rhd \Omega'\{\psi'\}}$$

We shall prove nevertheless that this rule is admissible, even if s contains non-local jumps (in fact, a stronger consequence rule from VDM [47] is admissible). Let us now call $\mathbf{HD}^c$ the whole deduction system for Hoare judgments (including the strong consequence rule). We shall prove that $\mathbf{HD}^c$ is also complete in the following sense: any command typable in $\mathbf{ID}^c$ can be transformed into a command with the same computational content and the same specification (up to minor syntactic transformations) whose correctness can be derived in $\mathbf{HD}^c$.

To summarize, the main contribution of this work is two-fold:

- We define a classical imperative type theory in which programs represent proof-terms of Peano's arithmetic. This programming language features higher-order procedural variables and non-local jumps.
- We show that a Floyd-Hoare logic, where judgments are akin to Hoare triples, can be embedded in this type theory: the expected rules, including the consequence rule, are admissible. Moreover, we prove that this logic is complete with respect to the underlying type theory.

Related work

Let us first emphasize that our focus is on the computational content of classical proofs. Although our framework may look similar to other systems for proving the correctness of imperative programs, some programming language features (such as aliasing) cannot be integrated in our framework simply because no formulas-as-types interpretation is currently known for these features.

Program extraction from proofs in classical logic The computational content of proofs in Peano's arithmetic was first studied by Murthy in his thesis [66]. His work is based on the so-called A-translation which results from the composition of a double negation translation and Friedman's "trick" [32] (which replaces $\bot$ by some formula A). Berger and Schwichtenberg noticed in [9] that this translation introduces many unnecessary negations. They thus refined the A-translation in order to distinguish the computationally irrelevant occurrences of negation (those for which no term should be associated in the extraction process). They showed that applying this refined translation on practical examples can simplify significantly the resulting programs.

Although our goal is very similar, our technique is quite different. Indeed, we do not try to determine which part of a formula is irrelevant by examining its shape: the programmer provides this information directly by combining procedure and function types (since a procedure type contain implicitly a double-negation). Our approach is thus closer to the logic defined by Thielecke in [88] where the double-negation is abstracted as a modality: this logic is classical if you ignore the modality but it is intuitionistic if the modality is interpreted as a double-negation. We refer the reader to [88] for a development of this idea in the propositional setting (and thus unencumbered by dependent types).

Imperative dependent type systems A dependent type system for an imperative programming language (without jumps) is defined in [90], where the dependent types are restricted to ensure that type checking remains decidable. The authors of [90] also made the observation that imperative dependent types requires that the types of variables be allowed to change during evaluation.

Proofs-as-Imperative-Programs [75, 76] adapts the proofs-as-programs paradigm for the synthesis of imperative SML programs with side-effect free return values. The type theory is however intrinsically constructive: it requires a strong existential quantifier which is not compatible with classical logic [40]. Similarly, an encoding of VDM inside a variant of Martin-Löf's type theory is studied in [58] (see also [39] for a discussion about irrelevant formulas in this framework).

The Imperative Hoare Logic [44, 43] is another framework for reasoning about effectful higher-order functions which also incorporates Hoare-style specifications into types. Similarly, in this type system, Hoare triples may explicitly mention references and aliasing. Notice that this work has been extended to control operators in [6]. However, assertions in the resulting calculus may refer to "ports" which is very unusual for a sequential language (they are actually inherited from a previous version of the type system developed for the π-calculus).

The Hoare Type Theory (HTT) [67] combines dependent types and a Hoare-style logic for a programming language with higher-order functions and imperative commands. In particular, in order to deal with aliasing, HTT types may contain Hoare triples in which assertions explicitly refer to heaps and locations. Exceptions are present in Ynot [68], the implementation of HTT in the Coq proof assistant, but control effects like full-fledged continuations are only mentioned as future work in [67].

Program logics Although several program logics have been designed for higher-order procedural mutable variables or non-local jumps, we are not aware of any work which combines both in an imperative setting.

Of course, there has been much research on Floyd-Hoare logics [31, 41, 42] (see the surveys [1] and [17]). Such program logics for higher-order procedures have been defined for instance in [24] (for Clarke's language L4 [11]) or more recently for stored parameterless procedures in [78]. Program logics for jumps exists since [13] and they have been successfully used recently for proving properties in low-level languages [30, 85].

One program logic which does combine higher-order procedures, mutable variables and non-local jump is Reynolds specification logic [86]. However, mutable variables have only ground types by design in this framework.

Plan of the paper

In Section 2, we recall the functional language **F**, its dependent type system, the notion of computational content of a proof and the continuation monad. In Section 3, we recall the imperative language Loop^ω with labels and jumps and its dependent type system $\mathbf{ID}^c$. We present a new direct translation from $\mathbf{ID}^c$ into **FD** and prove that it preserves dependent types. In section 4, we construct the Hoare Dependent Type System, we prove its soundness and we show that the consequence rule is admissible and we prove the completeness theorem. Finally, we present a classical example of an imperative program with jumps which can be proved correct.

2 Functional Type Theory

2.1 Language F

Gödel System T may be defined as the simply typed λ-calculus extended with a type of natural numbers and with primitive recursion at all types [33]. The language **F** we consider in this paper a call-by-value variant of System T with product types (n-ary tuples actually) and a constant-time predecessor operation (since any definition of this function as a term of System T is at least linear under the call-by-value evaluation strategy [14]). Moreover, we formulate this system directly as a context semantics (a set of reduction rules together with an inductive definition of evaluation contexts). As usual, we consider terms up to α-conversion. The rewriting system is summarized in Figure 2.1, where variables $x, x_1, ..., x_n, y$ range over a set of identifiers and $t[v_1/x_1, ..., v_n/x_n]$ denotes the usual capture-avoiding substitution.

Notation 2.1. *We introduce the following abbreviations:*

- *we write $\bar{q}$ for $S^q(0)$*
- *we write* **succ** *for $\lambda x.S(x)$*
- *we write $\vec{x}$ for $x_1, ..., x_n$ and $\vec{u}$ for $u_1, ..., u_n$*
- *we write $\lambda(x_1, ..., x_n).t$ (or $\lambda\vec{x}.t$) for $\lambda z.$*$\textbf{let}$ *$(x_1, ..., x_n) = z$* **in** *t (where z is fresh).*

(terms)

$$\begin{array}{rcl} t & ::= & x \\ & \mid & 0 \\ & \mid & S(t) \\ & \mid & \mathbf{pred}(t) \\ & \mid & t_1\ t_2 \\ & \mid & \lambda x.t \\ & \mid & (t_1, ..., t_n) \\ & \mid & \mathbf{let}\ (x_1, ..., x_n) = t_1\ \mathbf{in}\ t_2 \\ & \mid & \mathbf{rec}(t_1, t_2, t_3) \end{array}$$

(values)

$$\begin{array}{rcl} v & ::= & x \\ & \mid & 0 \\ & \mid & S(v) \\ & \mid & (v_1, ..., v_n) \\ & \mid & \lambda x.t \end{array}$$

(contexts)

$$\begin{array}{rcl} C[\,] & ::= & [\,] \\ & \mid & C[\,]\ t \\ & \mid & v\ C[\,] \\ & \mid & S(C[\,]) \\ & \mid & \mathbf{pred}(C[\,]) \\ & \mid & \mathbf{rec}(C[\,], t_2, t_3) \\ & \mid & \mathbf{rec}(v_1, C[\,], t_3) \\ & \mid & \mathbf{rec}(v_1, v_2, C[\,]) \\ & \mid & (v_1, ...v_{i-1}, C[\,], t_{i+1}..., t_n) \\ & \mid & \mathbf{let}\ (x_1, ..., x_n) = C[\,]\ \mathbf{in}\ t \end{array}$$

(evaluation rules)

$$\begin{array}{rcl} C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\ C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\ C[\mathbf{rec}(0, v_2, \lambda x.\lambda y.t)] & \rightsquigarrow & C[v_2] \\ C[\mathbf{rec}(S(v_1), v_2, \lambda x.\lambda y.t)] & \rightsquigarrow & C[(\lambda x.\lambda y.t)\ v_1\ \mathbf{rec}(v_1, v_2, \lambda x.\lambda y.t)] \\ C[(\lambda x.t)\ v] & \rightsquigarrow & C[t[v/x]] \\ C[\mathbf{let}\ (x_1, ..., x_n) = (v_1, ..., v_n)\ \mathbf{in}\ t] & \rightsquigarrow & C[t[v_1/x_1, ..., v_n/x_n]] \end{array}$$

Figure 2.1. Syntax and context semantics of Language **F**

2.2 Functional simple type system FS

The functional simple type system **FS** is defined as usual for a simply typed λ-calculus extended with tuples, natural numbers and with primitive recursion at all types. Simple functional types are defined by the following grammar:

$$\alpha \ ::= \ \mathbf{nat} \ \mid \ \mathbf{unit} \ \mid \ \alpha_1 \rightarrow \alpha_2 \ \mid \ \alpha_1 \times ... \times \alpha_n$$

As usual, a typing judgment has the form $\Sigma \vdash t\colon \alpha$ where t is a term, α is a type and Σ is a a list of pairs $x\colon \alpha$ (x ranges over variables and α over types). The type system is summarized in Appendix A.

2.3 Functional dependent type system FD

Following the definition of $\mathbf{IT}(\mathbb{N})$ [57], we enrich language **F** with dependent types. The type system is parameterized by a first-order signature and an equational system $\mathcal{E}$ which defines a set of functions in the style of Herbrand-Gödel. We consider only the sort **nat** (with constructors 0 and **s**), and we assume that $\mathcal{E}$ contains at least the usual defining equations for addition, multiplication and a predecessor function **p** (which is essential to derive all axioms of Peano's arithmetic [57]). The syntax of formulas is the following (where n, m are first-order terms):

$$\varphi \ ::= \ \mathbf{nat}(n) \ \mid \ (n = m) \ \mid \ \forall \vec{\imath}\, (\varphi_1 \Rightarrow \varphi_2) \ \mid \ \exists \vec{\imath}\, (\varphi_1 \wedge ... \wedge \varphi_k)$$

Note that first-order quantifiers are provided in the form of dependent products and dependent sums. As usual, implication and conjunction are recovered as special non-dependent cases (when $\vec{\imath}$ is empty). Similarly, relativized quantification $\forall x(\mathbf{nat}(x) \Rightarrow \varphi)$ and $\exists x(\mathbf{nat}(x) \wedge \varphi)$ are also obtained as special cases.

The functional dependent type system is summarized in Figure 2.2 (where $\vdash_{\mathcal{E}} n = m$ means that either $n = m$ or $m = n$ is an instance of $\mathcal{E}$).

Remark 2.2. Note that **FD** is formulated here as an *extensional* type theory since, in rule (SUBST), a derived propositional equality $\Sigma \vdash v\colon (n = m)$ is used as hypothesis (and not only a definitional equality $\vdash_{\mathcal{E}} n = m$). As a consequence, type checking is undecidable for this version of **FD** (since v does not occur in the conclusion). An intensional version of **FD** (with full proof-terms), which has been implemented in the Twelf proof assistant [73], is described in [20].

(IDENT) $$\frac{x\colon \varphi\in\Sigma}{\Sigma\vdash x\colon \varphi}$$

(ZERO) $$\Sigma\vdash 0\colon \mathbf{nat}(0)$$

(SUCC) $$\frac{\Sigma\vdash t\colon \mathbf{nat}(n)}{\Sigma\vdash S(t)\colon \mathbf{nat}(\mathbf{s}(n))}$$

(PRED) $$\frac{\Sigma\vdash t\colon \mathbf{nat}(n)}{\Sigma\vdash \mathbf{pred}(t) : \mathbf{nat}(\mathbf{p}(n))}$$

(TUPLE) $$\frac{\Sigma\vdash t_1\colon \varphi_1[\vec{m}/\vec{\imath}\,] \quad ... \quad \Sigma\vdash t_k\colon \varphi_k[\vec{m}/\vec{\imath}\,]}{\Sigma\vdash (t_1,...,t_k)\colon \exists\vec{\imath}\,(\varphi_1\wedge ...\wedge\varphi_k)}$$

(LET) $$\frac{\Sigma, x_1\colon \varphi_1,...,x_k\colon \varphi_k\vdash t\colon \psi \qquad \Sigma\vdash u\colon \exists\vec{\imath}\,(\varphi_1\wedge ...\wedge\varphi_k)}{\Sigma\vdash \mathbf{let}\ (x_1,...,x_k)=u\ \mathbf{in}\ t\colon \psi} \qquad \vec{\imath}\notin\mathcal{FV}(\Sigma,\psi)$$

(ABS) $$\frac{\Sigma, x\colon \varphi\vdash t\colon \psi}{\Sigma\vdash \lambda x.t\ \colon \forall\vec{\imath}\,(\varphi\Rightarrow\psi)} \qquad \vec{\imath}\notin\mathcal{FV}(\Sigma)$$

(APP) $$\frac{\Sigma\vdash t_1\colon \forall\vec{\imath}\,(\varphi\Rightarrow\psi) \quad \Sigma\vdash t_2\colon \varphi[\vec{n}/\vec{\imath}\,]}{\Sigma\vdash t_1\ t_2 : \psi[\vec{n}/\vec{\imath}\,]}$$

(REC) $$\frac{\Sigma\vdash t_1\colon \mathbf{nat}(n) \quad \Sigma\vdash t_2\colon \varphi[0/i] \quad \Sigma, x\colon \mathbf{nat}(i), y\colon \varphi\vdash t_3\colon \varphi[\mathbf{s}(i)/i]}{\Sigma\vdash \mathbf{rec}(t_1,t_2,\lambda x.\lambda y.t_3) : \varphi[n/i]} \qquad i\notin\mathcal{FV}(\Sigma)$$

(EQUAL) $$\frac{\vdash_{\mathcal{E}} n=m}{\Sigma\vdash ()\colon (n=m)}$$

(SUBST) $$\frac{\Sigma\vdash t\colon \varphi[n/i] \quad \Sigma\vdash v\colon (n=m)}{\Sigma\vdash t\colon \varphi[m/i]}$$

Figure 2.2. Functional dependent type system **FD**

Remark 2.3. The conditional is definable from the recursor. First, let us describe the expected typing rule for the conditional (where $n\neq 0$ is an abbreviation for $\exists i.n=\mathbf{s}(i)$):

$$\frac{\Sigma\vdash t_1\colon \mathbf{nat}(n) \qquad \Sigma\vdash t_2\colon (n\neq 0)\Rightarrow\psi \qquad \Sigma\vdash t_3\colon (n=0)\Rightarrow\psi}{\Sigma\vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \psi}$$

The idea is that a natural number n is either 0 or $s(i)$ for some i. Thus, if in both cases we are able to prove a formula ψ, then we have proved ψ for any n. Now, we would define the conditional as the following abbreviation (where x,y and h are not free in t_2,t_3):

$$\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \quad\equiv\quad \mathbf{rec}(t_1,\lambda h.(t_3\ h),\lambda x\lambda y\lambda h.(t_2\ h))\ ()$$

Note that t_3 is η-expanded in order to simulate its lazy evaluation (since we assumed a call-by-value strategy for System T) and t_2 is also η-expanded since we have to deal with the existential quantifier needed to express that $n\neq 0$. The above typing rule is indeed derivable:

$$\dfrac{\dfrac{\Sigma\vdash t_1\colon \mathbf{nat}(n) \quad \Sigma\vdash t_3\colon (n=0)\Rightarrow\psi \quad \dfrac{\dfrac{\Sigma\vdash t_2\colon (n\neq 0)\Rightarrow\psi \quad \dfrac{\dfrac{}{\Sigma, h\colon n=\mathbf{s}(i)\vdash h\colon n=\mathbf{s}(i)}}{\Sigma, h\colon n=\mathbf{s}(i)\vdash h\colon \exists i.n=\mathbf{s}(i)}}{\Sigma, h\colon n=\mathbf{s}(i)\vdash (t_2\ h)\colon \psi}}{\Sigma\vdash \lambda h.(t_2\ h)\colon (n\neq 0)\Rightarrow\psi}}{\Sigma\vdash \mathbf{rec}(t_1,\lambda h.(t_3\ h),\lambda x\lambda y\lambda h.(t_2\ h))\colon (n=n)\Rightarrow\psi} \quad \dfrac{}{\Sigma\vdash ()\colon (n=n)}}{\Sigma\vdash \mathbf{rec}(t_1,\lambda h.(t_3\ h),\lambda x\lambda y\lambda h.(t_2\ h))\ ()\colon \psi}$$

Remark 2.4. The above encoding assumes that t_1 evaluates to either 0 or 1. Indeed, as noticed in [14], evaluation in call-by-value System T suffers from the *ultimate obstinacy* property. Informally, this property captures the fact that a recursion must always unfold all the way to the end. As a consequence, it is not possible to encode a zero-test (or a conditional) which evaluates in constant time. Of course, such a conditional could be taken as primitive, with the corresponding reduction rules (as usual in call-by-value functional languages).

Alternatively, this defect can be fixed by taking an alternate reduction rule for the recursor. For instance, the *ultimate obstinacy* property does not hold any longer if we consider the following reduction rule for **rec** (which is derivable in call-by-name, but not in call-by-value):

$$\mathbf{rec}(S(n), v, \lambda x.\lambda y.t) \ \rightsquigarrow \ t[n/x, \mathbf{rec}(n, b, \lambda x.\lambda y.t)/y]$$

Note that we originally chose the regular call-by-value System T in [22] precisely to derive such complexity results for Loop^ω. Since this article is focused on proving program properties (and we do not consider complexity issues), we preferred to keep relying on the sub-optimal operational semantics developed in [22].

2.4 Computational content

The only minor difference between our dependent type system and the deduction system $\mathbf{IT}(\mathbb{N})$ described in [57] comes from the fact that in **FD** a derived sequent is decorated by a proof-term (a functional term), whereas in [57] an erasing function needs to be applied to the derivation to obtain the proof-term. Following [82], we shall call *contracting map* the function (called κ in [56]) which erases only the first-order part of formulas and *forgetful map* the function (also called κ in [57]) which also erases parts of formulas isomorphic to **unit**. Now, if Π is a derivation of a sequent $\Sigma \vdash \varphi$ in $\mathbf{IT}(\mathbb{N})$, then $\Sigma \vdash t \colon \varphi$ is derivable in **FD** where t is obtained by applying the contracting map to Π. Conversely, if Π is a derivation of $\Sigma \vdash t \colon \varphi$ in **FD**, then Π is also derivation of $\Sigma \vdash \varphi$ in $\mathbf{IT}(\mathbb{N})$ (just remove the proof-terms from the derivation).

Notation 2.5. *We shall write $\Sigma \vdash \varphi$ if $\Sigma \vdash t \colon \varphi$ is derivable in* **FD** *for some proof-term t, or equivalently, if $\Sigma \vdash \varphi$ is derivable in* $\mathbf{IT}(\mathbb{N})$.

Let us recall the formal definition of κ and derive the representation theorem for **FD** from Leivant's theorem for $\mathbf{IT}(\mathbb{N})$ [57]. For simplicity, we consider only pairs in the following definitions but the extension to arbitrary tuples, although technical, does not present any difficulty.

Definition 2.6. (Forgetful map for types). *For any functional dependent type φ its computational content $\kappa\varphi$ is defined by induction as follows:*

- $\kappa(n = m) = \mathbf{unit}$
- $\kappa(\mathbf{nat}(n)) = \mathbf{nat}$
- $\kappa(\exists \vec{\imath}\,(\varphi_1 \wedge \varphi_2)) = \begin{cases} \kappa\varphi_1 & \textit{if } \kappa\varphi_2 = \mathbf{unit} \\ \kappa\varphi_2 & \textit{if } \kappa\varphi_1 = \mathbf{unit} \\ \kappa\varphi_1 \times \kappa\varphi_2 & \textit{otherwise} \end{cases}$
- $\kappa(\forall \vec{\imath}\,(\varphi_1 \Rightarrow \varphi_2)) = \begin{cases} \kappa\varphi_2 & \textit{if } \kappa\varphi_1 = \mathbf{unit} \\ \mathbf{unit} & \textit{if } \kappa\varphi_2 = \mathbf{unit} \\ \kappa\varphi_1 \rightarrow \kappa\varphi_2 & \textit{otherwise} \end{cases}$

Definition 2.7. (Forgetful map for terms). *Given a term t such that $\Sigma \vdash t \colon \varphi$ is derivable, $\kappa(\Sigma \vdash t \colon \varphi)$ is defined by induction on the typing derivation. If $\kappa\varphi = \mathbf{unit}$ then $\kappa(\Sigma \vdash t \colon \varphi) = ()$ and otherwise define $\kappa(\Sigma \vdash t \colon \varphi)$ by cases:*

- (IDENT)
  $\kappa(\Sigma, x \colon \varphi \vdash x \colon \varphi) = x$
- (ZERO)
  $\kappa(\Sigma \vdash 0 \colon \mathbf{nat}(0)) = 0$
- (SUCC)
  $\kappa(\Sigma \vdash S(t) \colon \mathbf{nat}(\mathbf{s}(n))) = S(t')$ *where* $t' = \kappa(\Sigma \vdash t \colon \mathbf{nat}(n))$
- (PRED)

$\kappa(\Sigma \vdash \mathbf{pred}(t) \colon \mathbf{nat}(\mathbf{p}(n))) = \mathbf{pred}(t')$ *where* $t' = \kappa(\Sigma \vdash t \colon \mathbf{nat}(n))$

- (TUPLE)

  $$\kappa(\Sigma \vdash (t_1, t_2) \colon \exists \vec{\imath}\,(\varphi_1 \wedge \varphi_2)) = \begin{cases} t_1' & \text{if } \kappa\varphi_2 = \mathbf{unit} \\ t_2' & \text{if } \kappa\varphi_1 = \mathbf{unit} \\ (t_1', t_2') & \text{otherwise} \end{cases}$$

  where $t_1' = \kappa(\Sigma \vdash t_1 \colon \varphi_1[\vec{m}/\vec{\imath}])$ *and* $t_2' = \kappa(\Sigma \vdash t_2 \colon \varphi_2[\vec{m}/\vec{\imath}])$

- (LET)

  $$\kappa(\Sigma \vdash \mathbf{let}\ (x_1, x_2) = u\ \mathbf{in}\ t \colon \psi) = \begin{cases} \mathbf{let}\ x_2 = u'\ \mathbf{in}\ t'[()/x_1] & \text{if } \kappa\varphi_1 = \mathbf{unit} \\ \mathbf{let}\ x_1 = u'\ \mathbf{in}\ t'[()/x_2] & \text{if } \kappa\varphi_2 = \mathbf{unit} \\ \mathbf{let}\ (x_1, x_2) = u'\ \mathbf{in}\ t' & \text{otherwise} \end{cases}$$

  where $t' = \kappa(\Sigma, x_1 \colon \varphi_1, x_2 \colon \varphi_2 \vdash t \colon \psi)$ *and* $u' = \kappa(\Sigma \vdash u \colon \exists \vec{\imath}\,(\varphi_1 \wedge \varphi_2))$

- (ABS)

  $$\kappa(\Sigma \vdash \lambda x.t : \forall \vec{\imath}\,(\varphi \Rightarrow \psi)) = \begin{cases} t'[()/x] & \text{if } \kappa\varphi = \mathbf{unit} \\ \lambda x.t' & \text{otherwise} \end{cases}$$

  where $t' = \kappa(\Sigma, x \colon \varphi \vdash t : \psi)$

- (APP)

  $$\kappa(\Sigma \vdash (t_1\ t_2) \colon \psi[\vec{n}/\vec{\imath}]) = \begin{cases} t_1' & \text{if } \kappa\varphi = \mathbf{unit} \\ t_1'\ t_2' & \text{otherwise} \end{cases}$$

  where $t_1' = \kappa(\Sigma \vdash t_1 \colon \forall \vec{\imath}\,(\varphi \Rightarrow \psi))$ *and* $t_2' = \kappa(\Sigma \vdash t_2 \colon \varphi[\vec{n}/\vec{\imath}])$

- (REC)

  $\kappa(\Sigma \vdash \mathbf{rec}(t_1, t_2, \lambda x.\lambda y.t_3) : \varphi[n/i]) = \mathbf{rec}(t_1', t_2', \lambda x.\lambda y.t_3')$
  where $t_1' = \kappa(\Sigma \vdash t_1 \colon \mathbf{nat}(n))$, $t_2' = \kappa(\Sigma \vdash t_2 \colon \varphi[\mathbf{0}/i])$ *and* $t_3' = \kappa(\Sigma, x \colon \mathbf{nat}(i), y \colon \varphi \vdash t_3 \colon \varphi[\mathbf{s}(i)/i])$

- (EQUAL)

  $\kappa(\Sigma \vdash () \colon n = m) = ()$

- (SUBST)

  $\kappa(\Sigma \vdash t \colon \varphi[m/i]) = t'$ *where* $t' = \kappa(\Sigma \vdash t \colon \varphi[n/i])$

Definition 2.8. *A formula φ such that $\kappa\varphi = \mathbf{unit}$ is said to be irrelevant.*

Notation 2.9. *Although the computational content of a term t actually depends on its typing derivation, we shall write simply κt instead of $\kappa(\Sigma \vdash t \colon \varphi)$ whenever its typing judgment is clear from the context.*

Proposition 2.10. *If $\Sigma \vdash t \colon \varphi$ is derivable in* **FD** *then $\kappa\Sigma \vdash \kappa t \colon \kappa\varphi$ is derivable in* **FS**.

We also obtain the representation theorem for **FD** as a corollary of the same property for $\mathbf{IT}(\mathbb{N})$ (from [57], Theorem 36).

Definition 2.11. *Given a function $f \colon \mathbb{N}^k \rightarrow \mathbb{N}$ and a term t, we say that the term t represents the function f if $(t\ (\bar{q}_0, \ldots, \bar{q}_k)) \rightsquigarrow^\star \overline{f(q_0, \ldots, q_k)}$ where $\bar{q}$ is a notation for $S^q(0)$.*

Proposition 2.12. (Representation theorem for **FD**). *Given an equational system $\mathcal{E}$ and a k-ary function symbol f, if $\vdash t \colon \forall n_1, \ldots, n_k.(\mathbf{nat}(n_1) \wedge \ldots \wedge \mathbf{nat}(n_k)) \Rightarrow \mathbf{nat}(f(n_1, \ldots, n_k))$ is derivable in* **FD** *then κt represents f.*

2.5 Proof obligations

Since *irrelevant* proof-terms are erased by the contraction map it is natural to allow for incomplete proof-terms. The main advantage of this approach is to enable the generation of so-called *proof-obligations* and rely on external tools to decide whether $\Sigma \vdash \varphi$ is intuitionistically valid (φ *irrelevant*). We thus introduce the following deduction rule, where the question mark is used to denote missing parts in proof-terms (and the first premise corresponds to the proof-obligation):

$$\frac{\Sigma \vdash t \colon \varphi}{\Sigma \vdash ? \colon \varphi}\ (\varphi\ \textit{irrelevant})$$

The following lemma states that if we are only interested in the computational content of proofs we can always dispense with proof-terms of *irrelevant* formulas.

Lemma 2.13. *If* $\Sigma \vdash t\colon \varphi$ *is derivable in* **FD** *then* κt *is not incomplete (i.e.* κt *contains no question mark).*

2.6 Continuations

Since we are interested in imperative programs with non-local jumps, we shall need a continuation semantics. As is well-known [38], it is possible to factor a continuation-passing style semantics [74] through Moggi's computational meta-language [62, 63]. Note that from a logical standpoint, through the formulas-as-types interpretation, a monad is actually a modality [16, 5].

Following [16], we write $\neg\varphi$ for $\varphi \Rightarrow o$ where the *answer type* o is a fixed propositional variable. The continuation monad ∇ is then defined as $\nabla\varphi = \neg\neg\varphi$ together with the following two abbreviations:

$$\begin{array}{rcl} \mathbf{val}\ u & = & \lambda z.(z\ u) \\ \mathbf{let\ val}\ x = u\ \mathbf{in}\ t & = & \lambda z.(u\ \lambda x.(t\ z)) \end{array}$$

Remark 2.14. Those abbreviations (taken from [72]) correspond of course to *unit* and *bind*, but they shall be more convenient in the next section for defining the functional translation of imperative programs with jumps.

Lemma 2.15. *The following typing rules are derivable in* **FD***:*

$$\frac{\Sigma \vdash u\colon \varphi}{\Sigma \vdash \mathbf{val}\ u\colon \nabla\varphi} \qquad \frac{\Sigma \vdash u\colon \nabla\varphi \qquad \Sigma, x\colon \varphi \vdash t\colon \nabla\psi}{\Sigma \vdash \mathbf{let\ val}\ x = u\ \mathbf{in}\ t\colon \nabla\psi}$$

Notation 2.16. *We write* **let val** $\vec{x} = u$ **in** t *as an abbreviation for* **let val** $z = u$ **in let** $\vec{x} = z$ **in** t *(z fresh).*

Moreover, in the continuation monad, control operators *callcc* and *throw* are definable as the following abbreviations [81]:

$$\begin{array}{rcl} \mathbf{callcc} & = & \lambda h.\lambda k.(h\ k\ k) \\ \mathbf{throw} & = & \lambda(k,a).\lambda k'.(k\ a) \end{array}$$

Lemma 2.17. *Abbreviations* **callcc** *and* **throw** *are typable in* **FD** *as follows:*

$$\begin{array}{rcl} \mathbf{callcc} & : & (\neg\varphi \Rightarrow \nabla\varphi) \Rightarrow \nabla\varphi \\ \mathbf{throw} & : & (\neg\varphi \wedge \varphi) \Rightarrow \nabla\psi \end{array}$$

Remark 2.18. This choice of control operators is taken from [37] but it would be equivalent to take for instance $\mathcal{A}$ and $\mathcal{C}$ from [29] as in [64, 65]. Note that we do not consider any direct style semantics of these operators in this paper but only this indirect semantics based on the continuation monad.

Remark 2.19. We assume for the moment that o is not irrelevant and that $\kappa(o) = o$ (actually, o corresponds to the answer type of the whole program which is always **nat** in our framework). As a consequence, $\nabla\varphi$ is never irrelevant, even if φ is irrelevant.

3 Classical Imperative Type Theory

The imperative language we consider is essentially language Loop$^{\omega}$ introduced in [22]. The extension to non-local jumps and the dependent type system is presented in [19], Chapter 3. The main difference is that we consider a minor variant of the language where expressions also include purely functional terms of System T. Note that we do not consider any operational semantics for **I** in this paper: its semantics is only defined by translation into **F** (an operational semantics for Loop$^{\omega}$, without jumps, is described in [22]).

3.1 Language I

The raw syntax of imperative programs is given below. In the following grammar, x, y, z and k range over a set of identifiers and ε denotes the empty sequence.

$$
\begin{array}{rrcl}
(\textit{command}) & c & ::= & \{s\}_{\vec{x}} \\
 & & | & \textbf{for } y := 0 \textbf{ until } e\ \{s\}_{\vec{x}} \\
 & & | & e(\vec{e}\,;\vec{y}\,) \quad | \quad y := e \quad | \quad \textbf{inc}(y) \quad | \quad \textbf{dec}(y) \\
 & & | & k\colon \{s\}_{\vec{x}} \quad | \quad \textbf{jump}(k, \vec{e}\,)_{\vec{z}}
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{sequence}) & s & ::= & \varepsilon \\
 & & | & c\,;\ s \\
 & & | & \textbf{cst } y = e;\ s \\
 & & | & \textbf{var } y := e;\ s
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{expression}) & e & ::= & t \quad | \quad y \quad | \quad \textbf{proc}\ (\textbf{in } \vec{y}\,; \textbf{out } \vec{z}\,)\ \{s\}
\end{array}
$$

Remark 3.1. *(No aliasing).* In order to avoid parameter-induced aliasing problems, we assume that all y_i are pairwise distinct in a procedure call $p(\vec{e}\,;\vec{y}\,)$.

Remark 3.2. *(Annotations).* In a block $\{s\}_{\vec{x}}$, the variables in $\vec{x}$ represent the *output* of the block (they should be visible mutable variables according to standard C-like scoping rules). Moreover, $\vec{x}$ must contain all the free mutable variables occurring in the sequence. Such annotations can automatically be inferred statically by taking, for instance, all the visible mutable variables.

Remark 3.3. *(No back-patching).* No free mutable variable is allowed in the body of a procedure (except its **out** parameters). This restriction is required to prevent the well-known technique called "tying the recursive knot" [52] which takes advantage of higher-order mutable variables (or function pointers) to define arbitrary recursive functions.

Remark 3.4. *(Jumps).* The syntax k: $\{s\}_{\vec{x}}$ corresponds to the declaration of a (first-class) label whereas $\textbf{jump}(k, \vec{e}\,)_{\vec{z}}$ corresponds to a "jump with parameters" to *the end* of the block annotated with the label given as argument (which is akin to the semantics of **escape**/**goto** from [86]). Note that the output variables $\vec{z}$ are written as a subscript since they are only here for typing purpose. Indeed, in contrast to a regular procedure call, a **jump** never returns (i.e. the sequence which follows the **jump** shall not be executed). In practice, a **jump** is always the last instruction of a block and $\vec{z}$ can thus be identified with the variables which annotate the block.

3.2 Imperative Dependent Type System

The imperative language is equipped with a dependent type system (called $\mathbf{ID}^c$) which extends **FD**. The syntax of imperative dependent types, with higher-order procedures and first-class labels is the following:

$$\sigma, \tau \ ::= \ \varphi \quad | \quad \textbf{proc}\ \forall \vec{\imath}\,(\textbf{in } \vec{\tau}\,; \exists \vec{\jmath}\,\textbf{out } \vec{\sigma}\,) \quad | \quad \textbf{label}\ \exists \vec{\jmath}\,.\vec{\sigma}$$

A typing environment has the form $\Gamma;\Omega$ where Γ and Ω are (possibly empty) lists of pairs $x\colon\tau$ (x ranges over variables and τ over types). Γ stands for read-only variables (constants and **in** parameters) and Ω stands for mutable variables (local variables and **out** parameters). We use two typing judgments, one for expressions and one for sequences: $\Gamma;\Omega \vdash e\colon\tau$ has the usual meaning, whereas in $\Gamma;\Omega \vdash c \rhd \exists\vec{\jmath}\,.\Omega'$, the environment Ω' contains the (existentially quantified) types of the mutable variables at the end of the command c (and similarly for sequences). In particular, the domain of Ω' is always a subset of the domain of Ω (but, as explained in the introduction, the types of the variables may have changed).

For simplicity, we also assume that constants, mutable variables and logical variables belong to disjoint name spaces (i.e. a variable cannot occur both in a program and in a type). As usual, we consider programs up to renaming of bound variables, where the notion of free variable of a command is defined in the standard way. The dependent type system is summarized in Figure 3.1

(T.TERM)
$$\frac{\Gamma, \Omega \vdash_{\mathbf{FD}} t \colon \varphi}{\Gamma; \Omega \vdash t \colon \varphi}$$

(T.IDENT)
$$\frac{x \colon \tau \in \Gamma; \Omega}{\Gamma; \Omega \vdash x \colon \tau}$$

(T.PROC)
$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\top} \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\tau}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\,)\,\{s\} \colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,; \exists \vec{\jmath}\,\mathbf{out}\ \vec{\tau}\,)} \qquad \vec{\imath} \notin \mathcal{FV}(\Gamma)$$

(T.SUBST-I)
$$\frac{\Gamma; \Omega \vdash e \colon \tau[n/i] \qquad \Gamma, \Omega \vdash_{\mathbf{FD}} t \colon n = m}{\Gamma; \Omega \vdash e \colon \tau[m/i]}$$

(T.SUBST-II)
$$\frac{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'[n/i] \qquad \Gamma, \Omega \vdash_{\mathbf{FD}} t \colon n = m}{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'[m/i]}$$

(T.EMPTY)
$$\frac{}{\Gamma; \Omega, \vec{z} : \vec{\tau}\,[\vec{m}/\vec{\imath}\,] \vdash \varepsilon \rhd \exists \vec{\imath}\,.\vec{z} : \vec{\tau}}$$

(T.SEQ)
$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau} \qquad \Gamma; \Omega, \vec{x} : \vec{\tau} \vdash s \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash c;\ s\ \rhd \exists \vec{\jmath}\,.\Omega'} \qquad \vec{\imath} \notin \mathcal{FV}(\Gamma, \Omega, \exists \vec{\jmath}\,.\Omega')$$

(T.CST)
$$\frac{\Gamma; \Omega \vdash e \colon \tau \qquad \Gamma, y \colon \tau; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega \vdash \mathbf{cst}\ \ y = e;\ \ s\ \rhd \exists \vec{\jmath}\,.\Omega'}$$

(T.VAR)
$$\frac{\Gamma; \Omega \vdash e \colon \tau \qquad \Gamma; \Omega, y \colon \tau \vdash s \rhd \exists \vec{\jmath}\,.\Omega' \qquad y \notin \Omega'}{\Gamma; \Omega \vdash \mathbf{var}\ \ y := e;\ \ s\ \rhd \exists \vec{\jmath}\,.\Omega'}$$

(T.ASSIGN)
$$\frac{\Gamma; \Omega, y \colon \sigma \vdash e \colon \tau}{\Gamma; \Omega, y \colon \sigma \vdash y := e \rhd \Omega, y \colon \tau}$$

(T.INC)
$$\frac{}{\Gamma; \Omega, y \colon \mathbf{nat}(n) \vdash \mathbf{inc}(y) \rhd \Omega, y \colon \mathbf{nat}(\mathbf{s}(n))}$$

(T.DEC)
$$\frac{}{\Gamma; \Omega, y \colon \mathbf{nat}(n) \vdash \mathbf{dec}(y) \rhd \Omega, y \colon \mathbf{nat}(\mathbf{p}(n))}$$

(T.BLOCK)
$$\frac{\Gamma; \vec{x} : \vec{\tau} \vdash s \rhd \exists \vec{\jmath}\,.\vec{x} : \vec{\tau}\,'}{\Gamma; \Omega, \vec{x} : \vec{\tau} \vdash \{s\}_{\vec{x}} \rhd \exists \vec{\jmath}\,.\vec{x} : \vec{\tau}\,'}$$

(T.FOR)
$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,[\mathbf{0}/i] \vdash e \colon \mathbf{nat}(n) \qquad \Gamma, y \colon \mathbf{nat}(i); \vec{x} : \vec{\sigma} \vdash s \rhd \exists \vec{\jmath}\,.\vec{x} : \vec{\sigma}\,[\mathbf{s}(i)/i]}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,[\mathbf{0}/i] \vdash \mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s\}_{\vec{x}} \rhd \exists \vec{\jmath}\,.\vec{x} : \vec{\sigma}\,[n/i]} \qquad i, \vec{\jmath} \notin \mathcal{FV}(\Gamma)$$

(T.CALL)
$$\frac{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p \colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,; \exists \vec{\jmath}\,\mathbf{out}\ \vec{\tau}\,) \qquad \Gamma; \Omega, \vec{r} : \vec{\omega} \vdash \vec{e} : \vec{\sigma}\,[\vec{m}/\vec{\imath}\,]}{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p(\vec{e}\,; \vec{r}\,) \rhd \exists \vec{\jmath}\,.\vec{r} : \vec{\tau}\,[\vec{m}/\vec{\imath}\,]}$$

(T.LABEL)
$$\frac{\Gamma, k \colon \mathbf{label}\ \exists \vec{\jmath}\,.\vec{\sigma}\,; \vec{z} : \vec{\tau} \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}{\Gamma; \Omega, \vec{z} : \vec{\tau} \vdash k \colon \{s\}_{\vec{z}} \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

(T.JUMP)
$$\frac{\Gamma; \Omega, \vec{z} : \vec{\tau} \vdash k \colon \mathbf{label}\ \exists \vec{\jmath}\,.\vec{\sigma} \qquad \Gamma; \Omega, \vec{z} : \vec{\tau} \vdash \vec{e} : \vec{\sigma}\,[\vec{m}/\vec{\jmath}\,]}{\Gamma; \Omega, \vec{z} : \vec{\tau} \vdash \mathbf{jump}(k, \vec{e}\,)_{\vec{z}} \rhd \vec{z} : \vec{\tau}\,'}$$

Figure 3.1. Imperative dependent type system $\mathbf{ID}^c$

Remark 3.5. In rule (T.PROC), we write $\top$ simply as an abbreviation for $0 = 0$. Indeed, since the type of a mutable variable is updated by an assignment, we can freely assume that the type of an uninitialized variable is always $\top$. For instance, we can now introduce the following abbreviation which allows us to declare a procedure p with **out** parameters initialized by arbitrary default values:

$$\mathbf{proc}\ p(\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z} := \vec{e}\,)\,\{s_1\};\ \ s_2 \quad \equiv \quad \mathbf{cst}\ \vec{z}\,' := \vec{e}\,;\ \ \mathbf{cst}\ p = \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\,)\,\{\vec{z} := \vec{z}\,'; s_1\};\ \ s_2$$

The following typing rule, called (T.PROC'), can easily be derived for this abbreviation:

$$\frac{\Gamma;\Omega\vdash \vec{e}:\vec{\tau}\,' \qquad \Gamma,\vec{y}:\vec{\sigma};\vec{z}:\vec{\tau}\,'\vdash s_1\rhd\exists\vec{\jmath}\,.\vec{z}:\vec{\tau} \qquad \Gamma,p:\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma};\exists\vec{\jmath}\ \mathbf{out}\ \vec{\tau});\Omega\vdash s_2\rhd\exists\vec{\jmath}\,.\Omega'}{\Gamma;\Omega\vdash\mathbf{proc}\ p(\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z}:=\vec{e}\,)\,\{s_1\};\ s_2\rhd\exists\vec{\jmath}\,.\Omega'}$$

Remark 3.6. In rule (T.FOR), the types $\vec{\sigma}$ of the mutable variables which occur in the body correspond to the loop invariant. Consequently, $\vec{\sigma}\,[\mathbf{0}/i]$ should hold before the for-loop, in particular when type checking the upper bound e. Then, assuming that the body preserves the invariant, $\vec{\sigma}\,[n/i]$ holds after the for-loop.

Remark 3.7. In rule (T.LABEL), the type of the declared first-class label corresponds to the output types of the mutable variables of the sequence (it is thus also existentially quantified). Accordingly, rule (T.JUMP) requires that the type of the argument be an instance of the existential type of the label. Moreover, the mutable variables $\vec{z}$ (written as a subscript) can be updated to whatever type is required by the context. Note that this possibility is essential to obtain classical logic, otherwise we could erase every **jump** from a well-typed program (and, as a consequence, every label) and get a program typable with *the same dependent type* in intuitionistic logic.

Remark 3.8. As in the functional case, the conditional command is derivable from the for-loop. The expected typing rule is the following:

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash e:\mathbf{nat}(n) \qquad \Gamma,h:n\neq 0;\vec{x}:\vec{\tau}\vdash s_1\rhd\exists\vec{\jmath}\,.\vec{x}:\vec{\sigma} \qquad \Gamma,h:n=0;\vec{x}:\vec{\tau}\vdash s_2\rhd\exists\vec{\jmath}\,.\vec{x}:\vec{\sigma}}{\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash\mathbf{if}\ e\ \mathbf{then}\ \{s_1\}_{\vec{x}}\ \mathbf{else}\ \{s_2\}_{\vec{x}}\rhd\exists\vec{\jmath}\,.\vec{x}:\vec{\sigma}}$$

Recall that in a block $\{s\}_{\vec{x}}$ the only mutable variables which can occur in s are from $\vec{x}$, this typing rule is thus consistant with the typing rule for blocks. For the implementation, the intuition is the same as in the functional case: we introduce a procedural variable p which is initialized by a procedure which shall execute s_2 when invoked. If the value of e is not 0 then, using a for-loop, this procedural variable is overwritten by another procedure which shall execute s_1 when invoked. To complete the execution of the conditional, p is finally invoked. The following definition implements this idea (we rely here on the abbreviation defined in Remark 3.5):

```
$\mathbf{if}\ e\ \mathbf{then}\ \{s_1\}_{\vec{x}}\ \mathbf{else}\ \{s_2\}_{\vec{x}}\ \equiv$
    $\mathbf{cst}\ v=e;$
    $\{$
        $\mathbf{cst}\ \vec{x}\,'=\vec{x};$
        $\mathbf{proc}\ q_2(\mathbf{in}\ h;\ \mathbf{out}\ \vec{x}:=\vec{x}\,')\ \{s_2\}_{\vec{x}};$
        $\mathbf{var}\ p:=q_2;$
        $\mathbf{for}\ y:=0\ \mathbf{until}\ v\ \{$
            $\mathbf{proc}\ q_1(\mathbf{in}\ h';\ \mathbf{out}\ \vec{x}:=\vec{x}\,')\ \{\mathbf{cst}\ h=h';s_1\}_{\vec{x}};$
            $p:=q_1;$
        $\}_p;$
        $p((\,);\vec{x}\,);$
    $\}_{\vec{x}}$
```

The complete typing derivation is given in Appendix B. Note that the presence of the additional variable h in the typing rule can be seen as an artifact of the encoding. Fortunately, this artifact shall disappear when we reformulate the rule using Hoare triples (in Section 4).

Remark 3.9. Rule (T.TERM) relies on system **FD** to type check functional terms (since φ belongs to the language defined in section 2.3). Note that the environment Γ, Ω may contain procedure or label types, but this is harmless since rule (IDENT) from **FD** requires identifiers to have purely functional types. Similarly, rules (T.SUBST-I) and (T.SUBST-II) rely on system **FD** to check equality proofs.

Remark 3.10. We could dispense with existential types in the typing of commands (and sequences) by introducing explicit existentially typed records. The resulting type system would be closer to **FD**, with procedure types corresponding to dependent products and records types corresponding to dependent sums. However, we found out that the above type system is more convenient to encode Hoare judgments (see Remark 4.3).

3.3 Translation from $\mathbf{ID}^c$ to $\mathbf{FD}$

We define here the translation * from $\mathbf{ID}^c$ to $\mathbf{FD}$. This translation implements a CPS-transform for commands, sequences and procedures but leaves functional terms in direct style. We prove that translation * preserves dependent types and, as a corollary, we obtain the representation theorem for $\mathbf{ID}^c$.

Definition 3.11. (Translation of dependent types). *For any imperative dependent type τ, the corresponding functional dependent type $\tau^\star$ is defined inductively as follows:*

- $(\varphi)^\star = \varphi$
- $(\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\tau}\,; \exists\vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}\,))^\star = \forall\vec{\imath}\,(\vec{\tau}^{\,\star} \Rightarrow \nabla\exists\vec{\jmath}\,.\vec{\sigma}^{\,\star})$
- $(\mathbf{label}\ \exists\vec{\jmath}\,.\vec{\sigma}\,)^\star = \neg\exists\vec{\jmath}\,.\vec{\sigma}^{\,\star}$

Definition 3.12. *For any expression e, sequence s and variables $\vec{x}$, the translations $e^\star$ and $(s)^\star_{\vec{x}}$ into terms of language $\mathbf{F}$ are defined by mutual induction as follows:*

- $t^\star = t$
- $y^\star = y$
- $(\mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\,)\,\{s\})^\star = \lambda\vec{y}\,.(s)^\star_{\vec{z}}\,[\vec{()}/\vec{z}\,]$
- $(\varepsilon)^\star_{\vec{x}} = \mathbf{val}\ \vec{x}$
- $(\mathbf{var}\ \ y := e;\ \ s)^\star_{\vec{x}} = (s)^\star_{\vec{x}}[e^\star/y]$
- $(\mathbf{cst}\ \ y = e;\ \ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(y{:=}e;\ \ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{inc}(y);\ \ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{dec}(y);\ \ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(p(\vec{e}\,; \vec{z}\,);\ \ s)^\star_{\vec{x}} = \mathbf{let}\ \mathbf{val}\ \vec{z}\, = (p^\star\ \vec{e}^{\,\star})\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\{s_1\}_{\vec{z}};\ \ s_2)^\star_{\vec{x}} = \mathbf{let}\ \ \mathbf{val}\ \vec{z}\, = (s_1)^\star_{\vec{z}}\ \ \mathbf{in}\ \ (s_2)^\star_{\vec{x}}$
- $(\mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s_1\}_{\vec{z}};\ \ s_2)^\star_{\vec{x}} = \mathbf{let}\ \ \mathbf{val}\ \vec{z}\, = \mathbf{rec}(e^\star, \mathbf{val}\ \vec{z}\,, \lambda y.\lambda r.\mathbf{let}\ \ \mathbf{val}\ \vec{z}\, = r\ \ \mathbf{in}\ \ (s_1)^\star_{\vec{z}})\ \ \mathbf{in}\ \ (s_2)^\star_{\vec{x}}$
- $(k{:}\,\{s_1\}_{\vec{z}};\ \ s_2)^\star_{\vec{x}} = \mathbf{let}\ \ \mathbf{val}\ \vec{z}\, = \mathbf{callcc}\ \lambda k.(s_1)^\star_{\vec{z}}\ \ \mathbf{in}\ \ (s_2)^\star_{\vec{x}}$
- $(\mathbf{jump}\ \ (k, \vec{e}\,)_{\vec{z}}; s)^\star_{\vec{x}} = \mathbf{let}\ \ \mathbf{val}\ \vec{z}\, = \mathbf{throw}\ \ (k, \vec{e}^{\,\star})\ \ \mathbf{in}\ \ (s)^\star_{\vec{x}}$

Remark 3.13. Note that the assignment is translated into a **let** (and not a **let val**). As a consequence, an assignment of a irrelevant term is completely erased by κ (and similarly irrelevant constant declarations are completely erased). On the other hand, a **let val** is never erased (since $\nabla\varphi$ is never irrelevant, by Remark 2.19).

Theorem 3.14. (Soundness for $\mathbf{ID}^c$). *For any environments Γ and Ω, any expression e, any sequence s we have:*

- $\Gamma; \Omega \vdash e{:}\,\tau$ *in* $\mathbf{ID}^c$ *implies* $\Gamma^\star, \Omega^\star \vdash e^\star{:}\,\tau^\star$ *in* $\mathbf{FD}$.
- $\Gamma; \Omega \vdash s \rhd \exists\vec{\jmath}\,.\vec{z}{:}\,\vec{\sigma}$ *in* $\mathbf{ID}^c$ *implies* $\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}{:}\,\nabla\exists\vec{\jmath}\,.\vec{\sigma}^{\,\star}$ *in* $\mathbf{FD}$.

Proof. We proceed by induction on the typing derivation:

- (T.TERM)

$$\frac{\Gamma, \Omega \vdash t{:}\,\varphi}{\Gamma; \Omega \vdash t{:}\,\varphi}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash t^\star{:}\,\varphi^\star}{\Gamma^\star, \Omega^\star \vdash t^\star{:}\,\varphi^\star}$$

- (T.IDENT)

$$\frac{y{:}\,\tau \in \Gamma, \Omega}{\Gamma; \Omega \vdash y{:}\,\tau}$$

Indeed,

$$\frac{y\colon \tau^\star \in \Gamma^\star, \Omega^\star}{\Gamma^\star, \Omega^\star \vdash y\colon \tau^\star}$$

- (T.SUBST-I)

$$\frac{\Gamma; \Omega \vdash e\colon \tau[n/i] \quad \Gamma, \Omega \vdash t\colon n = m}{\Gamma; \Omega \vdash e\colon \tau[m/i]}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star\colon \tau^\star[n/i] \quad \Gamma^\star, \Omega^\star \vdash t^\star\colon n = m}{\Gamma^\star, \Omega^\star \vdash e^\star\colon \tau^\star[m/i]}$$

- (T.SUBST-II)

$$\frac{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}\,[n/i] \quad \Gamma, \Omega \vdash t\colon n = m}{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}\,[m/i]}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star[n/i] \quad \Gamma^\star, \Omega^\star \vdash t^\star\colon n = m}{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star[m/i]}$$

- (T.EMPTY)

$$\frac{}{\Gamma; \Omega, \vec{z} : \vec{\sigma}\,[\vec{m}/\vec{\jmath}\,] \vdash \varepsilon \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

Indeed,

$$\frac{\dfrac{}{\Gamma^\star, \Omega^\star, \vec{z} : \vec{\sigma}^\star[\vec{m}/\vec{\jmath}\,] \vdash \vec{z} : \exists \vec{\jmath}\,.\vec{\sigma}^\star}}{\Gamma^\star, \Omega^\star, \vec{z} : \vec{\sigma}^\star[\vec{m}/\vec{\jmath}\,] \vdash \mathbf{val}\ \vec{z} : \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.CST)

$$\frac{\Gamma; \Omega \vdash e\colon \tau \qquad \Sigma, y\colon \tau; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}{\Gamma; \Omega \vdash \mathbf{cst}\ y = e;\ s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star\colon \tau^\star \qquad \Gamma^\star, y\colon \tau^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{z}} : \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.VAR)

$$\frac{\Gamma; \Omega \vdash e\colon \tau \qquad \Gamma; \Omega, y\colon \tau \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma} \qquad y \notin \vec{z}}{\Gamma; \Omega \vdash \mathbf{var}\ y := e;\ s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

Indeed, by the substitution lemma (see [57], section 5)

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star\colon \tau^\star \qquad \Gamma^\star, y\colon \tau^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}[e^\star/y] : \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.BLOCK)

$$\frac{\Gamma; \vec{x} : \vec{\tau} \vdash s \rhd \exists \vec{\kappa}\,.\vec{x} : \vec{\sigma}\,' \qquad \Gamma; \Omega, \vec{x} : \vec{\sigma}\,' \vdash s' \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}{\Gamma; \Omega, \vec{x} : \vec{\tau} \vdash \{s\}_{\vec{x}}; s' \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

with $\vec{\kappa} \notin \mathcal{FV}(\Gamma, \Omega, \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}\,)$. Indeed,

$$\frac{\Gamma^\star, \vec{x} : \vec{\tau}^\star \vdash (s)^\star_{\vec{x}}\colon \nabla \exists \vec{\kappa}\,.\vec{\sigma}\,'^\star \qquad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}\,'^\star \vdash (s')^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\tau}^\star \vdash \mathbf{let\ val}\ \vec{x} = (s)^\star_{\vec{x}}\ \mathbf{in}\ (s')^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}$$

since $\vec{\kappa} \notin \mathcal{FV}(\Gamma^\star, \Omega^\star, \exists \vec{\jmath}\,.\vec{\sigma}^\star)$.

- (T.INC)

$$\frac{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y) \rhd \Omega, y\colon \mathbf{nat}(\mathbf{s}(n)) \qquad \Gamma; \Omega, y\colon \mathbf{nat}(\mathbf{s}(n)) \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y);\ s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(n) \vdash \mathbf{succ}(y)\colon \mathbf{nat}(\mathbf{s}(n)) \qquad \Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(\mathbf{s}(n)) \vdash (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(n) \vdash \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s)^\star_{\vec{z}}\colon \nabla \exists \vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.DEC)

$$\frac{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y) \rhd \Omega, y\colon \mathbf{nat}(\mathbf{p}(n)) \qquad \Gamma; \Omega, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y);\ s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\sigma}}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(n) \vdash \mathbf{pred}(y)\colon \mathbf{nat}(\mathbf{p}(n)) \qquad \Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(n) \vdash \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.ASSIGN)

$$\frac{\Gamma; \Omega, y\colon \sigma' \vdash e\colon \tau \qquad \Gamma; \Omega, y\colon \tau \vdash s \rhd \exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}}{\Gamma; \Omega, y\colon \sigma' \vdash y := e;\ s \rhd \exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}}$$

Indeed,

$$\frac{\Gamma^\star, \Omega^\star, y\colon \sigma'^\star \vdash e^\star\colon \tau^\star \qquad \Gamma^\star, \Omega^\star, y\colon \tau^\star \vdash (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star}{\Gamma^\star, \Omega^\star, y\colon \sigma'^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star}$$

- (T.FOR)

$$\frac{\Gamma; \Omega, \vec{x}:\vec{\sigma}\,[\mathbf{0}/i] \vdash e\colon \mathbf{nat}(n) \quad \Gamma, y\colon \mathbf{nat}(i); \vec{x}:\vec{\sigma} \vdash s \rhd \exists\vec{\jmath}\,.\vec{x}:\vec{\sigma}\,[\mathbf{s}(i)/i] \quad \Gamma; \Omega, \vec{x}:\vec{\sigma}\,[n/i] \vdash s' \rhd \exists\vec{\kappa}\,.\vec{z}:\vec{\sigma}\,'}{\Gamma; \Omega, \vec{x}:\vec{\sigma}\,[\mathbf{0}/i] \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ s' \rhd \exists\vec{\kappa}\,.\vec{z}:\vec{\sigma}\,'}$$

with $i, \vec{\jmath} \notin \mathcal{FV}(\Gamma)$ and $\vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega, \exists\vec{\kappa}\,.\vec{z}:\vec{\sigma}\,')$. Indeed, we have:

$$\frac{\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[\mathbf{0}/i] \vdash \vec{x}:\exists\vec{\jmath}\,.\vec{\sigma}^\star[\mathbf{0}/i]}{\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{val}\ \vec{x}:\nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star[\mathbf{0}/i]}$$

and since $\vec{\jmath} \notin \mathcal{FV}(\Gamma^\star)$, we also have:

$$\frac{\Gamma^\star, r\colon \nabla\exists\vec{\jmath}\,.\,\vec{\sigma}^\star[\mathbf{0}/i] \vdash r\colon \nabla\exists\vec{\jmath}\,.\,\vec{\sigma}^\star[\mathbf{0}/i] \qquad \Gamma^\star, y\colon \mathbf{nat}(i), \vec{x}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{x}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star[\mathbf{s}(i)/i]}{\Gamma^\star, \Omega^\star, y\colon \mathbf{nat}(i), r\colon \nabla\exists\vec{\jmath}\,.\,\vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{let\ val}\ \vec{x} = r\ \mathbf{in}\ (s)^\star_{\vec{x}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star[\mathbf{s}(i)/i]}$$

The following rule is thus derivable since $i \notin \mathcal{FV}(\Gamma^\star)$:

$$\mathcal{D} = \frac{\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[\mathbf{0}/i] \vdash e^\star\colon \mathbf{nat}(n) \qquad \Gamma^\star, y\colon \mathbf{nat}(i), \vec{x}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{x}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star[\mathbf{s}(i)/i]}{\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{rec}(e^\star, \mathbf{val}\ \vec{x}\,, \lambda y.\lambda r.\mathbf{let\ val}\ \vec{x} = r\ \mathbf{in}\ (s)^\star_{\vec{x}})\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}^\star[n/i]}$$

and since $\vec{\jmath} \notin \mathcal{FV}(\Gamma^\star, \Omega^\star, \exists\vec{\kappa}\,.\vec{z}:\vec{\sigma}\,'^\star)$ we complete the derivation with:

$$\frac{\mathcal{D} \qquad \Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[n/i] \vdash (s')^\star_{\vec{z}}\colon \nabla\exists\vec{\kappa}\,.\vec{\sigma}\,'^\star}{\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{let\ val}\ \vec{x} = \mathbf{rec}(e^\star, \mathbf{val}\ \vec{x}\,, \lambda y.\lambda r.\mathbf{let\ val}\ \vec{x} = r\ \mathbf{in}\ (s)^\star_{\vec{x}})\ \mathbf{in}\ (s')^\star_{\vec{z}}\colon \nabla\exists\vec{\kappa}\,.\vec{\sigma}\,'^\star}$$

- (T.PROC)

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}:\vec{\sigma}; \vec{z}:\vec{\top} \vdash s \rhd \exists\vec{\jmath}\,.\vec{z}:\vec{\tau}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\,)\,\{s\}\colon \mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,; \exists\vec{\jmath}\ \ \mathbf{out}\ \vec{\tau}\,)}$$

with $\vec{\imath} \notin \mathcal{FV}(\Gamma)$. Indeed,

$$\frac{\dfrac{\dfrac{\Gamma^\star, \vec{y}:\vec{\sigma}^\star, \vec{z}:\vec{\top} \vdash (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\tau}^\star}{\Gamma^\star, \vec{y}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{z}}[\vec{()}/\vec{z}\,]\colon \nabla\exists\vec{\jmath}\,.\vec{\tau}^\star}}{\Gamma^\star \vdash \lambda\vec{y}\,.(s)^\star_{\vec{z}}[\vec{()}/\vec{z}\,]\colon \forall\vec{\imath}\,(\vec{\sigma}^\star \Rightarrow \nabla\exists\vec{\jmath}\,.\vec{\tau}^\star)}}{\Gamma^\star, \Omega^\star \vdash \lambda\vec{y}\,.(s)^\star_{\vec{z}}[\vec{()}/\vec{z}\,]\colon \forall\vec{\imath}\,(\vec{\sigma}^\star \Rightarrow \nabla\exists\vec{\jmath}\,.\vec{\tau}^\star)}$$

since $\vec{\imath} \notin \mathcal{FV}(\Gamma^\star)$.

- (T.CALL)

$$\frac{\Gamma; \Omega, \vec{r}:\vec{\omega} \vdash p\colon \mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\tau}\,; \exists\vec{\kappa}\ \ \mathbf{out}\ \vec{\sigma}\,) \quad \Gamma; \Omega, \vec{r}:\vec{\omega} \vdash \vec{e}:\vec{\tau}\,[\vec{n}/\vec{\imath}\,] \quad \Gamma; \Omega, \vec{r}:\vec{\sigma}\,[\vec{n}/\vec{\imath}\,] \vdash s \rhd \exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}\,'}{\Gamma; \Omega, \vec{r}:\vec{\omega} \vdash p(\vec{e}\,; \vec{r}\,)\,; s \rhd \exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}\,'}$$

with $\vec{\kappa} \notin \mathcal{FV}(\Gamma, \Omega, \exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}\,')$. Indeed, we have:

$$\frac{\Gamma^\star, \Omega^\star, \vec{r}:\vec{\omega}^\star \vdash p^\star\colon \forall\vec{\imath}\,(\vec{\tau}^\star \Rightarrow \nabla\exists\vec{\kappa}\,.\vec{\sigma}^\star) \quad \Gamma^\star, \Omega^\star, \vec{r}:\vec{\omega}^\star \vdash \vec{e}^\star\colon (\vec{\tau}^\star)[\vec{n}/\vec{\imath}\,]}{\Gamma^\star, \Omega^\star, \vec{r}:\vec{\omega}^\star \vdash (p^\star\ \vec{e}^\star)\colon \nabla\exists\vec{\kappa}\,.\vec{\sigma}^\star[\vec{n}/\vec{\imath}\,]}$$

and since $\vec{\kappa} \notin \mathcal{FV}(\Gamma^\star, \Omega^\star, \exists\vec{\jmath}\,.\vec{\sigma}\,'^\star)$ we complete the derivation with:

$$\frac{\Gamma^\star, \Omega^\star, \vec{r}:\vec{\omega}^\star \vdash (p^\star\ \vec{e}^\star)\colon \nabla\exists\vec{\kappa}\,.\vec{\sigma}^\star[\vec{n}/\vec{\imath}\,] \qquad \Gamma^\star, \Omega^\star, \vec{r}:\vec{\sigma}^\star[\vec{n}/\vec{\imath}\,] \vdash (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}\,'^\star}{\Gamma^\star, \Omega^\star, \vec{r}:\vec{\omega}^\star \vdash \mathbf{let\ val}\ \vec{r} = (p^\star\ \vec{e}^\star)\ \mathbf{in}\ (s)^\star_{\vec{z}}\colon \nabla\exists\vec{\jmath}\,.\vec{\sigma}\,'^\star}$$

- (T.LABEL)

$$\frac{\Gamma, k\colon \textbf{label}\ \exists\vec{\jmath}.\vec{\sigma}; \vec{x}:\vec{\tau} \vdash s \rhd \exists\vec{\jmath}.\vec{x}:\vec{\sigma} \qquad \Gamma;\Omega,\vec{x}:\vec{\sigma}\vdash s' \rhd \exists\vec{\kappa}.\vec{z}:\vec{\sigma}'}{\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash k\colon\{s\}_{\vec{x}}; s' \rhd \exists\vec{\kappa}.\vec{z}:\vec{\sigma}'}$$

Indeed, we have:

$$\frac{\vdash \textbf{callcc}\colon (\neg\exists\vec{\jmath}.\vec{\sigma}^\star \Rightarrow \nabla\exists\vec{\jmath}.\vec{\sigma}^\star)\Rightarrow\nabla\exists\vec{\jmath}.\vec{\sigma}^\star \qquad \dfrac{\Gamma^\star, k\colon\neg\exists\vec{\jmath}.\vec{\sigma}^\star, \vec{x}:\vec{\tau}^\star\vdash (s)^\star_{\vec{x}}\colon\nabla\exists\vec{\jmath}.\vec{\sigma}^\star}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\tau}^\star\vdash\lambda k.(s)^\star_{\vec{x}}\colon\neg\exists\vec{\jmath}.\vec{\sigma}^\star\Rightarrow\nabla\exists\vec{\jmath}.\vec{\sigma}^\star}}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\tau}^\star\vdash\textbf{callcc}\ \lambda k.(s)_{\vec{x}}\colon\nabla\exists\vec{\jmath}.\vec{\sigma}^\star}$$

and we complete the derivation with:

$$\frac{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\tau}^\star\vdash\textbf{callcc}\ \lambda k.(s)_{\vec{x}}\colon\nabla\exists\vec{\jmath}.\vec{\sigma}^\star \qquad \Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star\vdash(s')^\star_{\vec{z}}\colon\nabla\exists\vec{\kappa}.\vec{\sigma}'^\star}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\tau}^\star\vdash\textbf{let val}\ \vec{x}=\textbf{callcc}\ \lambda k.(s)^\star_{\vec{x}}\ \textbf{in}\ (s')^\star_{\vec{z}}\colon\nabla\exists\vec{\kappa}.\vec{\sigma}'^\star}$$

- (T.JUMP)

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\omega}\vdash k\colon\textbf{label}\ \exists\vec{\jmath}.\vec{\sigma}\qquad\Gamma;\Omega,\vec{x}:\vec{\omega}\vdash\vec{e}:\vec{\sigma}[\vec{m}/\vec{\jmath}]\qquad\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash s\rhd\exists\vec{\jmath}.\vec{z}:\vec{\sigma}'}{\Gamma;\Omega,\vec{x}:\vec{\omega}\vdash\textbf{jump}(k,\vec{e})_{\vec{x}}; s\rhd\exists\vec{\jmath}.\vec{z}:\vec{\sigma}'}$$

Indeed, we have:

$$\frac{\vdash\textbf{throw}\colon(\neg\exists\vec{\jmath}.\vec{\sigma}^\star\wedge\exists\vec{\jmath}.\vec{\sigma}^\star)\Rightarrow\nabla\vec{\tau}^\star\qquad\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash k^\star\colon\neg\exists\vec{\jmath}.\vec{\sigma}^\star\qquad\dfrac{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash\vec{e}^\star\colon\vec{\sigma}^\star[\vec{m}/\vec{\jmath}]}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash\vec{e}^\star\colon\exists\vec{\jmath}.\vec{\sigma}^\star}}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash\textbf{throw}\ (k^\star,\vec{e}^\star)\colon\nabla\vec{\tau}^\star}$$

and we complete the derivation with:

$$\frac{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash\textbf{throw}\ (k^\star,\vec{e}^\star)\colon\nabla\vec{\tau}^\star\qquad\Gamma^\star,\Omega^\star,\vec{x}:\vec{\tau}^\star\vdash(s')^\star_{\vec{z}}\colon\nabla\exists\vec{\jmath}.\vec{\sigma}'^\star}{\Gamma^\star,\Omega^\star,\vec{x}:\vec{\omega}^\star\vdash\textbf{let val}\ \vec{x}=\textbf{throw}\ (k^\star,\vec{e}^\star)\ \textbf{in}\ (s')^\star_{\vec{z}}\colon\nabla\exists\vec{\jmath}.\vec{\sigma}'^\star}$$

□

We are now ready to state and prove the representation theorem for dependently-typed imperative programs. This theorem is a corollary of the representation theorem for **FD** and the soundness theorem for $\mathbf{ID}^c$.

Definition 3.15. *Given a function $f\colon \mathbb{N}^k \to \mathbb{N}$ and an anonymous procedure p we say that p represents the function f if $(p^\star\ (\bar{q}_0,\dots,\bar{q}_k)\ id)\rightsquigarrow^\star\overline{f(q_0,\dots,q_k)}$ (where id is the identity function).*

Theorem 3.16. (Representation for $\mathbf{ID}^c$). *Given an equational system $\mathcal{E}$ and a k-ary function symbol f, if*

$$\vdash p\colon \textbf{proc}\ \forall n_1,\dots,n_k.(\textbf{in nat}(n_1),\dots,\textbf{nat}(n_k);\textbf{out nat}(f(n_1,\dots,n_k)))$$

is derivable in $\mathbf{ID}^c$ then p represents f.

Proof. By Theorem 3.14 $\vdash p^\star\colon\forall n_1,\dots,n_k.(\textbf{nat}(n_1)\wedge\dots\wedge\textbf{nat}(n_k))\Rightarrow\nabla\textbf{nat}(f(n_1,\dots,n_k))$ is derivable in **FD**. Using Friedman's top level trick [32, 64], we replace the answer type o by $\textbf{nat}(f(n_1,\dots,n_k))$ in the derivation and obtain that $\vdash\lambda\vec{x}.(p^\star\ \vec{x}\ \ id)\colon\forall n_1,\dots,n_k.(\textbf{nat}(n_1)\wedge\dots\wedge\textbf{nat}(n_k))\Rightarrow\textbf{nat}(f(n_1,\dots,n_k))$ is also derivable in **FD** and p represents thus f by Proposition 2.12. □

4 Hoare Dependent Type System

Informally, it is almost straightforward to embed a Floyd-Hoare logic into $\mathbf{ID}^c$. Indeed, let us take a global mutable variable, dubbed *assert*, and let us assume that this global variable is simulated in the usual *state-passing style*. Any sequence shall thus be typed with a sequent of the form $\Gamma;\Omega,\textit{assert}\colon\varphi\vdash s\rhd\exists\vec{\jmath}.\Omega',\textit{assert}\colon\psi$ (where φ and ψ are assumed to be irrelevant). If we now introduce the usual Hoare notation for triples (thus hiding the name of variable *assert*), we obtain *Hoare judgments* of the form $\Gamma;\Omega\{\varphi\}\vdash s\rhd\exists\vec{\jmath}.\Omega'\{\psi\}$ for sequences (and commands), $\Gamma;\Omega\{\varphi\}\vdash e\colon\sigma$ (for expressions). In accordance with Notation 4.2, we shall also write simply $\Gamma,\Omega,\varphi\vdash\psi$ for proof-obligations (where ψ is irrelevant).

Moreover, variable *assert* should also be passed as an implicit **in** and **out** parameter to each procedure call. Consequently, we first introduce the notation $\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,\{\varphi\};\ \exists\vec{\jmath}\,\mathbf{out}\ \vec{\tau}\,\{\psi\})$ for *pre*/*post* conditions in a procedure type as syntactic sugar for $\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,,\ \varphi;\ \exists\vec{\jmath}\,\mathbf{out}\ \vec{\tau}\,,\ \psi)$. Then, a procedure call $p(\vec{e}\,;\vec{r}\,)$ becomes an abbreviation for $p(\vec{e}\,,\ \mathit{assert};\ \vec{r}\,,\ \mathit{assert})$ and an anonymous procedure $\mathbf{proc}\ (\mathbf{in}\ \vec{y}\,;\ \mathbf{out}\ \vec{z}\,)\,\{s\}$ is now an abbreviation for $\mathbf{proc}\ (\mathbf{in}\ \vec{y}\,,\mathit{assert}';\mathbf{out}\ \vec{z}\,,\mathit{assert})\,\{\mathit{assert}{:=}\,\mathit{assert}';s\}$.

Similarly, we hide *assert* from annotations of blocks, labels, jumps and loop bodies where, as expected, the type of *assert* corresponds to an invariant. The dependent type system obtained by obeying the above conventions is summarized in figure 4.1.

Remark 4.1. The idea of simulating Hoare triples with a global variable in state-passing style is reminiscent of the Hoare State Monad [84]. However, this similarity is only superficial since the Hoare State Monad is useful for proving some property about a global state simulated in state-passing style (and the proof-term depends on the generated proof obligations) whereas, in our encoding, it is the proof-term itself which is built in state-passing style.

Notation 4.2. *We write* $\Gamma,\Omega\vdash\psi$ *for proof-obligations (*ψ *irrelevant) as an abbreviation for* $\Gamma,\Omega\vdash_{\mathbf{FD}}?{:}\,\psi$*.*

Remark 4.3. Although at first sight the remaining explicit existential quantifiers on the right-hand side of sequents may seem awkward, they are actually quite convenient since they permit the encoding of various styles of specifications. For instance, the following specification is valid for **inc**:

$$\Gamma;\Omega,x{:}\,\mathbf{nat}(n)\{\varphi\}\vdash\mathbf{inc}(x)\rhd\exists n'.\Omega,x{:}\,\mathbf{nat}(n')\{\varphi\wedge n'=s(n)\}$$

We followed here the convention from the Z notation [83] and primed the new value of variable x (we could equally use *hooked* variables as in VDM [47] to represent old values). We can also simulate Hoare auxiliary variables [49] and obtain a sequent where both the old and the new value of variable x are called n (and N is an auxiliary variable):

$$\Gamma;\Omega,x{:}\,\mathbf{nat}(n)\{\varphi\wedge N=n\}\vdash\mathbf{inc}(x)\rhd\exists n.\Omega,x{:}\,\mathbf{nat}(n)\{\varphi[N/n]\wedge n=s(N)\}$$

Note the fact that we are able to easily accommodate different specification styles is a consequence of our choosing Leivant's system $\mathbf{IT}(\mathbb{N})$ [57] (instead of the usual presentation of Heyting arithmetic) as a target language of our translation. Indeed, in our framework an integer program variable and its value are only related through the unary predicate **nat** and we can thus freely use different names for the value in a *before-after* predicate *à la* Z [83] or VDM [47].

Remark 4.4. Rule (H.ASSIGN) looks quite unusual since this rule allows the type of variable y to change (as in $\mathbf{ID}^c$). If we restrict this rule to natural numbers, we obtain the following rule:

$$\frac{\Gamma;\Omega,\,y{:}\,\sigma\,\{\varphi\}\vdash e{:}\,\mathbf{nat}(n)}{\Gamma;\Omega,\,y{:}\,\sigma\,\{\varphi\}\vdash y:=e\rhd\Omega,\,y{:}\,\mathbf{nat}(n)\{\varphi\}}$$

A rule closer to what one would expect can then be derived. Indeed, in order to guarantee that φ holds for m' (the value of y after the assignment), the usual Hoare axiom requires that φ holds for n (the value of e) before the assignment. In our framework, such a rule would take the following shape:

$$\frac{\Gamma;\,y{:}\,\sigma\,\{\varphi[n/m']\}\vdash e{:}\,\mathbf{nat}(n)}{\Gamma;\,y{:}\,\sigma\,\{\varphi[n/m']\}\vdash y:=e;\rhd\exists m'.y{:}\,\mathbf{nat}(m')\{\varphi\}}$$

Let us call the above rule (H.ASSIGN') and let us prove that it is indeed derivable using rules (H.EMPTY) and (H.SEQ):

$$\frac{\dfrac{\Gamma;\,y{:}\,\sigma\,\{\varphi[n/m']\}\vdash e{:}\,\mathbf{nat}(n)}{\Gamma;\,y{:}\,\sigma\,\{\varphi[n/m']\}\vdash y:=e\rhd y{:}\,\mathbf{nat}(n)\{\varphi[n/m']\}}\qquad\dfrac{}{\Gamma;\,(y{:}\,\mathbf{nat}(m')\{\varphi\})[n/m']\vdash\varepsilon\rhd\exists m'.y{:}\,\mathbf{nat}(m')\{\varphi\}}}{\Gamma;\,y{:}\,\sigma\,\{\varphi[n/m']\}\vdash y:=e;\rhd\exists m'.y{:}\,\mathbf{nat}(m')\{\varphi\}}$$

Using the same technique, it is possible to derive a variant of rule (H.VAR), called (H.VAR'), for declaring local mutable variables initialized with natural numbers:

$$\frac{\Gamma;\Omega\{\varphi[n/m']\}\vdash e{:}\,\mathbf{nat}(n)\qquad\Gamma;\Omega,\,y{:}\,\mathbf{nat}(m')\{\varphi\}\vdash s\rhd\exists\vec{\jmath}\,.\Omega'\{\chi\}\qquad y\notin\Omega'}{\Gamma;\Omega\{\varphi[n/m']\}\vdash\mathbf{var}\ y:=e;\ s\ \rhd\exists\vec{\jmath}\,.\Omega'\{\chi\}}$$

(H.TERM)
$$\frac{\Gamma, \Omega, x\colon \varphi \vdash_{\mathbf{FD}} t\colon \psi}{\Gamma; \Omega\{\varphi\} \vdash t[?/x]\colon \psi}$$

(H.IDENT)
$$\frac{x\colon \tau \in \Gamma; \Omega}{\Gamma; \Omega\{\varphi\} \vdash x\colon \tau}$$

(H.PROC)
$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\top}\{\varphi\} \vdash s \rhd \exists \vec{j}\,.\vec{z} : \vec{\tau}\,\{\psi\}}{\Gamma; \Omega\{\gamma\} \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\ )\{s\}\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,\{\varphi\}; \exists \vec{j}\,\mathbf{out}\ \vec{\tau}\,\{\psi\})} \qquad \vec{\imath} \notin \mathcal{FV}(\Gamma)$$

(H.SUBST-I)
$$\frac{\Gamma; \Omega\{\varphi\} \vdash e\colon \tau[n/i] \qquad \Gamma, \Omega, \varphi \vdash n = m}{\Gamma; \Omega\{\varphi\} \vdash e\colon \tau[m/i]}$$

(H.SUBST-II)
$$\frac{\Gamma; \Omega\{\varphi\} \vdash s \rhd \exists \vec{j}\,.(\Omega'\{\psi\})[n/i] \qquad \Gamma, \Omega, \varphi \vdash n = m}{\Gamma; \Omega\{\varphi\} \vdash s \rhd \exists \vec{j}\,.(\Omega'\{\psi\})[m/i]}$$

(H.EMPTY)
$$\frac{}{\Gamma; \Omega, (\vec{z} : \vec{\tau}\,\{\varphi\})[\vec{m}/\vec{\imath}\,] \vdash \varepsilon \rhd \exists \vec{\imath}\,.\vec{z} : \vec{\tau}\,\{\varphi\}}$$

(H.SEQ)
$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\varphi\} \qquad \Gamma; \Omega, \vec{x} : \vec{\tau}\,\{\varphi\} \vdash s \rhd \exists \vec{j}\,.\Omega'\{\chi\}}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c;\ \ s\ \rhd \exists \vec{j}\,.\Omega'\{\chi\}} \qquad \vec{\imath} \notin \mathcal{FV}(\Gamma, \Omega, \exists \vec{j}\,.\Omega'\{\chi\})$$

(H.CST)
$$\frac{\Gamma; \Omega\{\gamma\} \vdash e\colon \tau \qquad \Gamma, y\colon \tau; \Omega\{\gamma\} \vdash s \rhd \exists \vec{j}\,.\Omega'\{\chi\}}{\Gamma; \Omega\{\gamma\} \vdash \mathbf{cst}\ \ y = e;\ \ s\ \rhd \exists \vec{j}\,.\Omega'\{\chi\}}$$

(H.VAR)
$$\frac{\Gamma; \Omega\{\gamma\} \vdash e\colon \tau \qquad \Gamma; \Omega, y\colon \tau\{\gamma\} \vdash s \rhd \exists \vec{j}\,.\Omega'\{\chi\} \qquad y \notin \Omega'}{\Gamma; \Omega\{\gamma\} \vdash \mathbf{var}\ \ y := e;\ \ s\ \rhd \exists \vec{j}\,.\Omega'\{\chi\}}$$

(H.ASSIGN)
$$\frac{\Gamma; \Omega, y\colon \sigma\{\varphi\} \vdash e\colon \tau}{\Gamma; \Omega, y\colon \sigma\{\varphi\} \vdash y := e \rhd \Omega, y\colon \tau\{\varphi\}}$$

(H.INC)
$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n)\{\varphi\} \vdash \mathbf{inc}(y) \rhd \Omega, y\colon \mathbf{nat}(\mathbf{s}(n))\{\varphi\}}$$

(H.DEC)
$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n)\{\varphi\} \vdash \mathbf{dec}(y) \rhd \Omega, y\colon \mathbf{nat}(\mathbf{p}(n))\{\varphi\}}$$

(H.BLOCK)
$$\frac{\Gamma; \vec{x} : \vec{\tau}\,\{\varphi\} \vdash s \rhd \exists \vec{j}\,.\Omega'\{\chi\}}{\Gamma; \Omega, \vec{x} : \vec{\tau}\,\{\varphi\} \vdash \{s\}_{\vec{x}} \rhd \exists \vec{j}\,.\Omega'\{\chi\}}$$

(H.FOR)
$$\frac{\Gamma; \Omega, (\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{0}/i] \vdash e\colon \mathbf{nat}(n) \qquad \Gamma, y\colon \mathbf{nat}(i); \vec{x} : \vec{\sigma}\,\{\varphi\} \vdash s \rhd \exists \vec{j}\,.(\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{s}(i)/i]}{\Gamma; \Omega, (\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{0}/i] \vdash \mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s\}_{\vec{x}} \rhd \exists \vec{j}\,.(\vec{x} : \vec{\sigma}\,\{\varphi\})[n/i]} \qquad i, \vec{j} \notin \mathcal{FV}(\Gamma)$$

(H.CALL)
$$\frac{\Gamma; \Omega\{\varphi[\vec{m}/\vec{\imath}\,]\} \vdash p\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,\{\varphi\}; \exists \vec{j}\ \mathbf{out}\ \vec{\tau}\,\{\chi\}) \quad \Gamma; \Omega\{\varphi[\vec{m}/\vec{\imath}\,]\} \vdash \vec{e} : \vec{\sigma}\,[\vec{m}/\vec{\imath}\,] \quad \vec{r} \subseteq \Omega}{\Gamma; \Omega\{\varphi[\vec{m}/\vec{\imath}\,]\} \vdash p(\vec{e}\,; \vec{r}\,) \rhd \exists \vec{j}\,.(\vec{r} : \vec{\tau}\,\{\chi\})[\vec{m}/\vec{\imath}\,]}$$

(H.LABEL)
$$\frac{\Gamma, k\colon \mathbf{label}\ \exists \vec{j}\,(\vec{\sigma}\,\{\chi\}); \vec{z} : \vec{\tau}\,\{\gamma\} \vdash s \rhd \exists \vec{j}\,.\vec{z} : \vec{\sigma}\,\{\chi\}}{\Gamma; \Omega, \vec{z} : \vec{\tau}\,\{\gamma\} \vdash k\colon \{s\}_{\vec{z}} \rhd \exists \vec{j}\,.\vec{z} : \vec{\sigma}\,\{\chi\}}$$

(H.JUMP)
$$\frac{\Gamma; \Omega\{\psi[\vec{m}/\vec{j}\,]\} \vdash k\colon \mathbf{label}\ \exists \vec{j}\,(\vec{\sigma}\,\{\psi\}) \qquad \Gamma; \Omega\{\psi[\vec{m}/\vec{j}\,]\} \vdash \vec{e} : \vec{\sigma}\,[\vec{m}/\vec{j}\,] \qquad \vec{z} \subseteq \Omega}{\Gamma; \Omega\{\psi[\vec{m}/\vec{j}\,]\} \vdash \mathbf{jump}(k, \vec{e}\,)_{\vec{z}} \rhd \vec{z} : \vec{\tau}\,\{\chi\}}$$

Figure 4.1. Hoare Dependent Type System

4.1 Soundness

In order to prove that the rules in figure 4.1 are indeed admissible, we first need to formalize how irrelevant parts of the program are "hidden". For that purpose, we shall rely on the fact that different typing derivations may correspond to the same computational content. We thus introduce the following equivalence relation which captures exactly this notion.

Definition 4.5.

- *Given two expressions e and e' such that $\Gamma;\Omega\vdash e{:}\,\sigma$ and $\Gamma';\Omega'\vdash e'{:}\,\sigma'$, we say that e and e' are equivalent, and we write $e\simeq e'$, if $\kappa(\Gamma^\star,\Omega^\star\vdash e^\star{:}\,\sigma^\star)=\kappa(\Gamma'^\star,\Omega'^\star\vdash e'^\star{:}\,\sigma'^\star)$.*
- *Given two sequences s and s' such that $\Gamma;\Omega\vdash s\rhd\exists\vec{\imath}\,.\vec{x}:\vec{\sigma}$ and $\Gamma';\Omega'\vdash s'\rhd\exists\vec{\jmath}\,.\vec{y}:\vec{\tau}$, we say that s and s' are equivalent, and we write $s\simeq s'$, if $\kappa(\Gamma^\star,\Omega^\star\vdash(s)^\star_{\vec{x}}{:}\,\exists\vec{\imath}\,.\vec{\sigma}^\star)=\kappa(\Gamma'^\star,\Omega'^\star\vdash(s')^\star_{\vec{y}}{:}\,\exists\vec{\jmath}\,.\vec{\tau}^\star)$.*

We are now ready to define formally the set of valid Hoare judgments:

Definition 4.6. (Validity of Hoare judgments).

- *For any expression e and any irrelevant formula φ, we say that a judgment $\Gamma;\Omega\{\varphi\}\vdash e{:}\,\sigma$ is valid if there is an expression $e'\simeq e$ such that $\Gamma;\Omega, \mathit{assert}{:}\,\varphi\vdash e'{:}\,\sigma$ is derivable in $\mathbf{ID}^c$.*
- *For any sequence s and any irrelevant formulas φ, χ, we say that a judgment $\Gamma;\Omega\{\varphi\}\vdash s\rhd\exists\vec{\jmath}\,.\Omega'\{\chi\}$ is valid if there is a sequence $s'\simeq s$ such that $\Gamma;\Omega, \mathit{assert}{:}\,\varphi\vdash s'\rhd\exists\vec{\jmath}\,.\Omega', \mathit{assert}{:}\,\chi$ is derivable in $\mathbf{ID}^c$.*

Notation 4.7. *We shall also write $c\simeq c'$ as an abbreviation for $c;\varepsilon\simeq c';\varepsilon$.*

Proposition 4.8. *All rules from the Hoare Dependent Type System (figure 4.1) are admissible.*

Proof. By construction, most of these rules are simply instances of rules from $\mathbf{ID}^c$. For instance, (H.ASSIGN), (H.LABEL) and (H.JUMP) correspond to the following instances:

- (H.ASSIGN)

$$\frac{\Gamma;\Omega, y{:}\,\sigma, \mathit{assert}{:}\,\varphi\vdash e{:}\,\tau}{\Gamma;\Omega, y{:}\,\sigma, \mathit{assert}{:}\,\varphi\vdash y:=e\rhd\Omega, y{:}\,\tau, \mathit{assert}{:}\,\varphi}$$

- (H.LABEL)

$$\frac{\Gamma, k{:}\,\mathbf{label}\ \exists\vec{\jmath}\,(\vec{\sigma}, \chi);\vec{z}:\vec{\tau}, \mathit{assert}{:}\,\gamma\vdash s\rhd\exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}, \mathit{assert}{:}\,\chi}{\Gamma;\Omega,\vec{z}:\vec{\tau}, \mathit{assert}{:}\,\gamma\vdash k{:}\,\{s\}_{\vec{z},\mathit{assert}}\rhd\exists\vec{\jmath}\,.\vec{z}:\vec{\sigma}, \mathit{assert}{:}\,\chi}$$

- (H.JUMP)

$$\frac{\Gamma;\Omega, \mathit{assert}{:}\,\psi[\vec{m}/\vec{\jmath}\,]\vdash k{:}\,\mathbf{label}\ \exists\vec{\jmath}\,(\vec{\sigma}, \psi)\qquad \Gamma;\Omega, \mathit{assert}{:}\,\psi[\vec{m}/\vec{\jmath}\,]\vdash\vec{e}:\vec{\sigma}\,[\vec{m}/\vec{\jmath}\,]\qquad \vec{z}\subseteq\Omega}{\Gamma;\Omega, \mathit{assert}{:}\,\psi[\vec{m}/\vec{\jmath}\,]\vdash\mathbf{jump}(k,\vec{e}, \mathit{assert})_{\vec{z}, \mathit{assert}}\rhd\vec{z}:\vec{\tau}, \mathit{assert}{:}\,\chi}$$

Let us now show how to deal with the rules for procedures:

- (H.CALL). We take $c'=p'(\vec{e}\,', \mathit{assert};\vec{r}, \mathit{assert})$ where $p'\simeq p$ and $\vec{e}\,'\simeq\vec{e}$ are given by the induction hypothesis, and we check that the following rule is derivable in $\mathbf{ID}^c$:

$$\frac{\Gamma;\Omega, \mathit{assert}{:}\,\varphi[\vec{m}/\vec{\imath}\,]\vdash p'{:}\,\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma}, \varphi;\exists\vec{\jmath}\ \mathbf{out}\ \vec{\tau}, \chi)\quad \Gamma;\Omega, \mathit{assert}{:}\,\varphi[\vec{m}/\vec{\imath}\,]\vdash\vec{e}\,'{:}\,\vec{\sigma}\,[\vec{m}/\vec{\imath}\,]\quad \vec{r}\subseteq\Omega}{\Gamma;\Omega, \mathit{assert}{:}\,\varphi[\vec{m}/\vec{\imath}\,]\vdash p'(\vec{e}\,', \mathit{assert};\vec{r}, \mathit{assert})\rhd\exists\vec{\jmath}\,.(\vec{r}:\vec{\tau}, \mathit{assert}{:}\,\chi)[\vec{m}/\vec{\imath}\,]}$$

  Moreover, we have $c'\simeq p(\vec{e};\vec{r})$ since φ and χ are irrelevant.

- (H.PROC). We take $p'=\mathbf{proc}\ (\mathbf{in}\ \vec{y}, \mathit{assert}';\mathbf{out}\ \vec{z}, \mathit{assert})\,\{\mathit{assert}:=\mathit{assert}';s'\}$ where $s'\simeq s$ is given by the induction hypothesis and we check that the following rule is derivable in $\mathbf{ID}^c$:

$$\frac{\vec{z}\neq\emptyset\qquad \Gamma,\vec{y}:\vec{\sigma};\vec{z}:\vec{\top}, \mathit{assert}{:}\,\varphi\vdash s'\rhd\exists\vec{\jmath}\,.\vec{z}:\vec{\tau}, \mathit{assert}{:}\,\chi}{\Gamma;\Omega, \mathit{assert}{:}\,\gamma\vdash\mathbf{proc}\ (\mathbf{in}\ \vec{y}, \mathit{assert}';\mathbf{out}\ \vec{z}, \mathit{assert})\,\{\mathit{assert}:=\mathit{assert}';s'\}{:}\,\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma}, \varphi;\exists\vec{\jmath}\ \mathbf{out}\ \vec{\tau}, \chi)}$$

  Moreover, we have $p'\simeq\mathbf{proc}\ (\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z})\,\{s\}$ since φ and χ are irrelevant.

□

Remark 4.9. The following rule, called (H.IF), is admissible for the conditional described in Remark 3.8 (where $n\neq 0$ is an abbreviation for $\exists m(n=\mathbf{s}(m))$):

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\tau}\,\{\varphi\}\vdash e{:}\,\mathbf{nat}(n)\quad \Gamma;\vec{x}:\vec{\tau}\,\{\varphi\wedge n\neq 0\}\vdash s_1\rhd\exists\vec{\imath}\,.\vec{x}:\vec{\sigma}\,\{\psi\}\quad \Gamma;\vec{x}:\vec{\tau}\,\{\varphi\wedge n=0\}\vdash s_2\rhd\exists\vec{\imath}\,.\vec{x}:\vec{\sigma}\,\{\psi\}}{\Gamma;\Omega,\vec{x}:\vec{\tau}\,\{\varphi\}\vdash\mathbf{if}\ e\ \mathbf{then}\ \{s_1\}_{\vec{x}}\ \mathbf{else}\ \{s_2\}_{\vec{x}}\rhd\exists\vec{\imath}\,.\vec{x}:\vec{\sigma}\,\{\psi\}}$$

Using the same technique as for the conditional, it is possible to derive an improved rule for the loop, called (H.FOR'), which gives access to the hypothesis $i < n$ when checking the body (where $i < n$ is an abbreviation for $\exists m(n = i + \mathbf{s}(m))$):

$$\frac{\Gamma; \Omega, (\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{0}/i] \vdash e \colon \mathbf{nat}(n) \qquad \Gamma, y \colon \mathbf{nat}(i); \vec{x} : \vec{\sigma}\,\{\varphi \wedge i < n\} \vdash s \rhd \exists \vec{\jmath}\,.(\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{s}(i)/i]}{\Gamma; \Omega, (\vec{x} : \vec{\sigma}\,\{\varphi\})[\mathbf{0}/i] \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}} \rhd \exists \vec{\jmath}\,.(\vec{x} : \vec{\sigma}\,\{\varphi\})[n/i]}$$

4.2 Consequence rule

In this section, we consider the admissibility of the consequence rule and, in particular, how post-condition weakening is related to the famous frame problem [10].

4.2.1 Pre-condition strengthening

Proposition 4.10. (Pre-condition strengthening). *The following rule is admissible:*

$$\frac{\Gamma, \Omega, \varphi \vdash \psi \qquad \Gamma; \Omega\{\psi\} \vdash s \rhd \exists \vec{\jmath}\,.\Omega'\{\chi\}}{\Gamma; \Omega\{\varphi\} \vdash s\ \rhd \exists \vec{\jmath}\,.\Omega'\{\chi\}}$$

Proof. We take $s'' = \mathit{assert} := ?; s'$ where $s' \simeq s$ is given by the induction hypothesis and we check that the following rule is derivable in $\mathbf{ID}^c$:

$$\frac{\Gamma, \Omega, \varphi \vdash ?\colon \psi \qquad \Gamma; \Omega, \mathit{assert}\colon \psi \vdash s' \rhd \exists \vec{\jmath}\,.\Omega', \mathit{assert}\colon \chi}{\Gamma; \Omega, \mathit{assert}\colon \varphi \vdash \mathit{assert} := ?; s' \rhd \exists \vec{\jmath}\,.\Omega', \mathit{assert}\colon \chi}$$

Moreover, we have $s'' \simeq s$ by Remark 3.13 since ψ is irrelevant. □

4.2.2 Post-condition weakening and the frame problem

A reader familiar with Floyd-Hoare logics for procedures would certainly have guessed that our rules from figure 4.1 cannot deal properly with the so-called frame problem (defined in [10] as *the inability to express that a procedure changes only those things it has to*). In our framework, all assertions remain true forever since they can only mention logical variables (see Remark 4.3), so it should not be a problem. Actually, our frame problem comes from the fact that our encoding relies on only *one* global assertion simulated in state-passing style. In order to remember the pre-condition which holds before a command (for instance a procedure call), we would need to generalize rule (H.SEQ) as follows:

Proposition 4.11. (Frame rule). *The following rule is admissible:*

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\sigma}\,'\{\varphi\} \qquad \Gamma; \Omega, \vec{x} : \vec{\sigma}\,'\{\gamma \wedge \varphi\} \vdash s \rhd \exists \vec{\jmath}\,.\Omega'\{\chi\}}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c;\ \ s\ \rhd \exists \vec{\jmath}\,.\Omega'\{\chi\}}$$

Proof. We take $s'' = \mathbf{cst}\ \mathit{assert}' = \mathit{assert}; c'; \mathit{assert} := (\mathit{assert}', \mathit{assert}); s'$ where $c' \simeq c$ and $s' \simeq s$ are given by the induction hypothesis and we check that the following rule is derivable in $\mathbf{ID}^c$:

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,, \mathit{assert}\colon \gamma \vdash c' \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\sigma}\,', \mathit{assert}\colon \varphi \qquad \Gamma; \Omega, \vec{x} : \vec{\sigma}\,', \mathit{assert}\colon \gamma \wedge \varphi \vdash s' \rhd \exists \vec{\jmath}\,.\Omega', \mathit{assert}\colon \chi}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,, \mathit{assert}\colon \gamma \vdash \mathbf{cst}\ \ \mathit{assert}' = \mathit{assert}; c'; \mathit{assert} := (\mathit{assert}', \mathit{assert}); s'\ \rhd \exists \vec{\jmath}\,.\Omega', \mathit{assert}\colon \chi}$$

Moreover, we have $s'' \simeq\ c; s$ since γ and φ are irrelevant.

□

Remark 4.12. In the proof above, we could have equally taken $s'' = \mathbf{cst}\ \mathit{assert}' = ?; c'; \mathit{assert} := ?; s'$ since proof-terms of irrelevant formulas are not required.

Example 4.13. Let Γ be the following environment: $p\colon \mathbf{proc}\ \forall i(\mathbf{in}\ \mathbf{nat}(i)\{\}; \exists j\ \mathbf{out}\ \mathbf{nat}(j)\{j = i\})$ and let us now consider a procedure call $p(0, y)$, where y is some mutable variable. Assuming that we also have another mutable variable x whose value is 1, we would like to prove the following judgment:

$$\Gamma; x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(m)\{n = 1\} \vdash p(0; y) \rhd \exists j.x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(j)\{n = 1 \wedge j = 0\}$$

Rule (H.CALL) gives us:

$$\mathcal{D} = \frac{\Gamma; x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(m)\{n=1\} \vdash p\colon \mathbf{proc}\ \forall i(\mathbf{in}\ \mathbf{nat}(i)\{\}; \exists j\ \mathbf{out}\ \mathbf{nat}(j)\{j=i\}) \quad \Gamma; \Omega\{n=1\} \vdash 0\colon \mathbf{nat}(0)}{\Gamma; x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(m)\{n=1\} \vdash p(0; \vec{r}\,) \rhd \exists j.y\colon \mathbf{nat}(j)\{j=0\}}$$

In the above conclusion, we forgot the fact that $n = 1$ in the post-condition. In order to get this information back, we need to apply the frame rule:

$$\frac{\mathcal{D} \qquad \Gamma; x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(j)\{n=1 \wedge j=0\} \vdash \varepsilon \rhd \exists j.x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(j)\{n=1 \wedge j=0\}}{\Gamma; x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(m)\{n=1\} \vdash p(0; y); \rhd \exists j.x\colon \mathbf{nat}(n), y\colon \mathbf{nat}(j)\{n=1 \wedge j=0\}}$$

Remark 4.14. An alternative rule, which is clearly equivalent to the frame rule, is the following rule (called *rule of inheritance* in VDM [47]):

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\varphi\}}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\gamma \wedge \varphi\}}$$

An advantage of this rule is that it can also be formulated as a generalized post-condition weakening rule:

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\varphi\} \qquad \Gamma, \Omega \vdash \gamma \Rightarrow \varphi \Rightarrow \psi}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\gamma\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\psi\}}$$

Combining the pre-condition strengthening rule with the above post-condition weakening rule, we obtain thus the following consequence rule which is similar to the rule of consequence from VDM (which is known to be stronger than the Hoare rule of consequence [49]):

Proposition 4.15. (Consequence rule)*. The following rule, called* (H.CONS)*, is admissible:*

$$\frac{\Gamma, \Omega \vdash \varphi' \Rightarrow \varphi \qquad \Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\varphi\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\psi\} \qquad \Gamma, \Omega \vdash \varphi \Rightarrow \psi \Rightarrow \psi'}{\Gamma; \Omega, \vec{x} : \vec{\sigma}\,\{\varphi'\} \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}\,\{\psi'\}}$$

Definition 4.16. *We call* $\mathbf{HD}^c$ *the Hoare Dependent Type System (figure 4.1) extended with the consequence rule* (H.CONS)*.*

4.3 Completeness

In this section we show that system $\mathbf{HD}^c$ is complete: any valid Hoare judgment is derivable (up to equivalence). This result is a corollary of a stronger result. Indeed, $\mathbf{HD}^c$ is actually complete for $\mathbf{ID}^c$ in the following sense: any command typable in $\mathbf{ID}^c$ is equivalent to a command typable in $\mathbf{HD}^c$ of the corresponding annotated specification. To be more specific, we first need to explain how we map types and judgments of $\mathbf{ID}^c$ onto types and judgments of $\mathbf{HD}^c$.

Notation 4.17. *If* $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$*, we write* $\hat{\sigma}$ *for the formula* $\sigma_1 \wedge \ldots \wedge \sigma_n$ *and by extension if* γ *is an environment* $x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n$*, we also write* $\hat{\gamma}$ *for the formula* $\sigma_1 \wedge \ldots \wedge \sigma_n$*.*

Definition 4.18. *Given an imperative dependent type* σ*, the corresponding annotated type* $\sigma^\vartriangle$ *of* $\mathbf{HD}^c$ *is defined inductively as follows:*

- $(\psi)^\vartriangle = \psi$
- $(\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,, \vec{\varphi}; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,, \vec{\chi}\,))^\vartriangle = \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}^{\,\vartriangle}\{\hat{\varphi}\}; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}^{\,\vartriangle}\{\hat{\chi}\,\})$
  where $\vec{\varphi}$*,* $\vec{\chi}$ *are irrelevant formulas and* $\vec{\sigma}$*,* $\vec{\tau}$ *are not.*
- $(\mathbf{label}\ \exists \vec{\jmath}\,(\vec{\sigma}\,, \vec{\varphi}\,))^\vartriangle = \mathbf{label}\ \exists \vec{\jmath}\,(\vec{\sigma}^{\,\vartriangle}\{\hat{\varphi}\,\})$
  where $\vec{\varphi}$ *are irrelevant formulas and* $\vec{\sigma}$ *are not.*

Notation 4.19. *If* $\Gamma = x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n$*, we write* $\bar{\Gamma}$ *for the environment* $x_1\colon \sigma_1^\vartriangle, \ldots, x_n\colon \sigma_n^\vartriangle$*.*

Theorem 4.20. (completeness of $\mathbf{HD}^c$ with respect to $\mathbf{ID}^c$).

- *For any sequence* s *such that* $\Gamma, \gamma; \Omega, \omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega', \chi$ *is derivable in* $\mathbf{ID}^c$*, where* γ*,* ω *and* χ *denote the irrelevant formulas of the sequent, there is a sequence* $s' \simeq s$ *such that* $\bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \hat{\omega}\,\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}\,\}$ *is derivable in* $\mathbf{HD}^c$*.*

- *For any expression e such that $\Gamma, \gamma; \Omega, \omega \vdash e\colon \varphi$ is derivable in $\mathbf{ID}^c$, where γ, ω and φ denote the irrelevant formulas of the sequent, $\bar{\Gamma}, \bar{\Omega}, \hat{\gamma} \wedge \hat{\omega} \vdash \varphi$ is derivable in $\mathbf{HD}^c$.*
- *For any expression e such that $\Gamma, \gamma; \Omega, \omega \vdash e\colon \sigma$ is derivable in $\mathbf{ID}^c$, where γ and ω denote the irrelevant formulas of the sequent, there is an expression $e' \simeq e$ such that $\bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \hat{\omega}\} \vdash e\colon \sigma^{\vartriangle}$ is derivable in $\mathbf{HD}^c$.*

Proof. The three statements of the theorem are proved simultaneously by mutual induction on the typing derivation of sequences and expressions. We consider only the rules which are not translated directly into instances of rules $\mathbf{HD}^c$ and we assume that meta-variables γ, ω and χ denote the irrelevant formulas of sequents.

- (T.VAR). We consider only the case where the type of y is irrelevant:

  $$\frac{\Gamma, \gamma; \Omega, \omega \vdash e\colon \varphi \qquad \Gamma, \gamma, y\colon \varphi; \Omega, \omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega', \chi}{\Gamma, \gamma; \Omega, \omega \vdash \mathbf{var}\ y\colon = e;\ s \rhd \exists \vec{\jmath}\,.\Omega', \chi}$$

  Indeed, we check that the following rule is derivable from the consequence rule, where $s' \simeq s$ is given by the induction hypothesis:

  $$\frac{\bar{\Gamma}, \bar{\Omega}, \hat{\gamma} \wedge \hat{\omega} \vdash \varphi \qquad \bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \varphi \wedge \hat{\omega}\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}\}}{\bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \hat{\omega}\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}\}}$$

  Moreover, we have $s' \simeq \mathbf{var}\ y\colon = e; s$ since φ is irrelevant.
- (T.CST). Similar to (T.VAR).
- (T.ASSIGN). We consider only the case where the type of y is irrelevant:

  $$\frac{\Gamma, \gamma; \Omega, \omega \vdash e\colon \varphi \qquad \Gamma, \gamma, y\colon \varphi; \Omega, \omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega', \chi}{\Gamma, \gamma; \Omega, y\colon \psi, \omega \vdash y\colon = e;\ s \rhd \exists \vec{\jmath}\,.\Omega', \chi}$$

  Indeed, we check that the following rule is derivable from the consequence rule, where $s' \simeq s$ is given by the induction hypothesis:

  $$\frac{\bar{\Gamma}, \bar{\Omega}, \hat{\gamma} \wedge \hat{\omega} \vdash \varphi \qquad \bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \varphi \wedge \hat{\omega}\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}\}}{\bar{\Gamma}; \bar{\Omega}\{\hat{\gamma} \wedge \psi \wedge \hat{\omega}\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}\}}$$

  Moreover, we have $s' \simeq y\colon = e; s$ since φ is irrelevant.
- (T.SEQ).

  $$\frac{\Gamma, \gamma; \Omega, \vec{x} : \vec{\sigma}, \omega, \chi \vdash c \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}, \omega' \qquad \Gamma, \gamma; \Omega, \vec{x} : \vec{\tau}, \omega', \chi \vdash s \rhd \exists \vec{\jmath}\,.\Omega', \chi'}{\Gamma, \gamma; \Omega, \vec{x} : \vec{\sigma}, \omega, \chi \vdash c; s \rhd \exists \vec{\jmath}\,.\Omega', \chi'}$$

  Indeed, we check that the following rule is a variant of the frame rule and is thus derivable from the consequence rule, where $c' \simeq c$ and $s' \simeq s$ are given by the induction hypothesis:

  $$\frac{\bar{\Gamma}, \bar{\Omega}, \vec{x} : \vec{\sigma}^{\vartriangle}\{\hat{\gamma} \wedge \hat{\omega} \wedge \hat{\chi}\} \vdash c' \rhd \exists \vec{\imath}\,.\vec{x} : \vec{\tau}^{\vartriangle}\{\hat{\omega}'\} \qquad \bar{\Gamma}; \bar{\Omega}, \vec{x} : \vec{\tau}^{\vartriangle}\{\hat{\gamma} \wedge \hat{\omega}' \wedge \hat{\chi}\} \vdash s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}'\}}{\bar{\Gamma}; \bar{\Omega}, \vec{x} : \vec{\sigma}^{\vartriangle}\{\hat{\gamma} \wedge \hat{\omega} \wedge \hat{\chi}\} \vdash c'; s' \rhd \exists \vec{\jmath}\,.\bar{\Omega}'\{\hat{\chi}'\}}$$

  Moreover, we clearly have $c'; s' \simeq c; s$.
- (T.LABEL).

  $$\frac{\Gamma, \gamma, k\colon \mathbf{label}\ \exists \vec{\jmath}\,(\vec{\sigma}, \chi'); \vec{z} : \vec{\tau}, \chi \vdash s \rhd \vec{z} : \vec{\sigma}, \chi'}{\Gamma, \gamma; \Omega, \vec{z} : \vec{\tau}, \omega, \chi \vdash k\colon \{s\}_{\vec{z}} \rhd \vec{z} : \vec{\sigma}, \chi'}$$

  Indeed, we check that the following rule is derivable from the consequence rule, where $s' \simeq s$ is given by the induction hypothesis:

  $$\frac{\bar{\Gamma}, k\colon \mathbf{label}\ \exists \vec{\jmath}\,(\vec{\sigma}^{\vartriangle}\{\hat{\chi}'\}); \vec{z} : \vec{\tau}^{\vartriangle}\{\hat{\gamma} \wedge \hat{\chi}\} \vdash s' \rhd \vec{z} : \vec{\sigma}^{\vartriangle}\{\hat{\chi}'\}}{\bar{\Gamma}; \bar{\Omega}, \vec{z} : \vec{\tau}^{\vartriangle}\{\hat{\gamma} \wedge \hat{\omega} \wedge \hat{\chi}\} \vdash k\colon \{s'\}_{\vec{z}} \rhd \vec{z} : \vec{\sigma}^{\vartriangle}\{\hat{\chi}'\}}$$

  Moreover, we clearly have $k\colon \{s'\}_{\vec{z}} \simeq k\colon \{s\}_{\vec{z}}$.

- (T.JUMP).

$$\frac{\Gamma,\gamma;\Omega,\omega\vdash k\colon \mathbf{label}\ \exists\vec{\jmath}\,(\vec{\sigma},\vec{\chi})\quad \Gamma,\gamma;\Omega,\omega\vdash\vec{e}:\vec{\sigma}[\vec{m}/\vec{\jmath}]\quad \Gamma,\gamma;\Omega,\omega\vdash\vec{u}:\vec{\chi}[\vec{m}/\vec{\jmath}]\quad \vec{z}\subseteq\Omega\quad \vec{z}\,'\subseteq\omega}{\Gamma,\gamma;\Omega,\omega\vdash\mathbf{jump}(k,\vec{e},\vec{u})_{\vec{z},\vec{z}\,'}\rhd\vec{z}:\vec{\tau},\omega'}$$

Indeed, we check that the following rule is derivable from the consequence rule, where $\vec{e}\,'\simeq\vec{e}$ is given by the induction hypothesis:

$$\frac{\bar{\Gamma};\bar{\Omega}\{\hat{\gamma}\wedge\hat{\omega}\}\vdash k\colon\mathbf{label}\ \exists\vec{\jmath}\,(\vec{\sigma}^{\vartriangle}\{\hat{\chi}\})\quad \bar{\Gamma};\bar{\Omega}\{\hat{\gamma}\wedge\hat{\omega}\}\vdash\vec{e}\,'\colon\vec{\sigma}^{\vartriangle}[\vec{m}/\vec{\jmath}]\quad \bar{\Gamma},\bar{\Omega}\{\hat{\gamma}\wedge\hat{\omega}\}\vdash\hat{\chi}[\vec{m}/\vec{\jmath}]\quad \vec{z}\subseteq\bar{\Omega}}{\bar{\Gamma};\bar{\Omega}\{\hat{\gamma}\wedge\hat{\omega}\}\vdash\mathbf{jump}(k,\vec{e}\,')_{\vec{z}}\rhd\vec{z}:\vec{\tau}^{\vartriangle}\{\hat{\omega}'\}}$$

Moreover, we have $\mathbf{jump}(k,\vec{e}\,')_{\vec{z}}\simeq\mathbf{jump}(k,\vec{e},\vec{u})_{\vec{z},\vec{z}\,'}$ since $\vec{\chi}$ is irrelevant.

□

Corollary 4.21. (completeness of $\mathbf{HD}^c$).

- *For any sequence s, if the Hoare judgment* $\Gamma;\Omega\{\varphi\}\vdash s\rhd\exists\vec{\jmath}.\Omega'\{\chi\}$ *is valid then there is some* $s'\simeq s$ *such that* $\Gamma;\Omega\{\varphi\}\vdash s'\rhd\exists\vec{\jmath}.\Omega'\{\chi\}$ *is derivable in* $\mathbf{HD}^c$.
- *For any expression e, if the Hoare judgment* $\Gamma;\Omega\{\varphi\}\vdash e\colon\sigma$ *is valid then there is some* $e'\simeq e$ *such that* $\Gamma;\Omega\{\varphi\}\vdash e'\colon\sigma$ *is derivable in* $\mathbf{HD}^c$.

Proof. If the Hoare judgment $\Gamma;\Omega\{\varphi\}\vdash s\rhd\exists\vec{\jmath}.\Omega'\{\chi\}$ is valid then, by definition, there is some $s'\simeq s$ such that $\Gamma;\Omega$, *assert*: $\varphi\vdash s'\rhd\exists\vec{\jmath}.\Omega'$, *assert*: χ is derivable in $\mathbf{ID}^c$, and by Theorem 4.20, there is some $s''\simeq s'$ such that $\Gamma;\Omega\{\varphi\}\vdash s''\rhd\exists\vec{\jmath}.\Omega'\{\chi\}$ is derivable in $\mathbf{HD}^c$. The case for expressions is similar. □

4.4 Example

As a final example, we consider the following classical example of an imperative program with jumps. Given a function $f\colon\mathbb{N}\to\mathbb{N}$, we would like to compute the product of its n first values. This product $p(n,f)$ can be defined inductively by the following equations:

$$p(0,f) = 1 \tag{4.1}$$

$$p(\mathbf{s}(n),f) = f(n)\times p(n,f) \tag{4.2}$$

In the following anonymous procedure, the loop variable i iterates over the range 0, ..., $n-1$ and, as an optimization, the loop exits and the procedure returns with 0 whenever $f(i)=0$ (where the conditional command was defined in Remark 3.8).

```
proc (in $F, N$; out $M$) {
    $M := 1$;                                  ] $s_1$
    $K$: {                                     ] $s_2$
        for $I := 0$ until $N$ {               ] $s_3$
            var $R := F(I)$;                   ] $s_4$
            if $R$ then {                      ] $s_5$
                $M := R*M$;          ]$s_6$
            }$_M$ else {
                jump$(K,0)_M$;       ]$s_7$
            }$_M$;
        }$_M$;
    }$_M$;
};
```

Let us call this procedure P and let us prove that the following specification is derivable:

$$\vdash P\colon\mathbf{proc}\ \forall f,n\ (\mathbf{in}\ \forall x(\mathbf{nat}(x)\to\mathbf{nat}(f(x))),\mathbf{nat}(n)\{\};\exists m.\mathbf{out}\ \mathbf{nat}(m)\{m=p(n,f)\})$$

Let us also define the following formula:

- $\varphi\equiv(m=p(i,f))\wedge i<n\wedge r=f(i)$

and the following environments:

- $\Gamma_0 \equiv F\colon \forall x(\mathbf{nat}(x) \to \mathbf{nat}(f(x))), N\colon \mathbf{nat}(n)$
- $\Gamma_1 \equiv \Gamma_0, K\colon \mathbf{label}\ \exists m'(\mathbf{nat}(m')\{m' = p(n, f)\})$
- $\Gamma_2 \equiv \Gamma_1, I\colon \mathbf{nat}(i)$
- $\Omega_0 \equiv M\colon \top$
- $\Omega_1 \equiv M\colon \mathbf{nat}(m)$
- $\Omega_2 \equiv \Omega_1, R\colon \mathbf{nat}(r)$

The typing derivation is now built inductively as follows:

- Sequence s_6

$$\mathcal{D}_0 = \dfrac{\dfrac{}{\Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash R\colon \mathbf{nat}(r)}(\textsc{ident}) \qquad \dfrac{}{\Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash M\colon \mathbf{nat}(m)}(\textsc{ident})}{\Gamma_2; \Omega_2, \varphi \wedge r \neq 0 \vdash (R, M)\colon \mathbf{nat}(r) \wedge \mathbf{nat}(m)}(\textsc{tuple})$$

$$\mathcal{D}_1 = \dfrac{*\colon \forall n, m(\mathbf{nat}(n) \wedge \mathbf{nat}(m) \Rightarrow \mathbf{nat}(n \times m)) \qquad \mathcal{D}_0}{\Gamma_2; \Omega_2, \varphi \wedge r \neq 0 \vdash R * M\colon \mathbf{nat}(r \times m)}(\textsc{app})$$

$$\mathcal{D}_2 = \dfrac{\mathcal{D}_1 \qquad \Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash r = f(i)}{\Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash R * M\colon \mathbf{nat}(f(i) \times m)}(\textsc{subst})$$

$$\mathcal{D}_3 = \dfrac{\mathcal{D}_2 \qquad \Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash m = p(i, f)}{\Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash R * M\colon \mathbf{nat}(f(i) \times p(i, f))}(\textsc{subst})$$

$$\mathcal{D}_4 = \dfrac{\dfrac{\dfrac{\mathcal{D}_3 \qquad \Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash p(\mathbf{s}(i), f) = f(i) \times p(i, f)}{\Gamma_2, \Omega_2, \varphi \wedge r \neq 0 \vdash R * M\colon \mathbf{nat}(p(\mathbf{s}(i), f))}(\textsc{subst})}{\Gamma_2; \Omega_2\{\varphi \wedge r \neq 0\} \vdash R * M\colon \mathbf{nat}(p(\mathbf{s}(i), f))}(\textsc{h.term})}{\Gamma_2; \Omega_2\{\varphi \wedge r \neq 0\} \vdash M := R{*}M \rhd \exists m'.M\colon \mathbf{nat}(m')\{m' = p(\mathbf{s}(i), f)\}}(\textsc{h.assign'})$$

- Sequence s_7

$$\mathcal{D}_5 = \dfrac{}{\Gamma_2; \Omega_2\{0 = p(n, f)\} \vdash K\colon \mathbf{label}\ \ \exists m'(\mathbf{nat}(m')\{m' = p(n, f)\})}(\textsc{h.ident})$$

$$\mathcal{D}_6 = \dfrac{\mathcal{D}_5 \qquad \dfrac{\Gamma_2, \Omega_2, 0 = p(n, f) \vdash 0\colon \mathbf{nat}(0)}{\Gamma_2; \Omega_2\{0 = p(n, f)\} \vdash 0\colon \mathbf{nat}(0)}(\textsc{h.term})}{\Gamma_2; \Omega_2\{0 = p(n, f)\} \vdash \mathbf{jump}(K, 0)_M; \rhd \exists m'.M\colon \mathbf{nat}(m')\{m' = p(\mathbf{s}(i), f)\}}(\textsc{h.jump})$$

- Sequence s_5

$$\mathcal{D}_7 = \dfrac{\mathcal{D}_4 \qquad \dfrac{\Gamma_2, \Omega_2 \vdash (\varphi \wedge r = 0) \Rightarrow (0 = p(n, f)) \quad \mathcal{D}_6}{\Gamma_2; \Omega_2\{\varphi \wedge r = 0\} \vdash \mathbf{jump}(K, 0)_M; \rhd \exists m'.M\colon \mathbf{nat}(m')\{m' = p(\mathbf{s}(i), f)\}}(\textsc{h.cons})}{\Gamma_2; \Omega_2\{\varphi\} \vdash \mathbf{if}\ R\ \mathbf{then}\ \{s_6\}\ \mathbf{else}\ \{s_7\}; \rhd \exists m'.M\colon \mathbf{nat}(m')\{m' = p(\mathbf{s}(i), f)\}}(\textsc{h.if})$$

- Sequence s_4

$$\mathcal{D}_8 = \dfrac{}{\Gamma_2, \Omega_1, m = p(i, f) \wedge i < n \vdash F\colon \forall x(\mathbf{nat}(x) \to \mathbf{nat}(f(x)))}(\textsc{ident})$$

$$\mathcal{D}_9 = \dfrac{\dfrac{\dfrac{\mathcal{D}_8 \qquad \dfrac{}{\Gamma_2, \Omega_1, m = p(i, f) \wedge i < n \vdash I\colon \mathbf{nat}(i)}(\textsc{ident})}{\Gamma_2, \Omega_1, m = p(i, f) \wedge i < n \vdash F(I)\colon \mathbf{nat}(f(i))}(\textsc{app})}{\Gamma_2; \Omega_1, \{m = p(i, f) \wedge i < n\} \vdash F(I)\colon \mathbf{nat}(f(i))}(\textsc{h.term}) \qquad \mathcal{D}_7}{\Gamma_2; \Omega_1\{m = p(i, f) \wedge i < n\} \vdash \mathbf{var}\ R := F(I); s_5; \rhd \exists m'.M\colon \mathbf{nat}(m')\{m' = p(\mathbf{s}(i), f)\}}(\textsc{h.var'})$$

- Sequence s_3

$$\mathcal{D}_{10} \;=\; \dfrac{\dfrac{}{\Gamma_1; \Omega_1\{m = p(0, f)\} \vdash N \colon \mathbf{nat}(n)}(\textsc{h.ident}) \qquad \mathcal{D}_9}{\Gamma_1; \Omega_1\{m = p(0, f)\} \vdash \mathbf{for}\ I := 0\ \mathbf{until}\ N\ \ \{s_4\}_M; \rhd \exists m'.M \colon \mathbf{nat}(m')\{m' = p(n, f)\}}(\textsc{h.for'})$$

- Sequence s_2

$$\mathcal{D}_{11} \;=\; \dfrac{\mathcal{D}_{10}}{\Gamma_0; \Omega_1\{m = p(0, f)\} \vdash K \colon \{s_3\}; \rhd \exists m'.M \colon \mathbf{nat}(m')\{m' = p(n, f)\}}(\textsc{h.label})$$

- Finally

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_0, \Omega_0 \vdash 1 \colon \mathbf{nat}(1) \quad \Gamma_0, \Omega_0 \vdash 1 = p(0, f)}{\Gamma_0, \Omega_0 \vdash 1 \colon \mathbf{nat}(p(0, f))}(\textsc{subst})}{\Gamma_0; \Omega_0\{\} \vdash 1 \colon \mathbf{nat}(p(0, f))}(\textsc{h.term})}{\Gamma_0; \Omega_0\{\} \vdash M := 1 \rhd \exists m.M \colon \mathbf{nat}(m)\{m = p(0, f)\}}(\textsc{h.assign'}) \qquad \mathcal{D}_{11}}{F \colon \forall x(\mathbf{nat}(x) \to \mathbf{nat}(f(x))), N \colon \mathbf{nat}(n); M \colon \top\{\} \vdash s_1 \rhd \exists m'.M \colon \mathbf{nat}(m')\{m' = p(n, f)\}}(\textsc{h.seq})}{\{\} \vdash P \colon \mathbf{proc}\ \forall f, n\ (\mathbf{in}\ \forall x(\mathbf{nat}(x) \to \mathbf{nat}(f(x))), \mathbf{nat}(n)\{\}; \exists m'\ \mathbf{out}\ \mathbf{nat}(m')\{m' = p(n, f)\})}(\textsc{h.proc})$$

In the above derivation, we assumed the existence of a binary function $* \colon \forall n, m(\mathbf{nat}(n) \wedge \mathbf{nat}(m) \Rightarrow \mathbf{nat}(n \times m))$ (such a term is easily definable in **FD**). We also relied on the derived rules ($\textsc{h.assign'}$) and ($\textsc{h.var'}$) from Remark 4.4, the improved rule ($\textsc{h.for'}$) and the rule for the conditional ($\textsc{h.if}$) from Remark 4.9. Finally, note that the only non-trivial proof-obligation is $(m = p(i, f) \wedge i < n \wedge r = f(i) \wedge r = 0) \Rightarrow p(n, f) = 0$ (which is used in derivation $\mathcal{D}_7$).

Remark 4.22. We have formally specified $\mathbf{ID}^c$, **FD** and the translation * in Twelf [73] and, thanks to Twelf's logic programming engine, those specifications are executable. Note that in order to obtain executable type-checkers from their specification, proof-terms need to be fully annotated (they contain all the information from the derivation). Type checking is then easily defined as a syntax directed function (implemented as a relation with the proper modes in Twelf). Moreover, we have mechanically checked the correctness of a few examples (the interested reader is referred to [20] for more details).

The formalization of $\mathbf{HD}^c$ in Twelf in currently in progress. However, from a practical standpoint, the goal is now to generate proof-obligations (from the occurrences of the consequence rule) which can be fed to some external tool (solver or automated theorem prover) for checking their validity (this idea is standard when implementing Floyd-Hoare logics [35]). The Twelf implementation shall thus be used only to check the syntax-directed part of the correctness proof.

5 Conclusion and future work

We have presented an imperative language with higher-order procedural variables and non-local jumps together with its dependent type system. Its semantics is defined by translation into a functional dependent type theory. The imperative type system is carefully designed in such way as to decorate intuitionistic proofs with functional terms and classical proofs with imperative commands. As usual with intuitionistic type systems, irrelevant proof-terms can thus be erased. We then rely on this property and on a simple state-passing style transform to derive a Hoare Dependent Type System (where assertions are supposed to be irrelevant).

As we have already mentioned, the target of our translation is close to the logic defined by Thielecke in [88] where the double-negation is abstracted as a modality. In fact, the system described in [88] also includes a delimited control operator. Although this needs to be investigated further, it seems that in our framework, delimited control provides a way to extend the language with so-called block-expressions (that is, the possibility to embed imperatives sequences into functional terms).

The semantics of this imperative type theory is defined by translation into a classical functional type theory. Although this translation is sufficient to derive the correctness of imperative programs, and it successfully accounts for the fact that mutable variables take on different values during computation, it does not capture the idea that an assignment destructively alters the contents of the store. This approach could however be refined to model properly in-place updates using (as in [70]) a linear λ-calculus as the target functional system.

Since we restrict ourselves to language constructs which correspond to proof-terms, we have to be careful when considering extensions. However, extensions for which the formulas-as-types interpretation is well-understood in the functional setting are good candidates for inclusion (on the proviso that a reasonable imperative syntax exists). This clearly concerns *polymorphism* (universal types), *abstract data types* (existential types) and *inductive data types* (lists, trees, ...). Finally, a *while*-loop may also be considered as a realizer for well-founded induction (this idea actually goes back to Nuprl [15]).

Finally, we are also interested in the computational content of intermediate logics. For instance, the first author has shown in [18] that a formulas-as-types interpretation of subtractive logic (also called bi-intuitionistic logic) exhibits an unusual form of functional coroutines. In contrast, we expect an imperative version of the deduction system from [18] to be closer to the more conventional Floyd-Hoare logic for imperative coroutines described in [12].

Appendix A Functional simple type system FS

The functional simple type system is summarized in Figure A.1.

$$\begin{array}{rc}
(\textsc{ident}) & \dfrac{x\colon\alpha\in\Sigma}{\Sigma\vdash x\colon\alpha} \\[2ex]
(\textsc{zero}) & \Sigma\vdash 0\colon\mathbf{nat} \\[2ex]
(\textsc{succ}) & \dfrac{\Sigma\vdash t\colon\mathbf{nat}}{\Sigma\vdash S(t)\colon\mathbf{nat}} \\[2ex]
(\textsc{pred}) & \dfrac{\Sigma\vdash t\colon\mathbf{nat}}{\Sigma\vdash \mathbf{pred}(t):\mathbf{nat}} \\[2ex]
(\textsc{tuple}) & \dfrac{\Sigma\vdash t_1\colon\alpha_1\quad ...\quad \Sigma\vdash t_n\colon\alpha_n}{\Sigma\vdash (t_1,...,t_n)\colon\alpha_1\times...\times\alpha_n} \\[2ex]
(\textsc{unit}) & \dfrac{}{\Sigma\vdash ()\colon\mathbf{unit}} \\[2ex]
(\textsc{let}) & \dfrac{\Sigma,x_1\colon\alpha_1,...,x_n\colon\alpha_n\vdash t\colon\alpha\qquad \Sigma\vdash u\colon\alpha_1\times...\times\alpha_n}{\Sigma\vdash \mathbf{let}\ (x_1,...,x_n)=u\ \mathbf{in}\ t\colon\alpha} \\[2ex]
(\textsc{abs}) & \dfrac{\Sigma,x\colon\alpha_1\vdash t\colon\alpha_2}{\Sigma\vdash\lambda x.t\ :\alpha_1\to\alpha_2} \\[2ex]
(\textsc{app}) & \dfrac{\Sigma\vdash t\colon\alpha_1\to\alpha_2\quad \Sigma\vdash u\colon\alpha_1}{\Sigma\vdash t\ u\colon\alpha_2} \\[2ex]
(\textsc{rec}) & \dfrac{\Sigma\vdash t_1\colon\mathbf{nat}\quad \Sigma\vdash t_2\colon\alpha\quad \Sigma,x\colon\mathbf{nat},y\colon\alpha\vdash t_3\colon\alpha}{\Sigma\vdash\mathbf{rec}(t_1,t_2,\lambda x.\lambda y.t_3):\alpha}
\end{array}$$

Figure A.1. Functional simple type system **FS**

Appendix B Deriving the conditional command

In this appendix, we show that the following typing rule is derivable in $\mathbf{ID}^c$:

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash e\colon\mathbf{nat}(n)\qquad \Gamma,h\colon n\neq 0;\vec{x}:\vec{\tau}\vdash s_1\rhd\exists\vec{\iota}\,.\vec{x}:\vec{\sigma}\qquad \Gamma,h\colon n=0;\vec{x}:\vec{\tau}\vdash s_2\rhd\exists\vec{\iota}\,.\vec{x}:\vec{\sigma}}{\Gamma;\Omega,\vec{x}:\vec{\tau}\vdash\mathbf{if}\ e\ \mathbf{then}\ \{s_1\}_{\vec{x}}\ \mathbf{else}\ \{s_2\}_{\vec{x}}\rhd\exists\vec{\iota}\,.\vec{x}:\vec{\sigma}}$$

Let us first annotate the definition of the conditional command from Remark 4.9 as follows:

$$\begin{array}{l}
\mathbf{if}\ e\ \mathbf{then}\ \{s_1\}_{\vec{x}}\ \mathbf{else}\ \{s_2\}_{\vec{x}}\ \equiv \\
\quad \mathbf{cst}\ v=e; \\
\quad \{ \\
\qquad \left.\begin{array}{l}
\mathbf{cst}\ \vec{x}\,'=\vec{x}; \\
\mathbf{proc}\ q_2(\mathbf{in}\ h;\ \mathbf{out}\ \vec{x}:=\vec{x}\,')\ \{s_2\}_{\vec{x}}; \\
\left.\begin{array}{l}
\mathbf{var}\ p:=q_2; \\
\left.\begin{array}{l}
\mathbf{for}\ y:=\mathbf{0}\ \mathbf{until}\ v\ \{ \\
\quad \left.\begin{array}{l}
\mathbf{proc}\ q_1(\mathbf{in}\ h';\ \mathbf{out}\ \vec{x}:=\vec{x}\,')\ \{\mathbf{cst}\ h=h';s_1\}_{\vec{x}}; \\
p:=q_1;
\end{array}\right] s_6 \\
\}_p;
\end{array}\right] c_5 \\
p(();\vec{x});
\end{array}\right] s_4
\end{array}\right] s_3 \\
\quad \}_{\vec{x}}
\end{array}$$

Let us also define the following environments:

- $\Gamma_1 = \Gamma, v\colon \mathbf{nat}(n)$
- $\Gamma_2 = \Gamma_1, \vec{x}\,'\colon \vec{\tau}\,, q_2\colon \mathbf{proc}\ (\mathbf{in}\ n = 0;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,)$
- $\Gamma_3 = \Gamma_2, y\colon \mathbf{nat}(j)$
- $\Gamma_4 = \Gamma_3, q_1\colon \mathbf{proc}\ (\mathbf{in}\ n = \mathbf{s}(j);\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,)$
- $\Omega_1 = \vec{x} : \vec{\tau}\,, p\colon \mathbf{proc}\ (\mathbf{in}\ n = 0;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,)$
- $\Omega_2 = p\colon \mathbf{proc}\ (\mathbf{in}\ n = j;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,)$
- $\Omega_2' = p\colon \mathbf{proc}\ (\mathbf{in}\ n = \mathbf{s}(j);\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,)$

The typing derivation is now built inductively as follows:

- Sequence s_6

$$\mathcal{D}_7 = \dfrac{\dfrac{\dfrac{\Gamma_3, h'\colon n = \mathbf{s}(j), \vec{x} : \vec{\tau} \vdash h'\colon n = \mathbf{s}(j)}{\Gamma_3, h'\colon n = \mathbf{s}(j), \vec{x} : \vec{\tau} \vdash h'\colon n \neq 0}(\textsc{tuple})}{\Gamma_3, h'\colon n = \mathbf{s}(j); \vec{x} : \vec{\tau} \vdash h'\colon n \neq 0}(\textsc{t.term}) \qquad \Gamma_3, h\colon n \neq 0; \vec{x} : \vec{\tau} \vdash s_1 \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}{\Gamma_3, h'\colon n = \mathbf{s}(j); \vec{x} : \vec{\tau} \vdash \mathbf{cst}\ h = h'; s_1 \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.cst})$$

$$\mathcal{D}_6 = \dfrac{\mathcal{D}_7 \qquad \Gamma_4; \Omega_2 \vdash\ p := q_1 \rhd \Omega_2'}{\Gamma_3; \Omega_2 \vdash \mathbf{proc}\ q_1(\mathbf{in}\ h';\ \mathbf{out}\ \vec{x} := \vec{x}\,')\ \{\mathbf{cst}\ h = h'; s_1\}_{\vec{x}}; p := q_1 \rhd p\colon \Omega_2'}(\textsc{t.proc'})$$

- Command c_5

$$\mathcal{D}_5 = \dfrac{\dfrac{}{\Gamma_2; \Omega_1 \vdash v\colon \mathbf{nat}(n)}(\textsc{t.ident}) \qquad \mathcal{D}_6}{\Gamma_2; \Omega_1 \vdash \mathbf{for}\ y := \mathbf{0}\ \mathbf{until}\ v\ \{s_6\}_p \rhd p\colon \mathbf{proc}\ (\mathbf{in}\ n = n;\ \exists\vec{\imath}\,.\mathbf{out}\ \vec{\sigma}\,)}(\textsc{t.for})$$

- Sequence s_4

$$\mathcal{D}_4 \ = \ \dfrac{\dfrac{\mathcal{D}_5 \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_2, p\colon \mathbf{proc}\ (\mathbf{in}\ \top;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,), \vec{x} : \vec{\tau} \vdash ()\colon \top}(\textsc{equal})}{\Gamma_2; p\colon \mathbf{proc}\ (\mathbf{in}\ \top;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,), \vec{x} : \vec{\tau} \vdash ()\colon \top}(\textsc{t.term})}{\Gamma_2; p\colon \mathbf{proc}\ (\mathbf{in}\ \top;\ \exists\vec{\imath}\,\mathbf{out}\ \vec{\sigma}\,), \vec{x} : \vec{\tau} \vdash p((); \vec{x}\,); \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.call})}{\Gamma_2; \Omega_1 \vdash c_5;\ p((); \vec{x}\,); \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.seq})}{\Gamma_2; \vec{x} : \vec{\tau} \vdash \mathbf{var}\ p := q_2;\ c_5;\ p((); \vec{x}\,); \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.var})$$

- Sequence s_3

$$\mathcal{D}_3 = \dfrac{\dfrac{\Gamma_1, \vec{x}\,'\colon \vec{\tau}\,, h\colon n = 0; \vec{x} : \vec{\tau} \vdash s_2 \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma} \qquad \mathcal{D}_4}{\Gamma_1; \vec{x}\,'\colon \vec{\tau} \vdash \mathbf{proc}\ q_2(\mathbf{in}\ h;\ \mathbf{out}\ \vec{x} := \vec{x}\,')\ \{s_2\}_{\vec{x}}; s_4 \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.proc'})}{\Gamma_1; \vec{x} : \vec{\tau} \vdash \mathbf{cst}\ \vec{x}\,' = \vec{x}\,; \mathbf{proc}\ q_2(\mathbf{in}\ h;\ \mathbf{out}\ \vec{x} := \vec{x}\,')\ \{s_2\}_{\vec{x}}; s_4 \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.cst})$$

- Finally

$$\dfrac{\Gamma; \Omega, \vec{x} : \vec{\tau} \vdash e\colon \mathbf{nat}(n) \qquad \dfrac{\mathcal{D}_3}{\Gamma, v\colon \mathbf{nat}(n); \Omega, \vec{x} : \vec{\tau} \vdash \{s_3\}_{\vec{x}} \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.block})}{\Gamma; \Omega, \vec{x} : \vec{\tau} \vdash \mathbf{cst}\ v = e; \{s_3\}_{\vec{x}} \rhd \exists\vec{\imath}\,.\vec{x} : \vec{\sigma}}(\textsc{t.cst})$$

Bibliography

[1] K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[2] P. Audebaud and E. Zucca. Deriving Proof Rules from Continuation Semantics. *Formal Aspects of Computing*, 11(4):426–447, 1999.

[3] F. Barbanera and S. Berardi. Extracting Constructive Content from Classical Logic via Control-like Reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag, 1994.

[4] J. Barnes. *High integrity software: the SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[5] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational Types from a Logical Perspective. *J. Funct. Program*, 8(2):177–193, 1998.

[6] M. Berger. Program Logics for Sequential Higher-Order Control. In *Proceedings of the Third IPM International Conference, FSEN 2009*, volume 5961 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2010.

[7] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114(1-3):3–25, 2002.

[8] U. Berger and H. Schwichtenberg. Program Development by Proof Transformation. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *Series F: Computer and Systems Sciences*, pages 1–45. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20 – August 1, 1993, Springer-Verlag, 1995.

[9] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In Daniel Leivant, editor, *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. Springer Berlin / Heidelberg, 1995.

[10] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, 2002.

[11] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axioms. *Journal of the ACM*, 26(1), January 1979.

[12] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2(1):50–63, 1973.

[13] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224, 1972.

[14] L. Colson and D. Fredholm. System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315, 1998.

[15] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Saski, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[16] T. Coquand. Computational Content of Classical Logic. In *Semantics and Logics of Computation*, pages 470–517. Cambridge University Press, 1996.

[17] P. Cousot. Methods and Logics for Proving Programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–994. Elsevier Science Publishers B.V. (North Holland), 1990.

[18] T. Crolard. A Formulæ-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14(4):529–570, 2004.

[19] T. Crolard. Certification de programmes impératifs d'ordre supérieur avec mécanismes de contrôle. Habilitation Thesis. LACL, Université Paris-Est, 2010.

[20] T. Crolard. A Formally Specified Program Logic for Higher-Order Procedural Variables and non-local Jumps. Technical Report TR-LACL-2011-5, Université Paris-Est, 2011. Also available as `arXiv:1112.1848`.

[21] T. Crolard and E. Polonowski. A program logic for higher-order procedural variables and non-local jumps. Technical Report TR-LACL-2011-4, Université Paris-Est, 2011. Chapter 3 of the first author's Habilitation thesis, also available as `arXiv:1112.1554`.

[22] T. Crolard, E. Polonowski, and P. Valarcher. Extending the Loop Language with Higher-Order Procedural Variables. *Special issue of ACM TOCL on Implicit Computational Complexity*, 10(4):1–37, 2009.

[23] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[24] W. Damm and B. Josko. A sound and relatively complete Hoare-logic for a language with higher type procedures. *Acta Informatica*, 20(1):59–101, 1983.

[25] O. Danvy. Back to direct style. In *ESOP'92*, pages 130–150. Springer, 1992.

[26] O. Danvy and J. L. Lawall. Back to direct style II: first-class continuations. *SIGPLAN Lisp Pointers*, V(1):299–310, 1992.

[27] P. de Groote. A simple calculus of exception handling. In *Second International Conference on Typed Lambda Calculi and Applications*, LNCS, pages 201–215, Edinburgh, United Kingdom, 1995.

[28] J. E. Donahue. Locations Considered Unnecessary. *Acta Inf.*, 8:221–242, 1977.

[29] M. Felleisen. *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages*. PhD thesis, Indiana University, Indianapolis, IN, USA, 1987.

[30] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 401–414, New York, NY, USA, June 2006. ACM Press.

[31] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[32] H. Friedman. Classically and intuitionistically provably recursive functions. *Higher Set Theory*, pages 21–27, 1978.

[33] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7. Cambridge Tracts in Theorical Comp. Sci., 1989.

[34] K. Gödel. Über eine bisher noch nicht benützteerweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[35] M. J. C. Gordon. Specification and verification I. Lecture notes, University of Cambridge, Computer Laboratory, 1988.

[36] T. G. Griffin. A formulæ-as-types notion of control. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Langages*, pages 47–58, 1990.

[37] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[38] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA, 1994. ACM.

[39] M. C. Henson. Information Loss in the Programming Logic TK. In *Programming Concepts and Methods*, pages 509–545. Elsevier, 1990.

[40] H. Herbelin. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In Pawel Urzyczyn, editor, *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005.

[41] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[42] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, volume 188, pages 102–116. Springer, 1971.

[43] K. Honda, M. Berger, and N. Yoshida. Descriptive and Relative Completeness of Logics for Higher-Order Functions. *Automata, Languages and Programming*, pages 360–371, 2006.

[44] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. *Symposium on Logic in Computer Science, LICS*, 5:270–279, 2005.

[45] W. A. Howard. The Formulæ-as-types Notion of Constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculs and Formalism*, pages 479–490. Academic Press, 1969.

[46] K. Jensen. Connection between Dijkstra's predicate transformers and denotational continuation semantics. Technical report, Technical Report DAIMI PB-86, Computer Science Dept., Aarhus Univ., 1978.

[47] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1990.

[48] R. Kelsey, W. Clinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[49] T. Kleymann. Hoare Logic and Auxiliary Variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[50] J.-L. Krivine. Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78, 1994.

[51] J.-L. Krivine and M. Parigot. Programming with proofs. *J. Inf. Process. Cybern. EIK*, 26(3):149–167, 1990.

[52] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

[53] P. J. Landin. A correspondence between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM*, 8(3):158–167, 1965.

[54] P. J. Landin. A Generalization of Jumps and Labels. Technical report, UNIVAC Systems Programming Research, 1965.

[55] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.

[56] D. Leivant. Contracting proofs to programs. In Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, 1990.

[57] D. Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 114(1-3):117–153, 2002.

[58] C. Lewington. Towards constructive program derivation in VDM. In Kesav Nori and C. Veni Madhavan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lecture Notes in Computer Science*, pages 115–132. Springer Berlin / Heidelberg, 1990.

[59] Y. Makarov. Practical Program Extraction from Classical Proofs. *Electronic Notes in Theoretical Computer Science*, 155:521 – 542, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[60] Y. Makarov. Simplifying Programs Extracted from Classical Proofs. In Stephen van Bakel and Stefano Berardi, editors, *Workshop on Classical logic and Computation*, July 2006.

[61] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proc. ACM Nat. Meeting*, 1976.

[62] E. Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[63] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[64] C. R. Murthy. *Extracting Constructive Content from Classical proofs*. PhD thesis, Cornell University, Department of Computer Science, 1990.

[65] C. R. Murthy. An evaluation semantics for classical proofs. In *Proc. 6th Annual IEEE Symp. on Logic in Computer Science*, pages 96–107, 1991.

[66] C. R. Murthy. Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science, 1991.

[67] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73. ACM New York, NY, USA, 2006.

[68] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*. Citeseer, 2008.

[69] M. J. O'Donnell. A critique of the foundations of Hoare style programming logics. *Commun. ACM*, 25(12):927–935, 1982. http://doi.acm.org/10.1145/358728.358748.

[70] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, 2000.

[71] M. Parigot. Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*, 1993.

[72] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(04):511–540, 2001.

[73] F. Pfenning and C. Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, London, UK, 1999. Springer-Verlag.

[74] G. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *TCS*, 1(2):125–159, 1975.

[75] I. Poernomo. Proofs-as-Imperative-Programs: Application to Synthesis of Contracts. *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003; Revised Papers*, 2003.

[76] I. Poernomo and J. N. Crossley. The Curry-Howard isomorphism adapted for imperative program synthesis and reasoning. *Proceedings of the 7th and 8th Asian Logic Conferences. World Scientific*, 2003.

[77] N. J. Rehof and M. H. Sørensen. The λ_Δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag, 1994.

[78] B. Reus and T. Streicher. About Hoare Logics for Higher-Order Store. *Automata, Languages and Programming*, pages 1337–1348, 2005.

[79] J. C. Reynolds. On the relation between direct and continuation semantics. *Automata, languages and programming*, pages 141–156, 1974.

[80] D. A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers Dubuque, IA, USA, 1986.

[81] D. Sitaram and M. Felleisen. Reasoning with continuations II: full abstraction for models of control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 161–175, New York, NY, USA, 1990. ACM.

[82] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[83] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[84] W. Swierstra. A Hoare logic for the state monad. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009.

[85] G. Tan and A. W. Appel. A Compositional Logic for Control Flow. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2006.

[86] R. D. Tennent and J. K. Tobin. Continuations in Possible-World Semantics. *Theor. Comput. Sci.*, 85(2):283–303, 1991.

[87] H. Thielecke. An Introduction to Landin's "A Generalization of Jumps and Labels". *Higher-Order and Symbolic Computation*, 11(2):117–123, 1998.

[88] H. Thielecke. Control Effects as a Modality. *Journal of Functional Programming*, 19:17–26, 2008.

[89] A. S. Troelstra. Realizability. In *Handbook of proof theory*, volume 137, chapter VI, pages 407–473. Elsevier, 1998.

[90] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.

Table of contents