Extending the LOOP Language with Higher-Order Procedural Variables

T. CROLARD, E. POLONOWSKI and P. VALARCHER
University of Paris XII

We extend Meyer and Ritchie's LOOP language with higher-order procedures and procedural variables and we show that the resulting programming language (called LOOP$^\omega$) is a natural imperative counterpart of Gödel System T. The argument is two-fold:

(1) we define a translation of the LOOP$^\omega$ language into System T and we prove that this translation actually provides a lock-step simulation,
(2) using a converse translation, we show that LOOP$^\omega$ is expressive enough to encode any term of System T.

Moreover, we define the "iteration rank" of a LOOP$^\omega$ program, which corresponds to the classical notion of "recursion rank" in System T, and we show that both translations preserve ranks. Two applications of these results in the area of implicit complexity are described.

Categories and Subject Descriptors: F.3.2 [**Theory of Computation**]: Logics and Meanings of Programs—*Semantics of Programming Languages*; F.1.3 [**Theory of Computation**]: Computation by Abstract Devices—*Complexity Measures and Classes*

Additional Key Words and Phrases: Loop language, Gödel System T, higher-order procedures, procedural variables.

1. INTRODUCTION

Primitive recursive functionals of finite type are representable as terms of the simply typed λ-calculus equipped with a type of natural numbers and primitive recursion at all types. This calculus (called System T) was first introduced by Gödel in its proof-theoretic study of Peano arithmetic (see its Dialectica paper on the consistency of arithmetic [Gödel 1958], reproduced with English translation in [Gödel 1990]). Moreover, System T is well suited to give a formal semantics to constructs found in (higher-order) programming languages [Girard et al. 1989].

The LOOP language [Meyer and Ritchie 1976] is a core imperative language in which programs consist only of assignments, sequences, and bounded loops. Meyer and Ritchie's proved in particular that LOOP programs compute exactly the class of primitive recursive functions. The LOOP language has since been widely studied in the literature (see for instance the textbooks [Davis and Weyuker 1983] and [Calude 1988]).

Authors address: Laboratoire d'Algorithmique, Complexité et Logique – Université Paris XII – 61, avenue du Général de Gaulle – 94010 Créteil Cedex, France.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2006 ACM 1529-3785/06/1000-0001 $5.00

In this paper, we introduce a statically typed extension of the LOOP language (called LOOP$^\omega$) with higher-order procedures and procedural variables and we argue that this programming language is a natural imperative counterpart of Gödel System T. The argument is two-fold:

(1) we define a translation of the LOOP$^\omega$ language into System T and we prove that this translation actually provides a lock-step simulation,
(2) using a converse translation, we show that LOOP$^\omega$ is expressive enough to encode any term of System T.

The first result is the main contribution of this paper: we prove that LOOP$^\omega$ is not only extensionally equivalent to System T, but also intensionally. In order to develop this point, we need to give some details about the operational semantics of LOOP$^\omega$. Actually, the semantics of LOOP$^\omega$ is presented first as a natural semantics (also called "big-step" semantics) in the style of [Kahn 1987]. Then we define a transition semantics (also called "small-step" semantics [Plotkin 1981]), tailored of the lock-step simulation, which refines the natural semantics. Now the simulation theorem states that each evaluation step of a LOOP$^\omega$ program (according to the transition semantics) is mapped to one reduction step of the transformed program in System T (using a call-by-value strategy). As a corollary of this theorem, we obtain that all LOOP$^\omega$ programs are always terminating.

Note that LOOP$^\omega$ is a genuine imperative language with first-class procedures (true closures) and mutable procedural variables (aka function pointers). For instance, any LOOP$^\omega$ can easily be written using C# syntax (where anonymous first-class procedures are called delegates [ISO 2003]) and then compiled with a C# compiler. However, it is a "pure" imperative language in the following sense: its type system forbids side-effects, parameter-induced aliasing and back-patching. Side-effects and aliasing are controversial features which are known to complicate the semantics, and back-patching is a technique which exploits procedural variables to define arbitrary recursive procedures (whereas LOOP$^\omega$ is a "total" programming language as advocated in [Turner 2004]). As a consequence of these choices, both semantics presented in this paper are simple but rather unusual for imperative languages: they are both location-free semantics (see [Donahue 1977], [Plotkin 1981] and [Felleisen and Friedman 1987]) and they rely crucially on the distinction between mutable and read-only (immutable) variables.

The second result is obtained by defining a converse translation (from System T into LOOP$^\omega$) and proving that the composition the two translations yields the identity in System T (up to $\beta\pi$-equivalence). Since both translations are syntax directed (compositional) and type-preserving, we can be even more specific about the correspondence between a functional program and its imperative counterpart. Recall that a major result of [Gödel 1958] states that functions on the natural numbers that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see for instance the survey [Avigad and Feferman 1998] or the textbook [Schütte 1967]). More precisely, that there is a syntactic hierarchy of fragments T_n of System T such that the class of functions representable in T_n is identical to the class of functions provably recursive in the fragment of Peano arithmetic where induction is restricted to Σ_{n+1} formulas. In particular, T_0 corresponds to the class of primitive recursive functions.

We define thus a similar hierarchy of fragments $\textsc{Loop}^n$ of $\textsc{Loop}^\omega$ and we show that both translations relate programs of $\textsc{Loop}^n$ and terms of T_n. As a corollary, we obtain that the functions representable in a language with higher-order procedures but without procedural variables (which is a sub-language of $\textsc{Loop}^0$) are primitive recursive. This corollary generalizes thus previous results presented in [Crolard et al. 2006] where Meyer and Ritchie's Loop language was translated into T_0. On the other hand, Ackermann function which is known not to be primitive recursive is representable in T_1 and thus also in $\textsc{Loop}^1$. As far as we know, $\textsc{Loop}^\omega$ is the first total imperative language allowing to program Ackermann function.

Applications. A first application of these results is extensional: we derive a new characterization of the class of Csillag-Kalmar elementary functions (the class $\mathcal{E}_3$ in Grzegorczyk hierarchy). In [Beckmann and Weiermann 2000], such a characterization is based on a syntactic restriction on terms of a variant of Gödel System T. As a corollary of various properties of our two translations we provide an imperative counterpart of this restriction. In particular, we obtain that any $\textsc{Loop}^\omega$ program in which any bound of loop is a read-only input variable is elementary.

A second application is intensional and is related to the so-called *minimum problem*. In [Colson and Fredholm 1998], the authors proved that in call-by-value System T, any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. As a consequence of this property, there is no term that computes the minimum of two natural numbers n and m in time $O(min(n,m))$. As a corollary of the lock-step simulation we obtain a similar negative result for $\textsc{Loop}^\omega$ programs.

Related works. There has been a lot of work on the semantics of Algol (major contributions are collected in [O'Hearn and Tennent 1997]) and this work has influenced the design of most modern programming languages. For instance, the $\textsc{Loop}^\omega$ language is very close in spirit to the language partially described in [Reynolds 1978] whose purpose was in particular to show that it is possible to avoid aliasing while retaining Algol-like higher-order procedures. More generally, our work can also be seen as providing a denotational semantics for an imperative language using a λ-calculus and follows thus to the Scott-Strachey tradition [Stoy 1977].

Our translation is also reminiscent of the state monad [Moggi 1991; Wadler 1990] used to encode a mutable state (or references) in a pure functional language. This encoding is usually global and changes the type of all terms. However, a local encoding can be obtained if an effect system [Gifford and Lucassen 1986] is used for the source language (instead of a conventional type system). A simulation based on such an encoding is described in [Wadler 1998].

Similarly, in [Filliâtre 2003], the author defines a monadic translation of simply-typed functional programs with references (annotated with Floyd-Hoare assertions) into a type theory. Their translation provides the core of the proof technique developed afterward for imperative programs [Filliâtre and Marché 2004]. Although similar to the one described in this paper, their translation is only considered in the context of program verification (no simulation is defined).

Plan of the paper. In Section 2, we present our variant of Gödel System T. In Section 3, we describe the LOOP$^\omega$ language (syntax, type system and semantics). In Section 4, we show how to translate LOOP$^\omega$ programs into functional programs and we prove the simulation theorem. In Section 5, we define the converse translation, and we use it to prove that LOOP$^\omega$ is as expressive as System T. Eventually, in Section 6, we describe the two applications.

2. GÖDEL SYSTEM T

Gödel System T is usually defined as the simply typed λ-calculus extended with a type of natural numbers and with primitive recursion at all types. We consider in this paper a variant of System T with product types (tuples and n-ary functions) and a constant-time predecessor operation. Moreover, since we are mainly interested here in the call-by-value evaluation strategy, we formulate this system directly as a contextual semantics (a set of reduction rules together with an inductive definition of evaluation contexts). As usual, we consider terms up to α-conversion and the set $FId(t)$ of free identifiers of a term t is defined in the standard way. The rewriting system is summarized in Figure 1, where $t[v_1/x_1, \ldots, v_n/x_n]$ denotes the

(types)

$$\begin{array}{rcl} \tau & ::= & int \\ & | & unit \\ & | & \tau_1 \to \tau_2 \\ & | & \tau_1 \times \ldots \times \tau_n \end{array}$$

(values)

$$\begin{array}{rcl} v & ::= & x \\ & | & 0 \\ & | & S(v) \\ & | & (v_1, \ldots, v_n) \\ & | & \lambda(x_1, \ldots, x_n).t \end{array}$$

(terms)

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

(contexts)

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

(evaluation rules)

$$\begin{array}{rcl} C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\ C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\ C[\mathbf{rec}(0, v_2, v_3)] & \rightsquigarrow & C[v_2] \\ C[\mathbf{rec}(S(v_1), v_2, v_3)] & \rightsquigarrow & C[(v_3\ \mathbf{rec}(v_1, v_2, v_3)\ v_1)] \\ C[\lambda(x_1, \ldots, x_n).t\ (v_1, \ldots, v_n)] & \rightsquigarrow & C[t[v_1/x_1, \ldots, v_n/x_n]] \end{array}$$

Fig. 1. Gödel System T

usual capture-avoiding substitution. We also recall the type system in Figure 2 and we consider only well-typed terms in the sequel.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad (\textsc{ident})$$

$$\frac{\Gamma \vdash t : \mathit{int}}{\Gamma \vdash S(t) : \mathit{int}} \quad (\textsc{succ})$$

$$\frac{\Gamma \vdash t : \mathit{int}}{\Gamma \vdash \mathbf{pred}(t) : \mathit{int}} \quad (\textsc{pred})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n} \quad (\textsc{tuple})$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \sigma}{\Gamma \vdash \lambda(x_1, \ldots, x_n).t \ : (\tau_1 \times \ldots \times \tau_n) \to \sigma} \quad (\textsc{abs})$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \ t_2 : \tau} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t_1 : \mathit{int} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau \to \mathit{int} \to \tau}{\Gamma \vdash \mathbf{rec}(t_1, t_2, t_3) : \tau} \quad (\textsc{rec})$$

Fig. 2. Functional type system

Remark 2.1. In order to distinguish the successor S (which is a constructor) from the successor seen as an operation (whose evaluation should imply a reduction step), we use the keyword **succ** as an abbreviation for $\lambda x.S(x)$.

Notation 2.2. We use the more verbose notation $\mathbf{fn}\ (\vec{x} : \vec{\sigma}) \Rightarrow t$ instead of simply $\lambda\vec{x}.t$ whenever we wish to make the types explicit. Moreover, we write $\vec{\tau}^{\times}$ as an abbreviation for $\tau_1 \times \ldots \times \tau_n$ and we consider *unit* as the special case of $\vec{\tau}^{\times}$ obtained when $n = 0$.

Remark 2.3. As usual, we write $\mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t$ be an abbreviation for the redex $\lambda(x_1, \ldots, x_n).t\ u$. The following typing rule and evaluation rule can be obtained:

$$\frac{\Gamma \vdash u : (\tau_1 \times \ldots \times \tau_n) \quad \Gamma, x_1 : \tau_1, \ldots x_n : \tau_n \vdash t : \sigma}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t : \sigma}$$

$$C[\mathbf{let}\ (x_1, \ldots, x_n) = (v_1, \ldots, v_n)\ \mathbf{in}\ t] \rightsquigarrow C[t[v_1/x_1, \ldots, v_n/x_n]]$$

Eventually, note that $\mathbf{let}\ (x_1, \ldots, x_n) = [\,]\ \mathbf{in}\ t$ is an evaluation context.

Let us recall the well-known infinite syntactic hierarchy of fragment T_n of Gödel System T. We call "recursion rank" of a term t the maximum degree of the types of the recursors which occur in t, where the degree of a type is defined as follows:

Definition 2.4. The degree $\partial(\tau)$ of a type τ is defined inductively by:

— $\partial(\mathit{int}) = 0$
— $\partial(\sigma \to \tau) = \max(\partial(\sigma) + 1, \partial(\tau))$
— $\partial(\tau_1 \times \ldots \times \tau_n) = \max(\partial(\tau_1), \ldots, \partial(\tau_n))$

Remark 2.5. Using standard type isomorphisms, any type is isomorphic either to *unit* or to some type which does not contain *unit*. Thus we may consider without restriction that τ does not contain *unit* in the above definition.

Definition 2.6. The fragment T_n of System T is defined as the set of terms with recursion rank less or equal to n.

2.1 Example: Ackermann function

Our running example in this paper is Ackermann function. This function is known not to be primitive recursive [Peter 1968] but it can be represented in System T, for instance as follows:

$$ack(m,n) = (\mathbf{rec}(m, \lambda y.\mathbf{succ}(y), \lambda h.\lambda i.\lambda y.\mathbf{rec}(y, (h\ S(0)), \lambda k.\lambda j.(h\ k)))\ n)$$

It is well-known (see [Avigad and Feferman 1998] for instance) that functions of type $N^k \to N$ that can be represented by a term of T_0 correspond exactly to primitive recursive functions. Not surprisingly, the term given in the above definition does not belong to T_0 but to T_1 (the outermost **rec** in the above definition has type $int \to int$ and thus has degree 1).

3. THE HIGHER-ORDER LOOP LANGUAGE

In this section, we present the Loop$^\omega$ language. First, we describe the syntax and the type system, then we detail the semantics of the language. Since the structured operational semantics is tailored for the lock-step simulation, we first present the natural semantics. Our purpose is to give evidence that the various syntactic constructs have the usual semantics. In particular, this semantics extends the usual semantics for first order LOOP programs with full-fledged higher-order procedures.

However natural semantics relates a program and an initial store directly to some final store and thus describes only the evaluation of terminating programs. In particular, it fails to distinguish between a non-terminating program and a run-time error. For this reason, we also define a transition semantics which refines the natural semantics. We use the transition semantics for proving the soundness of type systems and the simulation theorem (in order to derive that Loop$^\omega$ programs are always terminating).

Both semantics are location-free semantics. The benefit is clear: this kind of semantics is simpler than a traditional two-level semantics (where the environment binds variables to locations and the store maps locations to values). The main drawback is the difficulty to account for advanced features such as variable aliasing. However, for our purpose, a location-free semantics is sufficient. A discussion on this subject can be found in [Plotkin 1981] p. 70 (see also [Felleisen and Friedman 1987], [Donahue 1977] and [Reynolds 1981]).

3.1 Syntax

The syntax of imperative types and phrase types is the following:

$$\sigma, \tau ::= int \mid \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})$$
$$\mathcal{T} ::= \tau \mid \mathbf{comm} \mid \mathbf{seq}$$

Note that we consider only two formal parameter modes **in** and **out** (borrowed from Ada [DOD 1980]) which specify "abstractly" the direction of data flow between caller and callee (without implying a specific parameter-passing mechanism).

The raw syntax of imperative programs is given below. There is nothing particular to this syntax except that we annotate sequences with a set of variables (see remark 3.3).

$$
\begin{array}{rrcl}
(\textit{commands}) & c & ::= & s \\
 & & | & \textbf{for } y \textbf{ in } 1..e\ s \\
 & & | & y := e \quad | \quad \textbf{inc}(y) \quad | \quad \textbf{dec}(y) \\
 & & | & e(\vec{e};\vec{y}) \\
 & & | & \textbf{cst } y = e\ s \\
 & & | & \textbf{var } y : \tau := e\ s \\
(\textit{sequences}) & s & ::= & \{\}_{\vec{x}} \quad | \quad c\ ;\ s \\
(\textit{expressions}) & e & ::= & y \quad | \quad w \\
(\textit{values}) & w & ::= & \bar{q} \quad | \quad \textbf{proc } (\textbf{in } \vec{y} : \vec{\tau}; \textbf{out } \vec{z} : \vec{\sigma})\ s
\end{array}
$$

Notation 3.1. In the sequel, we adopt the more conventional notation for a sequence $\{c_1; \ldots ; c_m\}_{\vec{x}}$ as an abbreviation for $c_1\ ; \ldots \ ; c_m\ ; \{\}_{\vec{x}}$. Similarly, if s is a sequence, we write $\{s\}_{\vec{x}}$ to emphasize that this sequence is annotated with variables $\vec{x}$.

Remark 3.2. Since anonymous procedures are not a very popular feature in imperative languages, we often use a more conventional notation for declaring local procedures:

$$\textbf{proc } p(\textbf{in } \vec{y} : \vec{\tau}; \textbf{out } \vec{z} : \vec{\sigma})\{\mathit{body}\}_{\vec{z}}\ \ldots$$

This notation is defined as an abbreviation for a constant declaration:

$$\textbf{cst } p = \textbf{proc } (\textbf{in } \vec{y} : \vec{\tau}; \textbf{out } \vec{z} : \vec{\sigma})\{\mathit{body}\}_{\vec{z}} \ldots$$

Similarly, we allow for the declaration of several local variables, constants and procedures in the same declaration block. This should be seen simply as "syntactic sugar" for the corresponding nested declarations.

3.2 Type system

The type system of Loop$^\omega$ may be seen as a simple effect system [Gifford and Lucassen 1986; Talpin and Jouvelot 1994] since it is able to guarantee the absence of side-effects, aliasing and back-patching in well-typed programs. Its main feature is the distinction between mutable variables and read-only variables. More formally, a typing environment has the form $\Gamma; \Omega$ where Γ and Ω are (possibly empty) lists of pairs $x : \tau$ (x ranges overs 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). The type system is given in Figure 3. 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.

$$\frac{x : \tau \in \Gamma; \Omega}{\Gamma; \Omega \vdash x : \tau} \quad (\textsc{t.var})$$

$$\Gamma; \Omega \vdash \bar{q} : \mathit{int} \quad (\textsc{t.num})$$

$$\frac{\vec{x} \subset \Omega}{\Gamma; \Omega \vdash \{\}_{\vec{x}} : \mathbf{seq}} \quad (\textsc{t.seq-i})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma; \Omega \vdash c : \mathbf{comm} \qquad \Gamma; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \{c; s\}_{\vec{x}} : \mathbf{seq}} \quad (\textsc{t.seq-ii})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma; \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{comm}} \quad (\textsc{t.comm})$$

$$\frac{y : \mathit{int} \in \Omega}{\Gamma; \Omega \vdash \mathbf{inc}(y) : \mathbf{comm}} \quad (\textsc{t.inc})$$

$$\frac{y : \mathit{int} \in \Omega}{\Gamma; \Omega \vdash \mathbf{dec}(y) : \mathbf{comm}} \quad (\textsc{t.dec})$$

$$\frac{y : \tau \in \Omega \quad \Gamma; \Omega \vdash e : \tau}{\Gamma; \Omega \vdash y := e : \mathbf{comm}} \quad (\textsc{t.assign})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma; \Omega \vdash e : \tau \qquad \Gamma, y : \tau; \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \mathbf{cst}\ y = e\ \{s\}_{\vec{x}} : \mathbf{comm}} \quad (\textsc{t.cst})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma; \Omega \vdash e : \tau \qquad \Gamma; y : \tau, \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \mathbf{var}\ y : \tau := e\ \{s\}_{\vec{x}} : \mathbf{comm}} \quad (\textsc{t.var})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma; \Omega \vdash e : \mathit{int} \qquad \Gamma, y : \mathit{int}; \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{x}} : \mathbf{comm}} \quad (\textsc{t.for})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y} : \vec{\tau}; \vec{z} : \vec{\sigma} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\tau}; \mathbf{out}\ \vec{z} : \vec{\sigma})\{s\}_{\vec{z}} : \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})} \quad (\textsc{t.proc})$$

$$\frac{\Gamma; \Omega \vdash p : \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \qquad \Gamma; \Omega \vdash \vec{e} : \vec{\tau} \qquad \vec{r} : \vec{\sigma} \in \Omega}{\Gamma; \Omega \vdash p(\vec{e}; \vec{r}) : \mathbf{comm}} \quad (\textsc{t.call})$$

Fig. 3. Imperative type system

Remark 3.3. The type system guarantees that in a sequence $\{s\}_{\vec{x}}$, the variables of $\vec{x}$ are all visible (according to standard scoping rules) and $\vec{x}$ contains all the (free) mutable variables occurring in s except the eventual local variable (rule T.VAR). For simplicity, we assume that these annotations are supplied by the programmer. Note that missing annotations can automatically be inferred by taking either all the visible mutable variables of the sequence, or only its free mutable variables. However, we require that the body of a procedure is annotated by all its **out** parameters (rule T.PROC). Similarly, a local mutable variable should not occur in the list which annotates the corresponding local block (rule T.VAR).

Remark 3.4. *(aliasing).* In order to avoid parameter-induced aliasing problems, we assume that all r_i are pairwise distinct in rule (T.CALL). Indeed, we will see that in our semantics, mutable variables directly denote values and consequently different variables cannot refer to the same location in the store. However, aliasing is known to be problematic for most aspects of programming languages (see for instance [Gellerich and Plödereder 2001] and [Filliâtre 2003]) and we regard the absence of parameter-induced aliasing in LOOP$^\omega$ more as a feature than as a limitation.

Remark 3.5. *(side-effects).* Rule (T.PROC) implies that the only mutable variables which may occur inside the body of a procedure are its **out** parameters and its local mutable variables. This is enough to guarantee the absence of side-effects (i.e. modification of non-local variables). The purpose of this restriction is mainly to simplify the semantics of the language. However, side-effects can still be simulated in most cases by passing the non-local variable as an explicit **in out** parameter. This simulation is often referred to as the "state-passing transform" (see for instance [Filinski 1994] and [Wadler 1990]).

Remark 3.6. *(back-patching).* Rule (T.PROC) also forbids the reading of non-local mutable variables: this is necessary to prevents the definition of fix-points in the language. Indeed, there is a well-known technique (called "back-patching") which takes advantage of higher-order mutable variables (or function pointers) to define arbitrary recursive functions. This technique is used for instance in Scheme's semantics of the **letrec** construct [Kelsey et al. 1998]. Here is such a definition of a fix-point in the LOOP$^\omega$ syntax (but since fix : **proc** (**out** *int*) is a mutable variable, this sequence is not typable in LOOP$^\omega$):

```
{
   fix := proc (out x: int){}_x;
   var f: proc (out int);
   {
       f := proc (out x: int){ fix(x); }_x;
       fix := f;
   }_fix
}_fix
```

On the other hand, local procedures are not required to be closed: non-local read-only variable (such as outer **in** parameters) are allowed. We shall see that this is sufficient to encode a pure functional language.

3.3 Natural semantics

Since the evaluation relation in only defined for well-typed programs, we also need to define the notions of well-typed stores and states.

Definition 3.7. A store μ is a finite mapping from (mutable) variables to closed imperative values (i.e. integer literals and procedures). A state is a pair (c, μ) consisting of a command c and μ.

Definition 3.8. *(store typing).* We say that μ is typable in Ω, which we write as $\Omega \vdash \mu$, if and only if for all $x : \tau \in \Omega$, $x \in dom(\mu)$ and $\emptyset;\emptyset \vdash \mu(x) : \tau$.

Definition 3.9. *(state typing).* We say that a state (c, μ) is typable in Ω, which we write as $\Omega \vdash (c, \mu)$, if and only if $\emptyset;\Omega \vdash c : \mathbf{comm}$ and $\Omega \vdash \mu$.

Note that expressions do not require any evaluation (since they are either variables or values), but only fetching the corresponding value from the store whenever the expression is a mutable variable. We introduce thus the following notation:

Notation 3.10. Given a store μ, let φ_μ be the trivial extension of μ to expressions defined as follows $\varphi_\mu(x) = \mu(x)$ if x is a variable and $\varphi_\mu(w) = w$ otherwise. In the sequel, we write $e =_\mu w$ for $\varphi_\mu(e) = w$.

In order to assign default values to **out** parameters and uninitialized local variables, we define a closed imperative value for each imperative type. In the sequel, uninitialized local variables are assumed to be implicitly initialized by the default value corresponding to their type.

Definition 3.11. For each type σ, we define inductively a default closed imperative value $\epsilon(\sigma)$ as follows:

— $\epsilon(int) = 0$
— $\epsilon(\mathbf{proc}\ (\mathbf{in}\ \vec{\tau};\mathbf{out}\ \vec{\sigma})) = \mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\tau};\mathbf{out}\ z : \vec{\sigma})\{\}_{\vec{z}}$

LEMMA 3.12. *The typing judgment* $\vdash \epsilon(\sigma) : \sigma$ *is derivable.*

Remark 3.13. Default values could be dispensed with using standard data-flow analysis (see [Appel 1998] for instance). This technique (also called liveness analysis) allows to determine whether there is a potential execution path on which a variable is used before it has been assigned an initial value. We preferred not to complicate the static analysis of LOOP$^\omega$ programs although such an analysis is certainly convenient from a practical standpoint.

Notation 3.14. Let c be a command. We write $c[x \leftarrow w]$ for the substitution of a read-only variable x by a closed imperative value w and $c[y \hookleftarrow z]$ for the renaming of a mutable variable y by a mutable variable z. The formal definitions are given in Appendix B.

We are now ready define the natural semantics of Loop$^\omega$. The inductive definition of the evaluation relation (which we write $\Downarrow$) is summarized in Figure 4.

Remark 3.15. We rely on the renaming and substitution meta-operations in rule (S.CALL) in order to avoid building closures in the semantics. Besides, **in** parameters are passed by-copy (i.e. by-value) whereas **out** parameters are passed by-reference. Note that since parameter-induced aliasing is forbidden, this latter choice has no consequence on the denotational semantics of the language (but it is important from a complexity standpoint).

$$(\{\}_{\vec{x}}, \mu) \Downarrow \mu \qquad \text{(N.EMPTY)}$$

$$\frac{(\{s\}_{\vec{x}}, \mu) \Downarrow \mu'}{(\{\{\}_{\vec{z}}; s\}_{\vec{x}}, \mu) \Downarrow \mu'} \qquad \text{(N.SEQ-I)}$$

$$\frac{(\{s\}_{\vec{x}}, \mu) \Downarrow \mu'}{(\{\mathbf{var}\ y : \tau := e\ \{\}_{\vec{z}}; s\}_{\vec{x}}, \mu) \Downarrow \mu'} \qquad \text{(N.VAR-I)}$$

$$\frac{e =_\mu w \qquad (\{s\}_{\vec{z}}, \mu[y \leftarrow w]) \Downarrow \mu'}{(\{y := e; s\}_{\vec{z}}, \mu) \Downarrow \mu'} \qquad \text{(N.ASSIGN)}$$

$$\frac{(c, \mu) \Downarrow \mu' \qquad (\{s\}_{\vec{x}}, \mu') \Downarrow \mu''}{(\{c; s\}_{\vec{x}}, \mu) \Downarrow \mu''} \qquad \text{(N.SEQ-II)}$$

$$\frac{e =_\mu w \qquad (\{s\}_{\vec{x},y}, (\mu, y \leftarrow w)) \Downarrow (\mu', y \leftarrow w'))}{(\mathbf{var}\ y : \tau := e\ \{s\}_{\vec{x}}, \mu)) \Downarrow \mu'} \qquad \text{(N.VAR-II)}$$

$$\frac{\mu(y) = \bar{q}}{(\mathbf{inc}(y), \mu)) \Downarrow \mu[y \leftarrow \overline{q+1}]} \qquad \text{(N.INC)}$$

$$\frac{\mu(y) = \bar{q}}{(\mathbf{dec}(y), \mu) \Downarrow \mu[y \leftarrow \overline{q \dot{-} 1}]} \qquad \text{(N.DEC)}$$

$$\frac{\vec{e} =_\mu \vec{w}, p =_\mu \mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}} \quad (\{s[\vec{y} \leftarrow \vec{w}][\vec{z} \hookrightarrow \vec{r}]\}_{\vec{r}}, \mu[\vec{r} \leftarrow \epsilon(\vec{\tau})]) \Downarrow \mu'}{(p(\vec{e}; \vec{r}), \mu) \Downarrow \mu'} \qquad \text{(N.CALL)}$$

$$\frac{e =_\mu w \qquad (\{s\}_{\vec{z}}[y \leftarrow w], \mu) \Downarrow \mu'}{(\mathbf{cst}\ y = e\ \{s\}_{\vec{z}}, \mu) \Downarrow \mu'} \qquad \text{(N.CST)}$$

$$\frac{e =_\mu \bar{0}}{(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}}, \mu) \Downarrow \mu'} \qquad \text{(N.FOR-I)}$$

$$\frac{e =_\mu \overline{q+1} \qquad (\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}}, \mu) \Downarrow \mu' \qquad (\{s\}_{\vec{z}}[y \leftarrow \overline{q+1}], \mu') \Downarrow \mu''}{(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}}, \mu) \Downarrow \mu''} \qquad \text{(N.FOR-II)}$$

Fig. 4. Natural semantics

3.4 Transition semantics

As expected, the transition semantics of LOOP$^\omega$ is given by a transition system which defines inductively a binary relation between states [Plotkin 1981]. The main design choices were influenced by the expected cost of each command. For instance, this explains the rule (S.ASSIGN) where the assignment is dealt with directly inside the sequence. Similarly, the notation $=_\mu$ allows us to hide the cost of fetching the value of a variable from the store. The transition semantics is summarized in Figure 5.

Remark 3.16. This semantics is clearly deterministic since there is always at most one rule which can be applied (depending on the environment and the shape of the program). Moreover, the only case where no rule can be applied corresponds to the final state (when the program is reduced to an empty sequence).

$$(\{\{\}_{\vec{z}}; s\}_{\vec{x}}, \mu) \mapsto (\{s\}_{\vec{x}}, \mu) \qquad \text{(S.SEQ-I)}$$

$$(\{\mathbf{var}\ y : \tau := e\ \{\}_{\vec{z}}; s\}_{\vec{x}}, \mu) \mapsto (\{s\}_{\vec{x}}, \mu) \qquad \text{(S.VAR-I)}$$

$$\frac{e =_\mu w}{(\{y := e; s\}_{\vec{z}}, \mu) \mapsto (\{s\}_{\vec{z}}, \mu[y \leftarrow w])} \qquad \text{(S.ASSIGN)}$$

$$\frac{(c, \mu) \mapsto (c', \mu')}{(\{c; s\}_{\vec{x}}, \mu) \mapsto (\{c'; s\}_{\vec{x}}, \mu')} \qquad \text{(S.SEQ-II)}$$

$$\frac{e =_\mu w \qquad (\{s\}_{\vec{x},y}, (\mu, y \leftarrow w)) \mapsto (\{s'\}_{\vec{x},y}, (\mu', y \leftarrow w'))}{(\mathbf{var}\ y : \tau := e\ \{s\}_{\vec{x}}, \mu) \mapsto (\mathbf{var}\ y : \tau := w'\ \{s'\}_{\vec{x}}, \mu')} \qquad \text{(S.VAR-II)}$$

$$\frac{\mu(y) = \bar{q}}{(\mathbf{inc}(y), \mu) \mapsto (y := \overline{q+1}, \mu)} \qquad \text{(S.INC)}$$

$$\frac{\mu(y) = \bar{q}}{(\mathbf{dec}(y), \mu) \mapsto (y := \overline{q \dot{-} 1}, \mu)} \qquad \text{(S.DEC)}$$

$$\frac{\vec{e} =_\mu \vec{w} \qquad p =_\mu \mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}}}{(p(\vec{e}; \vec{r}), \mu) \mapsto (\{s[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}]\}_{\vec{r}}, \mu[\vec{r} \leftarrow \epsilon(\vec{\tau})])} \qquad \text{(S.CALL)}$$

$$\frac{e =_\mu w}{(\mathbf{cst}\ y = e\ \{s\}_{\vec{z}}, \mu) \mapsto (\{s\}_{\vec{z}}[y \leftarrow w], \mu)} \qquad \text{(S.CST)}$$

$$\frac{e =_\mu \bar{0}}{(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}}, \mu) \mapsto (\{\}_{\vec{z}}, \mu)} \qquad \text{(S.FOR-I)}$$

$$\frac{e =_\mu \overline{q+1}}{(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}}, \mu) \mapsto (\{\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}}; \mathbf{cst}\ y = \overline{q+1}\ \{s\}_{\vec{z}}\}_{\vec{z}}, \mu)} \qquad \text{(S.FOR-II)}$$

Fig. 5. Transition semantics

Remark 3.17. It is worth mentioning that rules (S.VAR-I) and (S.VAR-II) allows to give a simple semantics to local variable without dealing with an explicit stack. Although this is usual in natural semantics, this technique is not widespread in transition semantics. The ingenious idea consisting in updating a local variable directly in the source (in rule S.VAR-II) is attributed to Eugene Fink in [Reynolds 1998] p. 130.

As expected, the "subject reduction" property holds for the transition semantics. The proof of the following theorem is given in Appendix B.

THEOREM 3.18. *For any environment* Ω *and any state* (c, μ)*, we have that* $\emptyset; \Omega \vdash (c, \mu)$ *and* $(c, \mu) \mapsto (c', \mu')$ *implies* $\emptyset; \Omega \vdash (c', \mu')$ *and* $dom(\mu) = dom(\mu')$*.*

Moreover, the equivalence of the natural and transition semantics for LOOP$^\omega$ can be proved by establishing the following two usual lemmas.

LEMMA 3.19. *The relation* $(\{s\}_{\vec{x}}, \mu) \mapsto^\star (\{\}_{\vec{x}}, \mu')$ *is closed under the defining conditions of the* $\Downarrow$ *relation.*

LEMMA 3.20. *The $\Downarrow$ relation is closed under head expansion: if $(\{s\}_{\vec{x}}, \mu) \mapsto (\{s'\}_{\vec{x}}, \mu')$ and $(\{s'\}_{\vec{x}}, \mu') \Downarrow \mu''$ then $(\{s\}_{\vec{x}}, \mu) \Downarrow \mu''$.*

4. LOCK-STEP SIMULATION

In this section, we show how to translate a LOOP$^\omega$ into a term of System T and we prove the simulation theorem. Then we exhibit a hierarchy of fragments LOOP^n which is imperative counterpart of the fragment T_n of Gödel System T. Eventually, we introduce the notion of "unary" LOOP$^\omega$ program whose translation does not require the product type in System T.

4.1 Translation

In order to translate default imperative values, we shall need corresponding default functional values.

Definition 4.1. For each type σ, we define inductively a default closed value $\delta(\sigma)$ as follows:

— $\delta(\mathit{int}) = 0$
— $\delta(\sigma \to \tau) = \mathbf{fn}\ (x : \sigma) \Rightarrow \delta(\tau)$
— $\delta(\sigma_1 \times \ldots \times \sigma_n) = (\delta(\sigma_1), \ldots, \delta(\sigma_n))$

We also write $\delta(\vec{\sigma})$ as an abbreviation for $(\delta(\sigma_1), \ldots, \delta(\sigma_n))$.

LEMMA 4.2. *The typing judgment $\vdash \delta(\sigma) : \sigma$ is derivable.*

We are now ready to define the translation of imperative types. Note that the translation of a procedure type encode exactly the data-flow specified by the formal parameter modes **in** and **out** (and this is sufficient since side-effects are not allowed).

Definition 4.3. The translations $\sigma^\star$ of an imperative type σ is defined inductively as follows:

— $\mathit{int}^\star = \mathit{int}$
— $\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})^\star = (\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times$

Moreover, if Γ is $x_1 : \tau_1, \ldots, x_n : \tau_n$, we define $\Gamma^\star$ as $x_1 : \tau_1^\star, \ldots, x_n : \tau_n^\star$.

The intuition behind the translation of imperative programs is the following: a sequence $\{c_1; \ldots; c_n\}_{\vec{x}}$ is translated into:

$$\mathbf{let}\ \vec{x}_1 = c_1^\star\ \mathbf{in}\ \ldots\ \mathbf{let}\ \vec{x}_n = c_n^\star\ \mathbf{in}\ \vec{x}$$

where each $\vec{x}_i \subseteq \vec{x}$ corresponds to the "output" of command c_i and $\vec{x}$ is the output of the sequence. Note in particular that the same identifier is used again and again in order to simulate imperative updates. For instance, the sequence $\{\mathbf{inc}(x); \mathbf{inc}(x)\}_x$ is translated as:

$$\mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ x$$

Let us now give the formal definition the translation and then present a complete example.

Definition 4.4. The translations $e^\star$ and $\{s\}^\star_{\vec{x}}$ of respectively an expression e and a sequence $\{s\}_{\vec{x}}$ into a term of System T are defined by mutual induction as follows:

— $\bar{n}^\star = S^n(0)$
— $y^\star = y$
— $(\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}})^\star = \mathbf{fn}\ (\vec{y} : \vec{\sigma}^\star) \Rightarrow \{s\}^\star_{\vec{z}}\ [\delta(\vec{\tau}^\star)/\vec{z}]$
— $\{\}^\star_{\vec{x}} = \vec{x}$
— $\{y := e; s\}^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$
— $\{\mathbf{inc}(y); s\}^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$
— $\{\mathbf{dec}(y); s\}^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$
— $\{p(\vec{e}; \vec{z}); s\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = p^\star\ \vec{e}^\star\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$
— $\{\{s_1\}_{\vec{z}}; s_2\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = \{s_1\}^\star_{\vec{z}}\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}}$
— $\{\mathbf{var}\ y : \tau := e\ \{s_1\}_{\vec{z}}; s_2\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = \{s_1\}^\star_{\vec{z}}[e^\star/y]\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}}$
— $\{\mathbf{cst}\ y = e\ \{s_1\}_{\vec{z}}; s_2\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = (\mathbf{let}\ y = e^\star\ \mathbf{in}\ \{s_1\}^\star_{\vec{z}})\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}}$
— $\{\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s_1\}_{\vec{z}}; s_2\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = \mathbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s_1\}^\star_{\vec{z}})\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}}$

Remark 4.5. It is possible to factor out the translation of commands in the above definition. Indeed, the translation of a non-empty sequence $\{c; s\}_{\vec{x}}$ always follows the same pattern: $\{c; s\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = c^\star\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$ where $\vec{z}$ is a list of "output" variables of depending on the command c. The translation $c^\star$ of a command c together with its output variables $\mathcal{O}(c)$ are summarized in the following table:

c	$c^\star$	$\mathcal{O}(c)$
$y := e$	$e^\star$	y
$\mathbf{inc}(y)$	$\mathbf{succ}(y)$	y
$\mathbf{dec}(y)$	$\mathbf{pred}(y)$	y
$p(\vec{e}, \vec{r})$	$p^\star\ \vec{e}^\star$	$\vec{r}$
$\{s\}_{\vec{z}}$	$\{s\}^\star_{\vec{z}}$	$\vec{z}$
$\mathbf{var}\ y : \tau := e\ \{s\}_{\vec{z}}$	$\{s\}^\star_{\vec{z}}[e^\star/y]$	$\vec{z}$
$\mathbf{cst}\ y = e\ \{s\}_{\vec{z}}$	$\mathbf{let}\ y = e^\star\ \mathbf{in}\ \{s\}^\star_{\vec{z}}$	$\vec{z}$
$\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}}$	$\mathbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})$	$\vec{z}$

LEMMA 4.6. *For any imperative type σ, we have $\epsilon(\sigma)^\star = \delta(\sigma^\star)$.*

The following theorem states that translation $(\ )^\star$ is type-preserving (its proof is given in appendix C).

THEOREM 4.7. *For any environments Γ and Ω, any imperative expression e and any imperative sequence $\{s\}_{\vec{x}}$ we have:*

— $\Gamma; \Omega \vdash e : \tau$ *implies* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$
— $\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{seq}$ *implies* $\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash \{s\}^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times$

4.2 Example: Ackermann function

Here is an implementation of Ackermann function as an anonymous procedure of LOOP$^\omega$. This program was actually programmed by hand, but we shall see in Section 5 how to get almost the same program from the definition of *ack* in System T.

```
proc (in m:int, n: int; out r : int) {
    proc next(in y: int; out p: int) {
        p := y;
        inc(p);
    }p;
    var g : proc (in int; out int); {
        g := next;
        for i := 1 to m {
            cst h = g {
                proc aux(in y: int; out p: int) {
                    h(1, p);
                    for j := 1 to y {
                        h(p; p);
                    }p;
                }p;
                g := aux;
            }g;
        }g;
        g(n, r);
    }r;
}r;
```

By applying translation $(\ )^\star$, we obtain the following anonymous function. For clarity, we used Standard ML derived forms [Milner et al. 1997] for declaring several functions (keyword **fun**) and values (keyword **val**) with a single **let**.

```
fn (m: int, n: int) ⇒ let
   fun next(y: int) = let
        val p = y
        val p = succ(p)
   in p end
   val r = let
        val g = next
        val g = rec(m, g, λg.λi.let
                        val g = let
                            val h = g
                            fun aux(y: int) = let
                                val p = h S(0)
                                val p = rec(y, p, λp.λj.let
                                                val p = h p
                                            in p end)
                            in p end
                            val g = aux
                        in g end
                    in g end)
        val r = g n
   in r end
in r end
```

Note that although we could prove by hand that this functional program is equivalent to the term *ack* given in Section 4.2, we shall see that this property is mostly an application of the main theorem of Section 5.

4.3 Simulation theorem

Let us prove the main theorem which states that for any sequence $\{s\}_{\vec{x}}$, the evaluation of $\{s\}_{\vec{x}}$ runs in lock-step with the reduction of $\{s\}^{\star}_{\vec{x}}$. We first need some preliminary substitution lemmas:

Notation 4.8. If μ is a store and $\vec{x} \subset dom(\mu)$, we write $\mu(\vec{x})$ for $(\mu(x_1), \ldots, \mu(x_n))$. Moreover, if $\vec{v} = (v_1, \ldots, v_k)$ we write $\vec{v}^{\star}$ for the tuple $(v_1^{\star}, \ldots, v_k^{\star})$.

LEMMA 4.9. *For any sequence* $\{s\}_{\vec{z}}$ *such that* $\Gamma; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{seq}$*, the following equality holds:* $\{s\}^{\star}_{\vec{z}}[\delta(\vec{\tau})/\vec{z}] = \{s[\vec{z} \hookleftarrow \vec{r}]\}^{\star}_{\vec{r}}[\delta(\vec{\tau})/\vec{r}]$.

LEMMA 4.10. *For any command* c*, any read-only variable* x *and any closed imperative value* w*, we have* $(c[\vec{x} \leftarrow \vec{w}])^{\star} = c^{\star}[\vec{w}^{\star}/\vec{x}]$.

LEMMA 4.11. *If* $e =_{\mu} w$ *then either* $e = w$ *or* e *is some variable* x_i *with* $\mu(x_i) = w$ *and then* $c^{\star}[\mu(\vec{x})^{\star}/\vec{x}] = c^{\star}[w^{\star}/x_i][\mu(\vec{x})^{\star}/\vec{x}]$.

THEOREM 4.12. *For any well-typed state* (c, μ)*, if* $\vec{x} = dom(\mu)$ *we have:*

$$(c, \mu) \mapsto (c', \mu') \text{ implies } c^{\star}[\mu(\vec{x})^{\star}/\vec{x}] \rightsquigarrow c'^{\star}[\mu'(\vec{x})^{\star}/\vec{x}]$$

PROOF. By induction on the derivation of $(c, \mu) \mapsto (c', \mu')$. For brevity, we write $\vec{v}$ for $\mu(\vec{x})$ and $\vec{v}'$ for $\mu'(\vec{x})$.

—(S.SEQ-I)

$$\{\{\}_{\vec{y}}; s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}] \rightsquigarrow \{s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}]$$

Indeed:

$$\begin{aligned}
&\{\{\}_{\vec{y}}; s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{y} = \{\}^{\star}_{\vec{y}}\ \mathbf{in}\ \{s\}^{\star}_{\vec{z}})[\vec{v}^{\star}/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{y} = \vec{y}\ \mathbf{in}\ \{s\}_{\vec{z}})^{\star}[\vec{v}^{\star}/\vec{x}] \\
&\quad \rightsquigarrow \{s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}]
\end{aligned}$$

—(S.VAR-I)

$$\{\mathbf{var}\ y : \tau := e\ \{\}_{\vec{r}}; s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}] \rightsquigarrow \{s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}]$$

Indeed:

$$\begin{aligned}
&\{\mathbf{var}\ y : \tau := e\ \{\}_{\vec{r}}; s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{r} = \{\}^{\star}_{\vec{r}}[e^{\star}/y]\ \mathbf{in}\ \{s\}^{\star}_{\vec{z}})[\vec{v}^{\star}/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{r} = \vec{r}[e^{\star}/y]\ \mathbf{in}\ \{s\}^{\star}_{\vec{z}})[\vec{v}^{\star}/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{r} = \vec{r}\ \mathbf{in}\ \{s\}^{\star}_{\vec{x}})[\vec{v}^{\star}/\vec{x}] \\
&\quad \rightsquigarrow \{s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}]
\end{aligned}$$

—(S.ASSIGN)

$$\{x_i := e; s\}^{\star}_{\vec{z}}[\vec{v}^{\star}/\vec{x}] \rightsquigarrow \{s\}^{\star}_{\vec{z}}[\vec{v}'^{\star}/\vec{x}]$$

with $\vec{v}' = \mu[x_i \leftarrow w](\vec{x}) = (v_1, \ldots, v_{i-1}, w, v_{i+1}, \ldots, v_k)$. Since $e =_{\mu} w$, by lemma 4.11:

$$\begin{aligned}
&\{x_i := e; s\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \\
&\quad = (\textbf{let } x_i = e^\star \textbf{ in } \{s\}^\star_{\vec{z}})[(v_1,\ldots,v_k)^\star/\vec{x}] \\
&\quad = (\textbf{let } x_i = w^\star \textbf{ in } \{s\}^\star_{\vec{z}})[(v_1,\ldots,v_k)^\star/\vec{x}] \\
&\quad \rightsquigarrow \{s\}^\star_{\vec{z}}[w^\star/x_i][(v_1,\ldots,v_k)^\star/\vec{x}] \\
&\quad = \{s\}^\star_{\vec{z}}[(v_1,\ldots,v_{i-1},w,v_{i+1},\ldots,v_k)^\star/\vec{x}]
\end{aligned}$$

—(S.VAR-II)

$$\frac{\{s\}^\star_{\vec{z}}[\mu[y \leftarrow w](\vec{x})^\star/\vec{x}] \rightsquigarrow \{s'\}^\star_{\vec{z}}[\mu'(\vec{x})^\star/\vec{x}]}{(\textbf{var } y : \tau := e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow (\textbf{var } y : \tau := w'\ \{s'\}_{\vec{z}})^\star[\vec{v}'^\star/\vec{x}]}$$

Since $e =_\mu w$ and $w' = \mu'(y)$, by lemma 4.11:

$$\begin{aligned}
&(\textbf{var } y : \tau := e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}] \\
&\quad = (\{s\}^\star_{\vec{z}}[e^\star/y])[\vec{v}^\star/\vec{x}] \\
&\quad = (\{s\}^\star_{\vec{z}}[w^\star/y])[\vec{v}^\star/\vec{x}] \\
&\quad \rightsquigarrow \{s'\}^\star_{\vec{z}}[\vec{v}'^\star/\vec{x}] \text{ by induction hypothesis} \\
&\quad = (\{s'\}^\star_{\vec{z}}[w'^\star/y])[\vec{v}'^\star/\vec{x}] \text{ since } w' = \mu'(y) \\
&\quad = (\textbf{var } y : \tau := w'\ \{s'\}_{\vec{z}})^\star[\vec{v}'^\star/\vec{x}]
\end{aligned}$$

—(S.SEQ-II)

$$\frac{c^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow c'^\star[\vec{v}'^\star/\vec{x}]}{\{c; s\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \rightsquigarrow \{c'; s\}^\star_{\vec{z}}[\vec{v}'^\star/\vec{x}]}$$

Indeed, if $\vec{r} = \mathcal{O}(c)$ as defined in Remark 4.5, we have:

$$\begin{aligned}
&\{c; s\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \\
&\quad = (\textbf{let } \vec{r} = c^\star \textbf{ in } \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}] \\
&\quad = (\textbf{let } \vec{r} = c^\star[\vec{v}^\star/\vec{x}] \textbf{ in } \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}] \\
&\quad \rightsquigarrow (\textbf{let } \vec{r} = c'^\star[\vec{v}'^\star/\vec{x}] \textbf{ in } \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}] \text{ by induction hypothesis} \\
&\quad = (\textbf{let } \vec{r} = c'^\star[\vec{v}'^\star/\vec{x}] \textbf{ in } \{s\}^\star_{\vec{z}})[\vec{v}'^\star/\vec{x}] \text{ since } \mu^\star(y) = \mu'^\star(y) \text{ for any } y \notin \vec{r} \\
&\quad = (\textbf{let } \vec{r} = c'^\star \textbf{ in } \{s\}^\star_{\vec{z}})[\vec{v}'^\star/\vec{x}] \\
&\quad = \{c'; s\}^\star_{\vec{z}}[\vec{v}'^\star/\vec{x}]
\end{aligned}$$

—(S.INC)

$$(\textbf{inc}(y))^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow \overline{q+1}^\star[\vec{v}^\star/\vec{x}]$$

Indeed, since $\mu(y) = \bar{q}$:

$$\begin{aligned}
&(\textbf{inc}(y))^\star[\vec{v}^\star/\vec{x}] \\
&\quad = \textbf{succ}(y)[\vec{v}^\star/\vec{x}] \\
&\quad = \textbf{succ}(S^q(0)) \\
&\quad \rightsquigarrow (S^{q+1}(0))) \\
&\quad = \overline{q+1}^\star
\end{aligned}$$

—(S.DEC) This case is similar to (S.INC).

—(S.CALL)

$$p(\vec{e},\vec{r})^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow \{s[\vec{z} \hookleftarrow \vec{r}][\vec{y} \leftarrow \vec{w}]\}^\star_{\vec{r}}[\vec{v}'^\star/\vec{x}]$$

where $\vec{v}' = \mu'(\vec{x}) = \mu[\vec{r} \leftarrow \epsilon(\vec{\tau})](\vec{x})$. By Lemma 4.11, since $\vec{e} =_\mu \vec{w}$ and $p =_\mu$ **proc** (**in** $\vec{y} : \vec{\sigma}$; **out** $\vec{z} : \vec{\tau}$)$\{s\}_{\vec{z}}$:

$$
\begin{aligned}
&p(\vec{e},\vec{r})^\star[\vec{v}^\star/\vec{x}]\\
&\quad= (p^\star\ \vec{e}^\star)[\vec{v}^\star/\vec{x}]\\
&\quad= (\mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma};\mathbf{out}\ \vec{z}:\vec{\tau})\{s\}_{\vec{z}})^\star\ \vec{w}^\star)[\vec{v}^\star/\vec{x}]\\
&\quad= (\mathbf{fn}\ (\vec{y}:\vec{\sigma}^\star)\Rightarrow\{s\}^\star_{\vec{z}}[\delta(\vec{\tau}^\star)/\vec{z}])\ \vec{w}^\star)[\vec{v}^\star/\vec{x}]\\
&\quad\rightsquigarrow \{s\}^\star_{\vec{z}}[\delta(\vec{\tau}^\star)/\vec{z}][\vec{w}^\star/\vec{y}][\vec{v}^\star/\vec{x}]\\
&\quad= \{s[\vec{z}\hookleftarrow\vec{r}]\}^\star_{\vec{r}}[\delta(\vec{\tau}^\star)/\vec{r}][\vec{w}^\star/\vec{y}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.9}\\
&\quad= \{s[\vec{z}\hookleftarrow\vec{r}]\}^\star_{\vec{r}}[\vec{w}^\star/\vec{y}][\epsilon(\vec{\tau})^\star/\vec{r}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.6}\\
&\quad= \{s[\vec{z}\hookleftarrow\vec{r}][\vec{y}\leftarrow\vec{w}]\}^\star_{\vec{r}}[\epsilon(\vec{\tau})^\star/\vec{r}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.10}\\
&\quad= \{s[\vec{z}\hookleftarrow\vec{r}][\vec{y}\leftarrow\vec{w}]\}^\star_{\vec{r}}[\vec{v}'^\star/\vec{x}] \text{ since } \vec{r}\subset\vec{x}.
\end{aligned}
$$

—(S.CST)

$$(\mathbf{cst}\ y=e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\rightsquigarrow\{s[y\leftarrow w]\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e=_\mu w$, by Lemma 4.11:

$$
\begin{aligned}
&(\mathbf{cst}\ y=e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\\
&\quad=\ (\mathbf{let}\ y=e^\star\ \mathbf{in}\ \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad=\ (\mathbf{let}\ y=w^\star\ \mathbf{in}\ \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad\rightsquigarrow (\{s\}^\star_{\vec{z}}[w^\star/y])[\vec{v}^\star/\vec{x}]\\
&\quad=\ \{s[y\leftarrow w]\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \text{ by Lemma 4.10}
\end{aligned}
$$

—(S.FOR-I)

$$(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\rightsquigarrow\{\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e=_\mu 0$, by Lemma 4.11:

$$
\begin{aligned}
&(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\\
&\quad=\ \mathbf{rec}(e^\star,\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad=\ \mathbf{rec}(0,\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad\rightsquigarrow\ \vec{z}[\vec{v}^\star/\vec{x}]\\
&\quad=\ \{\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]
\end{aligned}
$$

—(S.FOR-II)

$$(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\rightsquigarrow\{\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}};\mathbf{cst}\ y=\overline{q+1}\ \{s\}_{\vec{z}}\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e=_\mu \overline{q+1}$, by Lemma 4.11:

$$
\begin{aligned}
&(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\\
&\quad=\ \mathbf{rec}(e^\star,\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad=\ \mathbf{rec}(S^{q+1}(0),\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad\rightsquigarrow ((\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})\ \mathbf{rec}(S^q(0),\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})\ S^{q+1}(0))[\vec{v}^\star/\vec{x}]\\
&\quad= (\mathbf{let}\ \vec{z}=\mathbf{rec}(S^q(0),\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{x}})\ \mathbf{in}\ \mathbf{let}\ y=S^{q+1}(0)\ \mathbf{in}\ \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad= (\mathbf{let}\ \vec{z}=\mathbf{rec}(S^q(0),\vec{z},\lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{x}})\ \mathbf{in}\ (\mathbf{cst}\ y=\overline{q+1}\ \{s\}_{\vec{z}})^\star)[\vec{v}^\star/\vec{x}]\\
&\quad= (\mathbf{let}\ \vec{z}=(\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}})^\star\ \mathbf{in}\ (\mathbf{cst}\ y=\overline{q+1}\ \{s\}_{\vec{z}})^\star)[\vec{v}^\star/\vec{x}]\\
&\quad= \{\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}};\mathbf{cst}\ y=\overline{q+1}\ \{s\}_{\vec{z}}\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]
\end{aligned}
$$

□

COROLLARY 4.13. *All programs of* LOOP$^\omega$ *are terminating.*

PROOF. Indeed, since System T is strongly normalizing (see [Girard et al. 1989] for instance). □

Let us now focus on the imperative counterpart of the hierarchy of fragment T_n of Gödel System T. We define first the level of an imperative type and then the "iteration rank" of a LOOP$^\omega$ program as follows:

Definition 4.14. The level $\ell(\tau)$ of an imperative type τ is defined inductively by:

—$\ell(\mathit{int}) = 0$
—$\ell(\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}}) = \max(\ell(\vec{\sigma}) + 1, \ell(\vec{\tau}))$
where $\ell(\vec{\tau}) = \max(\ell(\tau_1), \ldots, \ell(\tau_n))$.

Definition 4.15. We call "iteration rank" of a LOOP$^\omega$ program p the maximum level of the types of mutable variables which occur inside a loop of p.

Definition 4.16. The fragment LOOP^n of LOOP$^\omega$ is defined as the set of programs with iteration rank less than n.

LEMMA 4.17. *For any imperative types $\vec{\tau}$, $\partial((\vec{\tau}^\star)^\times) = \ell(\vec{\tau})$.*

PROPOSITION 4.18. *For any* LOOP$^\omega$ *program p, the iteration rank of p is the same as the recursion rank of $p^\star$.*

PROOF. Indeed, since for any loop which occurs in p with mutable variables $\vec{z} : \vec{\tau}$ we have:

$$(\mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}})^\star = \mathbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s\}^\star_{\vec{z}})$$

where this recursor has type $(\vec{\tau}^\star)^\times$ and $\partial((\vec{\tau}^\star)^\times) = \ell(\vec{\tau})$ by lemma 4.17. □

Remark 4.19. A function computable in the fragment LOOP^0 is thus computable in T_0, and is thus primitive recursive. Consequently, adding only higher-order procedures (without procedural variables) to the LOOP language does not increase its expressive power.

In the next section, we shall be interested in LOOP$^\omega$ programs which are translated into terms which contain no pairing and no pattern matching. The following lemma is straightforward (by definition of the translation).

Definition 4.20. We call "unary" a LOOP$^\omega$ program in which all procedures have exactly one **in** parameter and one **out** parameter and all sequences are annotated by exactly one mutable variable.

LEMMA 4.21. *For any unary* LOOP$^\omega$ *program p, the term $p^\star$ contains no product types (no tuples and no pattern matching).*

5. EXPRESSIVENESS

In this section, we show how to represent any functional of Gödel's system T by some LOOP$^\omega$ program. However, since there is no direct counterpart to pattern-matching in LOOP$^\omega$, we consider in this section a variant of System T with explicit pairing function and projections. On the other hand, it is straightforward to encode a (lazy) pair as an anonymous procedure with no input parameter (a thunk).

Definition 5.1. We define $(\sigma * \tau)$ as $\mathit{unit} \to (\sigma \times \tau)$ and call $\mathcal{S}$ the set of types built inductively from *int* using the type constructors $\to$ and $*$.

Definition 5.2. For any terms $t : \sigma$, $u : \tau$ we write $< t, u >: \sigma * \tau$ as an abbreviation for the term $\mathbf{fn}\ () \Rightarrow (t, u)$. Moreover, we define $\pi^1_{\sigma,\tau} : \sigma * \tau \to \sigma$ and $\pi^2_{\sigma,\tau} : \sigma * \tau \to \tau$ as the following abbreviations:

$$\pi^1_{\sigma,\tau}(z) = \mathbf{let}\ (x, y) = z()\ \mathbf{in}\ x \qquad \text{and} \qquad \pi^2_{\sigma,\tau}(z) = \mathbf{let}\ (x, y) = z()\ \mathbf{in}\ y$$

Lemma 5.3. *The following reduction rules are derivable:*

$$\pi^1_{\sigma,\tau} < t, u > \rightsquigarrow^\star t \qquad \pi^2_{\sigma,\tau} < t, u > \rightsquigarrow^\star u$$

Definition 5.4. For any type $\sigma \in \mathcal{S}$ the translation $\sigma^\diamond$ is defined as follows:

— $int^\diamond = int$
— $(\sigma \to \tau)^\diamond = \mathbf{proc}\ (\mathbf{in}\ \sigma^\diamond; \mathbf{out}\ \tau^\diamond)$
— $(\sigma * \tau)^\diamond = \mathbf{proc}\ (\mathbf{out}\ \sigma^\diamond, \tau^\diamond)$

Lemma 5.5. *For any type $\sigma \in \mathcal{S}$, we have $(\sigma^\diamond)^\star = \sigma$.*

We are now ready to translate terms of System T into Loop$^\omega$ programs. More precisely, a term t of type σ is translated into a sequence with exactly one free mutable variable of type $\sigma^\diamond$ for the result. However, we define the translation only for terms that obey some syntactic criterion (in order to obtain unary Loop$^\omega$ programs whenever the source term contains no product types).

Definition 5.6. We call $\mathcal{L}$ the set of terms of System T defined by the following syntax:

$$\begin{array}{rcl} t, u & ::= & x \mid S^n(0) \mid \mathbf{succ}(t) \mid \mathbf{pred}(t) \mid \mathbf{rec}(x, u, \lambda z.\lambda y.t) \\ & \mid & (\lambda x.t\ u) \mid (x\ u) \mid \lambda x.t \mid \pi^1_{\sigma,\tau}(x) \mid \pi^2_{\sigma,\tau}(x) \mid < t, u > \end{array}$$

Remark 5.7. Clearly, terms of $\mathcal{L}$ can be seen as terms of System T since **succ**, $\pi^1_{\sigma,\tau}$, $\pi^2_{\sigma,\tau}$ and $< t, u >$ are just macro-definitions.

Definition 5.8. For any term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$, the translation $t^\diamond_r$ where r is a fresh variable of type $\sigma^\diamond$ is defined by mutual induction as follows (the types of the terms are written as superscript when required):

— $S^n(0)^\diamond_r = \{r := \bar{n}\}_r$
— $y^\diamond_r = \{r := y\}_r$
— $\mathbf{succ}(u)^\diamond_r = \{u^\diamond_r; \mathbf{inc}(r)\}_r$
— $\mathbf{pred}(u)^\diamond_r = \{u^\diamond_r; \mathbf{dec}(r)\}_r$
— $(\lambda x^\tau.t^\omega)^\diamond_r = \{r := \mathbf{proc}\ (\mathbf{in}\ x : \tau^\diamond; \mathbf{out}\ y : \omega^\diamond)\ t^\diamond_y\}_r$
— $(t^{\tau\to\sigma}\ y)^\diamond_r = \{\mathbf{var}\ p : (\tau \to \sigma)^\diamond\ \{t^\diamond_p; p(y, r)\}_r\}_r$
— $(x^{\tau\to\sigma}\ u)^\diamond_r = \{\mathbf{var}\ y : \tau^\diamond\ \{u^\diamond_y; x(y, r)\}_r\}_r$
— $((\lambda x^\tau.t)\ u)^\diamond_r = \{\mathbf{var}\ z : \tau^\diamond\ \{u^\diamond_z; (\mathbf{proc}\ (\mathbf{in}\ x : \tau^\diamond; \mathbf{out}\ y : \sigma^\diamond)\ t^\diamond_y)\ (z, r)\}_r\}_r$
— $\mathbf{rec}(x, u, \lambda z.\lambda y.t)^\diamond_r = \{u^\diamond_r; \mathbf{for}\ y\ \mathbf{in}\ 1..x\ \{\mathbf{cst}\ z = r\ t^\diamond_r\}_r\}_r$
— $\pi^1_{\sigma,\tau}(z)^\diamond_r = \{\mathbf{var}\ y : \tau^\diamond\ \{z(r, y)\}_r\}_r$
— $\pi^2_{\sigma,\tau}(z)^\diamond_r = \{\mathbf{var}\ x : \sigma^\diamond\ \{z(x, r)\}_r\}_r$
— $(< t, u >)^\diamond_r = \{r := \mathbf{proc}\ (\mathbf{out}\ x : \sigma^\diamond, y : \tau^\diamond)\{t^\diamond_x; u^\diamond_y\}_{x,y}\}_r$

Remark 5.9. Any term of System T (as defined in figure 1) is clearly equivalent to a term of $\mathcal{L}$. Indeed, the requirements that x be an identifier in any sub-term of the form $\pi^1_{\sigma,\tau}(x)$, $\pi^2_{\sigma,\tau}(x)$ or $\mathbf{rec}(x, u, \lambda z.\lambda y.t)$ and that t be either an abstraction or a identifier in any application $(t\ u)$ are easily fulfilled by naming intermediate results when necessary (using a β-redex to simulate a **let**). In Definition 5.8, we tried to reach a compromise between the simplicity of the translation and the effort required to prepare the term before its translation.

Remark 5.10. It is possible (but tedious) to generalize the above translation to arbitrary terms of System T (without restricting them to $\mathcal{L}$). For instance, $\mathbf{rec}(e, u, \lambda z.\lambda y.t)^\diamond_r$ where e is an arbitrary term (not an identifier) can be translated as follows:

$$\mathbf{rec}(e, u, \lambda z.\lambda y.t)^\diamond_r = \{u^\diamond_r; \mathbf{var}\ n : \mathit{int}\ \{e^\diamond_n; \mathbf{for}\ y\ \mathbf{in}\ 1..n\ \{\mathbf{cst}\ z = r\ t^\diamond_r\}_r\}_r\}_r$$

Example 5.11. Recall the definition of Ackermann function given in section 2:

$$\mathit{ack} = \lambda(m, n).(\mathbf{rec}(m, \lambda y.\mathbf{succ}(y), \lambda h.\lambda i.\lambda y.\mathbf{rec}(y, (h\ S(0)), \lambda k.\lambda j.(h\ k)))\ n)$$

By applying the translation with p : **proc** (**in** *int*; **out** *int*) for the result, we get the following sequence for $(\mathit{ack})^\diamond_p$:

```
{
    p:= proc (in m: int, n: int; out r: int) {
        var g : proc (in int; out int) {
            g:= proc (in y: int; out p: int) {
                    p:= y;
                    inc(p);
                }p;
            for i in 1..m {
                cst h = g {
                    g:= proc (in y: int; out p: int) {
                            h(1, p);
                            for j:= 1 to y {
                                cst k = p {
                                    h(k; p);
                                }p;
                            }p;
                        }p;
                }g;
            }g;
            g(n,r);
        }r;
    }r;
}p
```

We obtain thus essentially the imperative version of Ackermann function given in section 4.2 (apart from convenient but non-mandatory constant declarations).

Note that all identifiers of the source term are mapped to read-only variables (hence the introduction of a constant declaration in the translation of a recursor). This property ensures that mutable variables do not occur in the body of procedures in the resulting LOOP$^\omega$ program: the only mutable variables are fresh variables introduced during the translation.

LEMMA 5.12. *Give a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and a fresh mutable variable r of type $\sigma^\diamond$ we have $\Gamma^\diamond; r : \sigma^\diamond \vdash t_r^\diamond : \mathbf{seq}$.*

PROOF. By induction on t. □

For any term $t \in \mathcal{L}$, the fresh mutable variable r in $(t_r^\diamond)$ is always initialized before it is read, and thus r is not free in $(t_r^\diamond)^\star$. This property is stated more formally by the following lemma.

LEMMA 5.13. *Give a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and a fresh mutable variable r of type $\sigma^\diamond$ we have $r \notin FId((t_r^\diamond)^\star)$.*

PROOF. By induction on t. □

Let us write $\twoheadrightarrow$ for the reflexive, transitive and contextual closure (with respect to arbitrary contexts) of the following general $\beta\pi$-reduction rule (where $t_1, \ldots, t_n$ are not necessarily values):

$$\mathbf{let}\ (x_1, \ldots, x_n) = (t_1, \ldots, t_n)\ \mathbf{in}\ u \mapsto u[t_1/x_1, \ldots, t_n/x_n]$$

As expected, the translated $(t_r^\diamond)^\star$ of a term t requires several "administrative" reductions in order to recover the original term.

THEOREM 5.14. *Give a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and a fresh mutable variable r of type $\sigma^\diamond$ we have $(t_r^\diamond)^\star \equiv t$.*

PROOF. By induction on t:

—$(S^n(0)_r^\diamond))^\star = \{r := \bar{n}\}_r^\star = \mathbf{let}\ r = S^n(0)\ \mathbf{in}\ r \twoheadrightarrow S^n(0)$

—$(y_r^\diamond)^\star = \{r := y\}_r^\star = \mathbf{let}\ r = y\ \mathbf{in}\ r \twoheadrightarrow y$

—$(\mathbf{succ}(u)_r^\diamond)^\star$
$= \{u_r^\diamond; \mathbf{inc}(r)\}_r^\star$
$= \mathbf{let}\ r = (u_r^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r = \mathbf{succ}(r)\ \mathbf{in}\ r$
$\twoheadrightarrow \mathbf{let}\ r = \mathbf{succ}((u_r^\diamond)^\star)\ \mathbf{in}\ r$
$\twoheadrightarrow \mathbf{succ}((u_r^\diamond)^\star)$
$\twoheadrightarrow \mathbf{succ}(u)$ by induction hypothesis.

—The case of **pred** is similar to **succ**.

—$(\lambda x^\tau.t^\omega)^\star$
$= \{r := \mathbf{proc}(\mathbf{in}\ x : \tau^\diamond; \mathbf{out}\ y : \omega^\diamond)\ u_y^\diamond\}_r^\star$
$= \mathbf{let}\ r = \mathbf{fn}\ (x : \tau) \Rightarrow (u_y^\diamond)^\star[\delta(\omega)/y]\ \mathbf{in}\ r$
$= \mathbf{let}\ r = \mathbf{fn}\ (x : \tau) \Rightarrow (u_y^\diamond)^\star\ \mathbf{in}\ r$ since $y \notin FId((u_y^\diamond)^\star)$
$\twoheadrightarrow \mathbf{fn}\ (x : \tau) \Rightarrow (u_y^\diamond)^\star$
$\twoheadrightarrow \lambda x.t$ by induction hypothesis.

—$((t\ y)^{\diamond}_r)^{\star}$
$= \{\mathbf{var}\ p : (\tau \to \sigma)^{\diamond}\ \{t^{\diamond}_p; p(y,r)\}_r\}^{\star}_r$
$=\ \mathbf{let}\ p = (t^{\diamond}_p)^{\star}[\delta(\sigma \to \tau)/p]\ \mathbf{in}\ \mathbf{let}\ r = p\ y\ \mathbf{in}\ r$
$=\ \mathbf{let}\ p = (t^{\diamond}_p)^{\star}\ \mathbf{in}\ \mathbf{let}\ r = p\ y\ \mathbf{in}\ r$ since $p \notin FId((u^{\diamond}_p)^{\star})$
$\twoheadrightarrow (t^{\diamond}_p)^{\star} y$
$=\ (t\ y)$ by induction hypothesis.

—$((x\ u)^{\diamond}_r)^{\star}$
$= \{\mathbf{var}\ y : \tau^{\diamond}\ \{u^{\diamond}_y; x(y,r)\}_r\}^{\star}_r$
$= \mathbf{let}\ y = (u^{\diamond}_y)^{\star}[\delta(\sigma)/y]\ \mathbf{in}\ \mathbf{let}\ r = x\ y\ \mathbf{in}\ r$
$= \mathbf{let}\ y = (u^{\diamond}_y)^{\star}\ \mathbf{in}\ \mathbf{let}\ r = x\ y\ \mathbf{in}\ r$ since $y \notin FId((u^{\diamond}_y)^{\star})$
$\twoheadrightarrow x\ (u^{\diamond}_y)^{\star}$
$=\ (x\ u)$ by induction hypothesis.

—$((\lambda x^{\tau}.t)\ u)^{\diamond}_r)^{\star}$
$= \{\mathbf{var}\ z : \tau^{\diamond}\ \{u^{\diamond}_z; (\mathbf{proc}\ (\mathbf{in}\ x : \tau^{\diamond}; \mathbf{out}\ y : \sigma^{\diamond})\ t^{\diamond}_y)\ (z,r)\}_r\}^{\star}_r$
$= \mathbf{let}\ z = (u^{\diamond}_z)^{\star}[\delta(\sigma)/z]\ \mathbf{in}\ \mathbf{let}\ r = (\mathbf{fn}\ (x : \tau) \Rightarrow (t^{\diamond}_y)^{\star})\ z\ \mathbf{in}\ r$
$= \mathbf{let}\ z = (u^{\diamond}_z)^{\star}\ \mathbf{in}\ \mathbf{let}\ r = (\mathbf{fn}\ (x : \tau) \Rightarrow (t^{\diamond}_y)^{\star})\ z\ \mathbf{in}\ r$ since $z \notin FId((u^{\diamond}_z)^{\star})$
$\twoheadrightarrow (\mathbf{fn}\ (x : \tau) \Rightarrow (t^{\diamond}_y)^{\star})\ (u^{\diamond}_y)^{\star}$
$=\ (\lambda x.t\ u)$ by induction hypothesis.

—$(\mathbf{rec}(x, u, \lambda z.\lambda y.t)^{\diamond}_r)^{\star}$
$= \{u^{\diamond}_r; \mathbf{for}\ y\ \mathbf{in}\ 1..x\ \{\mathbf{cst}\ z = r\ t^{\diamond}_r\}_r\}^{\star}_r$
$=\ \mathbf{let}\ r = (u^{\diamond}_r)^{\star}\ \mathbf{in}\ \mathbf{let}\ r = \mathbf{rec}(x, r, \lambda r.\lambda y.\mathbf{let}\ z = r\ \mathbf{in}\ (t^{\diamond}_r)^{\star})\ \mathbf{in}\ r$
$\twoheadrightarrow \mathbf{let}\ r = \mathbf{rec}(x, (u^{\diamond}_r)^{\star}, \lambda z.\lambda y.\mathbf{let}\ z = z\ \mathbf{in}\ (t^{\diamond}_r)^{\star})\ \mathbf{in}\ r$ since $r \notin FId((t^{\diamond}_r)^{\star})$
$\twoheadrightarrow \mathbf{rec}(x, (u^{\diamond}_r)^{\star}, x, \lambda z.\lambda y.(t^{\diamond}_r)^{\star})$
$=\ \mathbf{rec}(x, u, \lambda z.\lambda y.t)$ by induction hypothesis.

—$(\pi^1_{\sigma,\tau}(z)^{\diamond}_r)^{\star} =$
$= \{\mathbf{var}\ y : \tau^{\diamond}\ \{z(r,y)\}_{r,y}\}^{\star}_r$
$=\ \mathbf{let}\ r = (\mathbf{let}\ (r,y) = z()\ \mathbf{in}\ r)[\delta(\tau)/y]\ \mathbf{in}\ r$
$\twoheadrightarrow \mathbf{let}\ (r,y) = z()\ \mathbf{in}\ r$
$= \pi^1_{\sigma,\tau}(z)$ by definition.

—The case of $\pi^2_{\sigma,\tau}$ is similar to $\pi^1_{\sigma,\tau}$.

—$(< t, u >^{\diamond}_r)^{\star}$
$=\ \{r := \mathbf{proc}(\mathbf{out}\ x : \sigma^{\diamond}, y : \tau^{\diamond})\{t^{\diamond}_x; u^{\diamond}_y\}_{x;y}\}^{\star}_r$
$=\ \mathbf{let}\ r = (\mathbf{fn}\ () \Rightarrow \mathbf{let}\ x = (t^{\diamond}_x)^{\star}\ \mathbf{in}\ \mathbf{let}\ y = (u^{\diamond}_y)^{\star}\ \mathbf{in}\ (x,y))\ \mathbf{in}\ r$
$\twoheadrightarrow\ \mathbf{let}\ r = \mathbf{fn}\ () \Rightarrow ((t^{\diamond}_x)^{\star}, (u^{\diamond}_y)^{\star})\ \mathbf{in}\ r$
$\twoheadrightarrow\ \mathbf{fn}\ () \Rightarrow ((t^{\diamond}_x)^{\star}, (u^{\diamond}_y)^{\star})$
$= < t, u >$ by induction hypothesis.

□

LEMMA 5.15. *For any functional type τ, $\partial(\tau^{\diamond}) = \ell(\tau)$.*

PROPOSITION 5.16. *Give a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and a fresh mutable variable p of type $\sigma^{\diamond}$, the iteration rank of $t^{\diamond}_p$ is the same as the recursion rank of t.*

PROOF. Indeed, any occurrence of a recursor type τ in t is translated as follows:

$$\mathbf{rec}(x, u, \lambda z.\lambda y.s)^{\diamond}_r = \{u^{\diamond}_r; \mathbf{for}\ y\ \mathbf{in}\ 1..x\ \{\mathbf{cst}\ z = r\ s^{\diamond}_r\}_r\}_r$$

where r is a mutable variable with type $\tau^\diamond$. We can conclude since r occurs inside a loop and $\partial(\tau^\diamond) = \ell(\tau)$ by lemma 5.15. □

Remark 5.17. Note that a term with no product types (no pairs and no projections) is translated into a unary Loop$^\omega$ program.

6. APPLICATIONS

6.1 Characterizing elementary functions

A first application of the results of this paper is extensional: we give a new characterization of the class of Csillag-Kalmar elementary recursive functions (the class $\mathcal{E}_3$ in Grzegorczyk hierarchy). In [Beckmann and Weiermann 2000], the authors consider a variant of System T with an explicit discriminator $\mathcal{D}$ and an iterator $\mathcal{I}$ instead of a recursor and they prove the following theorem:

Theorem 6.1. *[Beckmann and Weiermann 2000] The set of terms of System T which contain no λ-abstraction of the form $\lambda x \ldots \mathcal{I}(t, \ldots)$ where x occurs in t compute exactly the class of elementary recursive functions.*

It is well-known that both operators $\mathcal{D}$ and $\mathcal{I}$ are special cases of the recursor where $\mathcal{D}(n, a, b) \equiv \mathbf{rec}(n, a, \lambda x.\lambda i.b)$ and $\mathcal{I}(n, b, \lambda y.a) \equiv \mathbf{rec}(n, b, \lambda y.\lambda i.a)$ where x, i are fresh identifiers. On the imperative side, these operators correspond respectively to the standard compound statements **if** ... **then** ... **else** ... and **loop** n $\{\ldots\}$ (where there is no iteration variable available inside the body of the loop).

Given a term t, their restriction amounts to requiring that in any sub-term $\mathcal{I}(u, \ldots)$ of t the free identifiers of u are actually free identifiers of t (input identifiers). Let us now define the imperative counterpart of this restriction.

Definition 6.2. We call "elementary" any unary Loop$^\omega$ program p such that $\Gamma; r : \mathit{int} \vdash p : \mathbf{seq}$ and any loop has the form $\{\mathbf{var}\ n : \mathit{int}\ \{\{s_1\}_n; \mathbf{loop}\ n\ \{s_2\}_z\}_z\}_z$ where the only free read-only variables of s_1 are in Γ.

Remark 6.3. In this definition, the **loop** statement corresponds to a macro-definition for a **for** loop in which the iteration variable does not occur. Note also that **if** ... **then** ... **else** ... statements are allowed without restriction in p.In particular, we obtain that any Loop$^\omega$ program in which any bound of loop is a read-only input variable is elementary.

As a corollary of theorems 4.12 and 5.14 we obtain the following result:

Corollary 6.4. *The elementary unary Loop$^\omega$ programs compute exactly the class of elementary recursive functions.*

Proof. Indeed, let p be and elementary unary Loop$^\omega$ program. For any occurrence of a loop statement in p we have:

$$\begin{aligned}
&\{\mathbf{var}\ n : \mathit{int}\ \{\{s_1\}_n; \mathbf{loop}\ n\ \{s_2\}_z\}_z\}^\star_z \\
&\quad = \mathbf{let}\ z = (\mathbf{let}\ n = \{s_1\}^\star_n\ \mathbf{in\ let}\ z = \mathbf{rec}(n, z, \lambda z.\lambda y.\{s_2\}^\star_z)\ \mathbf{in}\ z)\ \mathbf{in}\ z \\
&\quad \twoheadrightarrow \mathbf{rec}(\{s_1\}^\star_n, z, \lambda z.\lambda y.\{s_2\}^\star_z) = \mathcal{I}(\{s_1\}^\star_n, z, \lambda z.\{s_2\}^\star_z)
\end{aligned}$$

Since the only free read-only variables of $\{s_1\}^\star_n$ are input variables, the term $p^\star$ computes thus an elementary function. Conversely, for any term $t : \sigma$ of System T which obeys Beckmann and Weirmann's restriction, the term $t^\diamond_r$ (where r is a fresh

mutable variable of type $\sigma^\diamond$) is an elementary unary Loop$^\omega$ programs (see remark 5.10). □

Remark 6.5. If x is an identifier in $\mathbf{rec}(x, z, \lambda z.\lambda y.\{s\}^\star_z)$ there is no need for introducing the local variable n in its imperative translation and we obtain thus the following simpler form $\mathbf{loop}\ x\ \{s\}_z$. As a special case, we obtain thus that any Loop$^\omega$ program in which any bound of a loop is an input read-only variable is elementary. Actually, we conjecture that such programs are sufficient to represent any elementary function (a proof of this result would follow step by step the proof given for System T in section 4 of [Beckmann and Weiermann 2000]).

Remark 6.6. For simplicity, we restricted ourselves to unary Loop$^\omega$ programs since [Beckmann and Weiermann 2000] do not consider product types. However, since their result is extensional, any encoding of product types in System T (which does not use the recursor) would allow to generalize the above corollary to non-unary elementary Loop$^\omega$ programs. Such an encoding is the following:

The product $\sigma \times \tau$ is defined by induction on σ and then on τ. Firstly, $int \times int$ is encoded as $int \to int$ where $\langle t, u\rangle_{int\times int} \equiv \lambda b.\mathcal{D}(b, t, u)$ and with $\pi^1_{int\times int}(p) \equiv (p\ 0)$ and $\pi^2_{int\times int}(p) \equiv (p\ S(0))$. Now $int \times (\sigma \to \varsigma)$ is defined as $\sigma \to (int \times \varsigma)$ and $(\sigma \to \varsigma)\times\tau$ as $\sigma \to (\varsigma \times \tau)$. The corresponding pairing functions and projections are summarized below. It is straightforward to prove that $\pi^1\,\langle t, u\rangle = t$ and $\pi^2\,\langle t, u\rangle = u$ for any types σ, τ (note however that the η-rule is required).

$$\begin{array}{lll}
\langle t, u\rangle_{int\times(\sigma\to\varsigma)} & = & \lambda x : \sigma.\, \langle t, (u\ x)\rangle_{int\times\varsigma} \\
\pi^1_{int\times(\sigma\to\varsigma)}(z) & = & \pi^1_{int\times\varsigma}(z\ \delta(\sigma)) \\
\pi^2_{int\times(\sigma\to\varsigma)}(z) & = & \lambda x : \sigma.\pi^2_{int\times\varsigma}(z\ x)
\end{array}
\qquad
\begin{array}{lll}
\langle t, u\rangle_{(\sigma\to\varsigma)\times\tau} & = & \lambda x : \sigma.\, \langle (t\ x), u\rangle_{\varsigma\times\tau} \\
\pi^1_{(\sigma\to\varsigma)\times\tau}(z) & = & \lambda x : \sigma.\pi^1_{\varsigma\times\tau}(z\ x) \\
\pi^2_{(\sigma\to\varsigma)\times\tau}(z) & = & \pi^2_{\varsigma\times\tau}(z\ \delta(\sigma))
\end{array}$$

6.2 The minimum problem

The second application is intensional and is related to the so-called *minimum problem*. In [Colson and Fredholm 1998], the authors proved that in call-by-value System T, any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. As a consequence, they obtain the following result:

Theorem 6.7. *[Colson and Fredholm 1998] There is no term of call-by-value System T which computes the minimum of two natural numbers with time-complexity* $O(min(n, m))$.

Remark 6.8. It is worth noticing that this property depends strongly on the reduction rules used for the recursor (and not only on the strategy). For instance, the property does not hold if we consider the following reduction rule for **rec** (which is derivable in call-by-name, but not in call-by-value):

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

Indeed, since we consider weak reduction and since y may occurs under the scope of a λ-abstraction in t, there no reason that $\mathbf{rec}(n, b, \lambda x.\lambda y.t)$ be the next redex to contract. In fact, one can easily show that call-by-name evaluation can be simulated under call-by-value evaluation using this rule and the usual thunk-based encoding [Hatcliff and Danvy 1997].

Note that [Colson and Fredholm 1998] do not consider with product types nor a predecessor operation. However, as a direct corollary of this theorem and the lock-step simulation, we can still obtain:

COROLLARY 6.9. *There is no unary* LOOP$^\omega$ *program without* **dec** *which computes the minimum of two natural numbers n,m with time-complexity* $O(min(n, m))$.

In order to generalize this result to arbitrary LOOP$^\omega$ programs, we need to extend first Colson and Fredholm's theorem. Although their proof technique seems to apply to product types, we did not succeed in extending it with a one-step predecessor operation. However, we present a direct proof of the result for our variant of System T in Appendix A and then we get the expected corollary.

COROLLARY 6.10. *There is no* LOOP$^\omega$ *program which computes the minimum of two natural numbers n,m with time-complexity* $O(min(n, m))$.

REFERENCES

APPEL, A. W. 1998. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK.

AVIGAD, J. AND FEFERMAN, S. 1998. Gödel's functional ("dialectica") interpretation. In *Handbook of Proof Theory*, S. R. Buss, Ed. Elsevier Science Publishers, Amsterdam, 337–405.

BECKMANN, A. AND WEIERMANN, A. 2000. Characterizing the elementary recursive functions by a fragment of Gödel's T. *Archive for Mathematical Logic V39*, 475–491.

CALUDE, C. 1988. *Theories of computational complexity.* Elsevier Science Inc., New York, NY, USA.

COLSON, L. 1991. About primitive recursive algorithms. *Theoretical Computer Science 83*, 57–69.

COLSON, L. AND FREDHOLM, D. 1998. System T, call-by-value and the minimum problem. *Theor. Comput. Sci. 206,* 1-2, 301–315.

CROLARD, T., LACAS, S., AND VALARCHER, P. 2006. On the Expressive Power of the Loop Language. *Nordic Journal of Computing 13,* 1-2, 46–57.

DAVIS, M. AND WEYUKER, E. 1983. *Computability, Complexity and Languages.* Academic Press.

DOD. 1980. *The Programming Language Ada. Reference Manual.* Springer, Berlin.

DONAHUE, J. E. 1977. Locations considered unnecessary. *Acta Inf. 8*, 221–242.

FELLEISEN, M. AND FRIEDMAN, D. P. 1987. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM Press, New York, NY, USA, 314.

FILINSKI, A. 1994. Representing monads. In *POPL.* 446–457.

FILLIÂTRE, J.-C. 2003. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program 13,* 4, 709–745.

FILLIÂTRE, J.-C. AND MARCHÉ, C. 2004. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004.* Lecture Notes in Computer Science, vol. 3308. Springer-Verlag, 15–29.

GELLERICH, W. AND PLÖDEREDER, E. 2001. Parameter-induced aliasing in ada. In *Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference Leuven, Belgium, May 14-18, 2001, Proceedings*, D. Craeynest and A. Strohmeier, Eds. Lecture Notes in Computer Science, vol. 2043. Springer, 88–99.

GIFFORD, D. AND LUCASSEN, J. 1986. Integrating functional and imperative programming. In *ACM Symposium on Principles of Programming Languages.*

GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. 1989. *Proofs and Types.* Vol. 7. Cambridge Tracts in Theorical Comp. Sci.

GÖDEL, K. 1958. Über eine bisher noch nicht benütz teerweiterung des finiten standpunktes. *Dialectica 12*, 280–287.

GÖDEL, K. 1990. *Collected Works, Volume 2.* Oxford University Press, Oxford.

Hatcliff, J. and Danvy, O. 1997. Thunks and the lambda-calculus. *J. Funct. Program. 7,* 3, 303–319.

ISO. 2003. C♯ language specification ISO/IEC 23270.

Kahn, G. 1987. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*. LNCS, vol. 247. Springer-Verlag, 22–39.

Kelsey, R., Clinger, W., and Rees, J. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11,* 1, 7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

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

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML,* Revised edition. MIT Press.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93,* 1, 55–92.

O'Hearn, P. W. and Tennent, R. D., Eds. 1997. *Algol-like Languages*. Birkhaüser.

Peter, R. 1968. *Recursive Functions*. Academic Press.

Plotkin, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University. Sept.

Reynolds, J. C. 1978. Syntactic control of interference. In *POPL*. 39–46.

Reynolds, J. C. 1981. The essence of Algol. In *Algorithmic Languages*, de Bakker and van Vliet, Eds. IFIP, North-Holland, Amsterdam, 345–372.

Reynolds, J. C. 1998. *Theories of programming languages*. Cambridge University Press.

Schütte, K. 1967. *Proof theory*. Addison Wesley.

Stoy, J. 1977. *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT.

Talpin, J.-P. and Jouvelot, P. 1994. The type and effect discipline. *Information and Computation 111,* 2 (June), 245–296.

Turner, D. A. 2004. Total functional programming. *J. UCS 10,* 7, 751–768.

Wadler, P. 1990. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*. Nice, France, 61–78.

Wadler, P. 1998. The marriage of effects and monads. In *International Conference on Functional Programming*. ACM, Baltimore, 63–74.

A. ULTIMATE OBSTINACY

In this appendix, we show that (non-trivial) terms of System T are "ultimately obstinate" (this terminology is borrowed from [Colson 1991]) even in the presence a one-step predecessor operation and product types. However, since the techniques developed in [Colson and Fredholm 1998] do not seem to generalize easily to this case, we present here a direct proof of this result.

We first extend the syntax of terms and values with an infinite number of constants $\bot_i^k$ of type N and we supplement our set of reduction rules with the following rule:

$$C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$$

We denote by $\rightsquigarrow_\bot$ this extended rewriting system. Intuitively, a constant $\bot_i^k$ represents a "sufficiently large" natural number (on which the predecessor can be applied at least k times). The strong normalization of reduction $\rightsquigarrow_\bot^*$ for well-typed terms is a consequence of the following lemma (since the reduction $\rightsquigarrow$ is strongly normalizing for well-typed terms).

LEMMA A.1. *If* $t[\bot_1^{k_1}/x_1, \ldots, \bot_n^{k_n}/x_n] \rightsquigarrow u[\bot_1^{k_1}/x_1, \ldots, \bot_n^{k_n}/x_n]$ *for some terms* t, u *then* $t[0/x_1, \ldots, 0/x_n] \rightsquigarrow u[0/x_1, \ldots, 0/x_n]$.

Since terms now possibly contain constants $\bot_i^k$ of type N, even a well-typed term can get stuck during evaluation (the term cannot be evaluated further) although it is not yet a value. This lemma allows us to split the set of terms into constant, trivial and ultimately obstinate terms.

LEMMA A.2. *Given a well-typed term* t *of type* N *with free variables* $x_1, \ldots, x_n$ *of type* N, *if* $t[\bot_1^0/x_1, \ldots, \bot_n^0/x_n] \rightsquigarrow_\bot^p v$ *where* v *is in normal form then one of the following cases holds:*

—v *is a value* $S^k(0)$ *(we say that* t *is "constant")*

—v *is a value* $S^k(\bot_i^m)$ *with* $m \leqslant p$ *(we say that* t *is "trivial")*

—*there is an evaluation context* $C[\,]$ *such that* $v = C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ *and* $m \leqslant p$ *(we say that* t *is "ultimately obstinate")*

PROOF. By inspection of normal forms (and then by induction on the derivation to show that $m \leqslant p$). □

Definition A.3. The S-substitution $t\,[\![S(\bot_i)/\bot_i]\!]$ is the homomorphic extension of the basic function defined as follows:

—$\bot_i^0\,[\![S(\bot_i)/\bot_i]\!] = S(\bot_i^0)$

—$\bot_i^{k+1}\,[\![S(\bot_i)/\bot_i]\!] = \bot_i^k$

We write $t\,[\![S^p(\bot_i)/\bot_i]\!]$ for the generalized S-substitution defined by induction on p with $t\,[\![S^0(\bot_i)/\bot_i]\!] = t$ and $t\,[\![S^{p+1}(\bot_i)/\bot_i]\!] = t\,[\![S(\bot_i)/\bot_i]\!]\,[\![S^p(\bot_i)/\bot_i]\!]$.

LEMMA A.4. *For any terms* t, u, *if* $t \rightsquigarrow u$ *then* $t\,[\![S(\bot_i)/\bot_i]\!] \rightsquigarrow u\,[\![S(\bot_i)/\bot_i]\!]$

PROOF. Check that S-substitution commutes with the reduction rules given in Figure 1 and with the rule $C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$. □

The following lemma (together with Lemma A.1) allows us to apply Lemma A.2 on genuine natural numbers.

LEMMA A.5. *Given a well-typed term t with constants $\bot_1^0, \ldots, \bot_n^0$ of type N, if $t \rightsquigarrow_\bot v$ then $t[\![S^p(\bot_1)/\bot_1, \ldots, S^p(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot v[\![S^p(\bot_1)/\bot_1, \ldots, S^p(\bot_n)/\bot_n]\!]$*

PROOF. Apply Lemma A.4 repeatedly p times for each $\bot_i$. □

LEMMA A.6. (CONSTANT TERMS) *Given a well-typed constant term $t : N$ with free variables $x_1, \ldots, x_n$ of type N, there exist $k, p \in \mathbb{N}$ such that for any $p_1 \geqslant 0, \ldots, p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$.*

PROOF. Let $t' = t[\bot_1^0/x_1, \ldots, \bot_n^0/x_n]$ and p, k be such that $t' \rightsquigarrow^p S^k(0)$ by Lemma A.2. Since by Lemma A.5, $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(0)$ and since $t[S^{p_1}(\bot_1^0)/x_1, \ldots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!]$ we have $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(0)$. We conclude by Lemma A.1 that $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$. □

LEMMA A.7. (TRIVIAL TERMS) *Given a well-typed trivial term $t : N$ with free variables $x_1, \ldots, x_n$ of type N, there exist i, m, p, k with $1 \leqslant i \leqslant n$ and $m \leqslant p$ such that for any $p_1 \geqslant 0, \ldots, p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^{p_i \dot{-} m + k}(0)$.*

PROOF. Let $t' = t[\bot_1^0/x_1, \ldots, \bot_n^0/x_n]$ and p, k, m be such that $t' \rightsquigarrow^p S^k(\bot_i^m)$ by Lemma A.2. By Lemma A.5, $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(S^{p_i - m}(\bot_i^0))$ if $p_i \geqslant m$, and $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(\bot_i^{m-p_i})$ if $p_i < m$. Since $t[S^{p_1}(\bot_1^0)/x_1, \ldots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!]$, we conclude by Lemma A.1 that $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(S^{p_i \dot{-} m}(0))$. □

Remark A.8. Clearly, trivial terms compute only functions definable (with the usual mathematical notation) as $(x_1, \ldots, x_n) \mapsto (x_i \dot{-} m) + k$ (for some constants m, k).

LEMMA A.9. *Given an evaluation context $C[\,]$, if $C[\mathbf{rec}(S^k(0), v_2, v_3)]$ is well-typed and $k > 0$ then $C[\mathbf{rec}(S^k(0), v_2, v_3)] \rightsquigarrow C'[\mathbf{rec}(S^{k-1}(0), v_2, v_3)]$ where $C'[\,]$ is again an evaluation context.*

PROOF. Indeed, $C' = C[(v_3\ [\,]\ S^k(0))]$. □

THEOREM A.10. (ULTIMATELY OBSTINATE TERMS) *Given an ultimately obstinate term t of type N with free variables $x_1, \ldots, x_n$ of type N, there exist $i, m \in \mathbb{N}$ with $1 \leqslant i \leqslant n$ such that for any $p_1 \geqslant p, \ldots, p_n \geqslant p$, $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n]$ reaches its normal form in at least p_i reductions steps.*

PROOF. If $t' = t[\bot_1^0/x_1 \ldots, \bot_n^0/x_n]$, by Lemma A.2, $t' \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ for some p, m. Now, $t[S^{p_1}(\bot_1^0)/x_1 \ldots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1 \ldots, S^{p_n}(\bot_n)/\bot_n]\!]$ and since $p_i \geqslant m$, $t'[\![S^{p_1}(\bot_1)/\bot_1 \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^{p_i - m}, v_2, v_3)]$ by Lemma A.5. We have then $t[S^{p_1}(0)/x_1 \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p C[\mathbf{rec}(S^{p_i - m}(0), v_2, v_3)]$ by Lemma A.1. Finally $C[\mathbf{rec}(S^{p_i - m}(0), v_2, v_3)] \rightsquigarrow^{p_i - m} C[\mathbf{rec}(0, v_2, v_3)]$ by Lemma A.9 and we can conclude since $p \geqslant m$ and thus $p + p_i - m \geqslant p_i$. □

B. SUBJECT REDUCTION FOR LOOP$^\omega$

B.1 Substitution and renaming

Recall that renaming is only defined for mutable variables and that the imperative substitution is only defined for read-only variables. Since we consider commands up to α-conversion, we may assume that no variable capture can occur.

Definition B.1. The renaming meta-operation on imperative commands, sequences and expressions, denoted respectively by $c[x \hookleftarrow y]$, $\{s\}_{\vec{z}}[x \hookleftarrow y]$ and $e'[x \hookleftarrow y]$ is defined in Table I.

$x[x \hookleftarrow y]$	$=$	y	
$z[x \hookleftarrow y]$	$=$	z	if $x \neq z$
$w[x \hookleftarrow y]$	$=$	w	
$\{\}_{\vec{z}}[x \hookleftarrow y]$	$=$	$\{\}_{\vec{z}[x \hookleftarrow y]}$	
$\{c; s\}_{\vec{z}}[x \hookleftarrow y]$	$=$	$\{c[x \hookleftarrow y]; s[x \hookleftarrow y]\}_{\vec{z}[x \hookleftarrow y]}$	
$(\textbf{for } x \textbf{ in } 1..e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{for } x \textbf{ in } 1..(e'[x \hookleftarrow y])\ s$	
$(\textbf{for } z \textbf{ in } 1..e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{for } z \textbf{ in } 1..(e'[x \hookleftarrow y])\ (s[x \hookleftarrow y])$	if $x, y \neq z$
$(x := e')[x \hookleftarrow y]$	$=$	$y := (e'[x \hookleftarrow y])$	
$(z := e')[x \hookleftarrow y]$	$=$	$z := (e'[x \hookleftarrow y])$	if $x \neq z$
$(\textbf{inc}(x))[x \hookleftarrow y]$	$=$	$\textbf{inc}(y)$	
$(\textbf{inc}(z))[x \hookleftarrow y]$	$=$	$\textbf{inc}(z)$	if $x \neq z$
$(\textbf{dec}(x))[x \hookleftarrow y]$	$=$	$\textbf{dec}(y)$	
$(\textbf{dec}(z))[x \hookleftarrow y]$	$=$	$\textbf{dec}(z)$	if $x \neq z$
$(e'(\vec{e}'', \vec{y}))[x \hookleftarrow y]$	$=$	$e'[x \hookleftarrow y](\vec{e}''[x \hookleftarrow y], \vec{y}[x \hookleftarrow y])$	
$(\textbf{cst } x = e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{cst } x = (e'[x \hookleftarrow y])\ s$	
$(\textbf{cst } z = e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{cst } z = (e'[x \hookleftarrow y])\ (s[x \hookleftarrow y])$	if $x, y \neq z$
$(\textbf{var } x : \tau := e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{var } x : \tau := (e'[x \hookleftarrow y])\ s$	
$(\textbf{var } z : \tau := e'\ s)[x \hookleftarrow y]$	$=$	$\textbf{var } z : \tau := (e'[x \hookleftarrow y])\ (s[x \hookleftarrow y])$	if $x, y \neq z$

Table I. The renaming meta-operation

Definition B.2. The substitution meta-operation on imperative commands, sequences and expressions, denoted respectively by $c[x \leftarrow e]$, $\{s\}_{\vec{z}}[x \leftarrow e]$ and $e'[x \leftarrow e]$ is defined in Table II.

LEMMA B.3. *If* $\Gamma, x : \tau; \Omega \vdash c : \mathcal{T}$ *and* $\emptyset; \emptyset \vdash e : \tau$ *then* $\Gamma; \Omega \vdash c[x \leftarrow e] : \mathcal{T}$.

PROOF. Straightforward induction on c. □

LEMMA B.4. *If* $\Gamma; x : \tau, \Omega \vdash c : \mathcal{T}$ *then* $\Gamma; y : \tau, \Omega \vdash c[x \hookleftarrow y] : \mathcal{T}$.

PROOF. Straightforward induction on c. □

B.2 Preliminary lemmas

LEMMA B.5. *If* $\Omega \vdash \mu$ *and for all* $\Delta \subset \Omega$, $\emptyset; \Delta \vdash e : \tau$ *and* $e =_\mu w$, *then we have* $\emptyset; \emptyset \vdash w : \tau$.

PROOF. The case $e = w$ is trivial and if e is some variable x then by definition of $\Omega \vdash \mu$, we have $\emptyset; \emptyset \vdash \mu(x) = w : \tau$. □

LEMMA B.6. *If* $\Omega \vdash \mu$, *then for all* $\Gamma \subset \Omega$, *we have* $\Gamma \vdash \mu$.

PROOF. By definition of store typing. □

$x[x \leftarrow e]$	$=$	e	
$y[x \leftarrow e]$	$=$	y	if $x \neq y$
$\bar{q}[x \leftarrow e]$	$=$	$\bar{q}$	
$\{\}_{\vec{z}}[x \leftarrow e]$	$=$	$\{\}_{\vec{z}}$	
$\{c; s\}_{\vec{z}}[x \leftarrow e]$	$=$	$\{c[x \leftarrow e]; s[x \leftarrow e]\}_{\vec{z}}$	
$(\mathbf{for}\ x\ \mathbf{in}\ 1..e'\ s)[x \leftarrow e]$	$=$	$\mathbf{for}\ x\ \mathbf{in}\ 1..(e'[x \leftarrow e])\ s$	
$(\mathbf{for}\ y\ \mathbf{in}\ 1..e'\ s)[x \leftarrow e]$	$=$	$\mathbf{for}\ y\ \mathbf{in}\ 1..(e'[x \leftarrow e])\ (s[x \leftarrow e])$	if $x \neq y$
$(y := e')[x \leftarrow e]$	$=$	$y := (e'[x \leftarrow e])$	
$(\mathbf{inc}(y))[x \leftarrow e]$	$=$	$\mathbf{inc}(y)$	
$(\mathbf{dec}(y))[x \leftarrow e]$	$=$	$\mathbf{dec}(y)$	
$(e'(\vec{e}'', \vec{y}))[x \leftarrow e]$	$=$	$e'[x \leftarrow e](\vec{e}''[x \leftarrow e], \vec{y})$	
$(\mathbf{cst}\ x = e'\ s)[x \leftarrow e]$	$=$	$\mathbf{cst}\ x = (e'[x \leftarrow e])\ s$	
$(\mathbf{cst}\ y = e'\ s)[x \leftarrow e]$	$=$	$\mathbf{cst}\ y = (e'[x \leftarrow e])\ (s[x \leftarrow e])$	if $x \neq y$
$(\mathbf{var}\ x : \tau := e'\ s)[x \leftarrow e]$	$=$	$\mathbf{var}\ x : \tau := (e'[x \leftarrow e])\ s$	
$(\mathbf{var}\ y : \tau := e'\ s)[x \leftarrow e]$	$=$	$\mathbf{var}\ y : \tau := (e'[x \leftarrow e])\ (s[x \leftarrow e])$	if $x \neq y$
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y}, x : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\ s)[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y}, x : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\ s$	
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z}, x : \vec{\tau})\ s)[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z}, x : \vec{\tau})\ s$	
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\ s)[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\ (s[x \leftarrow e])$	if $x \notin \vec{y}, \vec{z}$

Table II. The substitution meta-operation

LEMMA B.7. *If* $\Gamma; \Omega \vdash p : T$ *and* $\Gamma' \supset \Gamma$ *and* $\Omega' \supset \Omega$ *then* $\Gamma'; \Omega' \vdash p : T$.

PROOF. By induction on typing derivation. □

B.3 Proof of Theorem 3.18

THEOREM 3.18. *For any environment* Ω *and any state* (c, μ)*, we have that* $\emptyset; \Omega \vdash (c, \mu)$ *and* $(c, \mu) \mapsto (c', \mu')$ *implies* $\emptyset; \Omega \vdash (c', \mu')$ *and* $dom(\mu) = dom(\mu')$.

PROOF. By induction on the command, and then by cases on the last reduction rule used:

—(S.SEQ-I): we have $\Omega \vdash \mu$ and

$$\frac{\vec{x} \subset \Omega \quad \dfrac{\vec{z} \subset \Omega}{\emptyset; \Omega \vdash \{\}_{\vec{z}} : \mathbf{comm}} \quad \emptyset; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset; \Omega \vdash \{\{\}_{\vec{z}}; s\}_{\vec{x}} : \mathbf{seq}}$$

then we easily get $\Omega \vdash \mu$ and $\emptyset; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{comm}$.

—(S.VAR-I): we have $\Omega \vdash \mu$ and

$$\frac{\vec{x} \subset \Omega \quad \dfrac{\vec{z} \subset \Omega \quad \emptyset; \Omega \vdash e : \tau \quad \dfrac{(\vec{z}, y) \subset (\Omega_{|\vec{z}}, y : \tau)}{\emptyset; y : \tau, \Omega_{|\vec{z}} \vdash \{\}_{\vec{z}} : \mathbf{seq}}}{\emptyset; \Omega \vdash \mathbf{var}\ y : \tau := e\ \{\}_{\vec{z}} : \mathbf{comm}} \quad \emptyset; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset; \Omega \vdash \{\mathbf{var}\ y : \tau := e\ \{\}_{\vec{z}}; s\}_{\vec{x}} : \mathbf{seq}}$$

then we easily get $\Omega \vdash \mu$ and $\emptyset; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}$.

—(S.ASSIGN): we have $\Omega \vdash \mu$ and

$$\frac{\vec{x} \subset \Omega \quad \dfrac{y : \tau \in \Omega \quad \emptyset; \Omega \vdash e : \tau}{\emptyset; \Omega \vdash y := e : \mathbf{comm}} \quad \emptyset; \Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset; \Omega \vdash \{y := e; s\}_{\vec{x}} : \mathbf{seq}}$$

then we get $\emptyset;\Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}$. By Lemma B.5, we have $\emptyset;\emptyset \vdash w : \tau$, and since $y : \tau \in \Omega$ we get $\Omega \vdash \mu[y \leftarrow w]$ and $dom(\mu) = dom(\mu[y \leftarrow w])$.

—(S.SEQ-II): we have $\Omega \vdash \mu$ and

$$\frac{\vec{x} \subset \Omega \quad \emptyset;\Omega \vdash c : \mathbf{comm} \quad \emptyset;\Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset;\Omega \vdash \{c; s\}_{\vec{x}} : \mathbf{seq}}$$

By Lemma B.6 we get $\Omega \vdash \mu$, and by induction hypothesis on $\Omega \vdash (c, \mu)$, we obtain $dom(\mu) = dom(\mu')$ and $\Omega_{|\vec{x}} \vdash (c', \mu')$ which gives us $\emptyset;\Omega \vdash c' : \mathbf{comm}$ and $\Omega \vdash \mu'$. We can build the following typing derivation:

$$\frac{\vec{x} \subset \Omega \quad \emptyset;\Omega \vdash c' : \mathbf{comm} \quad \emptyset;\Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset;\Omega \vdash \{c; s\}_{\vec{x}} : \mathbf{seq}}$$

Since c' is typed with Ω, only the variables in Ω can be modified by the reduction step in μ. For those variables, we already have $\Omega \vdash \mu'$, and since the other one have not been modified, $\Omega \backslash \Omega_{|\vec{x}} \vdash \mu$ implies $\Omega \backslash \Omega_{|\vec{x}} \vdash \mu'$, and finally we get $\Omega \vdash \mu'$.

—(S.VAR-II): we have $\Omega \vdash \mu$ and

$$\frac{\vec{x} \subset \Omega \qquad \emptyset;\Omega \vdash e : \tau \quad \emptyset; y : \tau, \Omega_{|\vec{x}} \vdash \{s\}_{\vec{x}} : \mathbf{seq}}{\emptyset;\Omega \vdash \mathbf{var}\ y : \tau := e\ \{s\}_{\vec{x}} : \mathbf{comm}}$$

By Lemma B.5, $\emptyset;\Omega \vdash e : \tau$, $\Omega \vdash \mu$ and $e =_\mu w$ implies $\emptyset;\emptyset \vdash w : \tau$. By definition of store typing, $\Omega \vdash \mu$ then implies $\Omega, y : \tau \vdash (\mu, y \leftarrow w)$. By induction hypothesis on $\Omega, y : \tau \vdash (\{s\}_{\vec{x},y}, (\mu, y \leftarrow w))$, we obtain $dom(\mu, y \leftarrow w) = dom(\mu', y \leftarrow w')$ which implies $dom(\mu) = dom(\mu')$, and $\Omega, y : \tau \vdash (\{s'\}_{\vec{x},y}, (\mu, y \leftarrow w'))$ which implies $\emptyset;\Omega \vdash \mathbf{var}\ y : \tau := w'\ \{s'\}_{\vec{x}}$ and $\Omega, y : \tau \vdash (\mu', y \leftarrow w')$. This last assertion trivially implies $\Omega \vdash \mu'$ by definition of store typing, and $\emptyset;\emptyset \vdash w' : \tau$. We can then build the following typing derivation to conclude:

$$\frac{\vec{x} \subset \Omega \qquad \emptyset;\Omega \vdash w' : \tau \quad \emptyset; y : \tau, \Omega_{|\vec{x}} \vdash \{s'\}_{\vec{x},y} : \mathbf{seq}}{\emptyset;\Omega \vdash \mathbf{var}\ y : \tau := w'\ \{s'\}_{\vec{x}} : \mathbf{comm}}$$

—(S.INC): we have $\Omega \vdash \mu$ and

$$\frac{y : \mathbf{int} \in \Omega}{\emptyset;\Omega \vdash \mathbf{inc}(y) : \mathbf{comm}}$$

then we easily get $\Omega \vdash \mu$ and

$$\frac{y : \mathbf{int} \in \Omega \quad \emptyset;\Omega \vdash \overline{q+1} : \mathbf{int}}{\emptyset;\Omega \vdash y := \overline{q+1} : \mathbf{comm}}$$

—(S.DEC): similar to above.

—(S.CALL): we have $\Omega \vdash \mu$ and

$$\frac{\emptyset;\Omega \vdash p : \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau}) \quad \emptyset;\Omega \vdash e_i : \sigma_i \quad r_j : \tau_j \in \Omega}{\emptyset;\Omega \vdash p(\vec{e}, \vec{r}) : \mathbf{comm}}$$

By Lemma B.5, $\emptyset;\Omega \vdash e_i : \sigma_i$, $\Omega \vdash \mu$ and $\vec{e} =_\mu \vec{w}$ implies $\emptyset;\emptyset \vdash w_i : \sigma_i$. Still by Lemma B.5, $\emptyset;\Omega \vdash p : \mathbf{proc}(\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})$, $\Omega \vdash \mu$ and $p =_\mu \mathbf{proc}(\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}}$ implies $\emptyset;\emptyset \vdash \mathbf{proc}(\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}} : \mathbf{proc}(\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})$, that is

$$\frac{\vec{z} \not= \emptyset \qquad \vec{y} : \vec{\sigma}; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\emptyset;\emptyset \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y} : \vec{\sigma}; \mathbf{out}\ \vec{z} : \vec{\tau})\{s\}_{\vec{z}} : \mathbf{proc}(\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})}$$

By Lemmas B.3 and B.4, $\vec{y} : \vec{\sigma}; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{seq}$ and $\emptyset; \emptyset \vdash w_i : \sigma_i$ implies $\emptyset; \vec{r} : \vec{\tau} \vdash \{s\}_{\vec{z}}[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}] = \{s[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}]\}_{\vec{r}} : \mathbf{seq}$, and since $r_j : \tau_j \in \Omega$, by Lemma B.7 we obtain $\emptyset; \Omega \vdash \{s[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}]\}_{\vec{r}} : \mathbf{seq}$. By Lemma 3.12, we have $\emptyset; \emptyset \vdash \epsilon(\tau_j) : \tau_j$, and since $r_j : \tau_j \in \Omega$, we obtain $\Omega \vdash \mu[\vec{r} \leftarrow \epsilon(\vec{\tau})]$.

—(S.CST): we have $\Omega \vdash \mu$ and

$$\frac{\vec{z} \subset \Omega \quad \emptyset; \Omega \vdash e : \tau \quad y : \tau; \Omega_{|\vec{z}} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{cst}\ y = e\ \{s\}_{\vec{z}} : \mathbf{comm}}$$

We trivially have $\Omega \vdash \mu$. By Lemma B.5, $\emptyset; \Omega \vdash e : \tau$, $\Omega \vdash \mu$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w : \tau$. Finally, by Lemma B.3, $y : \tau; \Omega_{|\vec{z}} \vdash \{s\}_{\vec{z}} : \mathbf{seq}$ and $\emptyset; \emptyset \vdash w : \tau$ implies $\emptyset; \Omega \vdash \{s\}_{\vec{z}}[y \leftarrow w] : \mathbf{seq}$.

—(S.FOR-I): we have $\Omega \vdash \mu$ and

$$\frac{\vec{z} \subset \Omega \qquad \emptyset; \Omega \vdash e : \mathit{int} \quad y : \mathit{int}; \Omega_{|\vec{z}} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}} : \mathbf{comm}}$$

We trivially have $\Omega \vdash \mu$ and $\emptyset; \Omega \vdash \{\}_{\vec{z}} : \mathbf{comm}$.

—(S.FOR-II): we have $\Omega \vdash \mu$ and

$$\frac{\vec{z} \subset \Omega \qquad \emptyset; \Omega \vdash e : \mathit{int} \quad y : \mathit{int}; \Omega_{|\vec{z}} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{for}\ y\ \mathbf{in}\ 1..e\ \{s\}_{\vec{z}} : \mathbf{comm}}$$

We trivially have $\Omega \vdash \mu$ and the following rule is clearly derivable:

$$\frac{\vec{z} \subset \Omega \qquad y : \mathit{int}; \Omega_{|\vec{z}} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\emptyset; \Omega \vdash \{\mathbf{for}\ y\ \mathbf{in}\ 1..\bar{q}\ \{s\}_{\vec{z}}; \mathbf{cst}\ y = \overline{q+1}\ \{s\}_{\vec{z}}\}_{\vec{z}} : \mathbf{seq}}$$

□

C. TRANSLATION $(\ )^\star$ IS TYPE-PRESERVING

C.1 Preliminary lemmas

LEMMA C.1. *For any environments* Γ *and* Δ *and any sequence* $\{s\}_{\vec{z}}$*, if* $\Gamma;\Omega \vdash \{s\}_{\vec{z}} : \mathbf{seq}$ *then* $\vec{z} \subset \Omega$*.*

PROOF. By case on the last typing rule used. □

LEMMA C.2. *For any environment* Γ*, variable* y*, and terms* t *and* e *of System T, if* $\Gamma, y : \tau \vdash t : \tau'$ *and* $\Gamma, \Omega \vdash e : \tau$ *then* $\Gamma, \Omega \vdash t[e/y] : \tau'$*.*

PROOF. By induction on t. □

LEMMA C.3. *For any environments* Γ *and* Ω*, and any term* t *of System T,* $\Gamma \vdash t : \tau$ *implies* $\Gamma, \Omega \vdash t : \tau$*.*

PROOF. By induction on the typing judgment of t. □

LEMMA C.4. *For any environment* Γ*, variable* y*, type* τ' *and term* t *of System T, if* $y \not\in FId(t)$ *then* $\Gamma, y : \tau' \vdash t : \tau$ *implies* $\Gamma \vdash t : \tau$*.*

PROOF. By induction on the typing judgment of t. □

Notation C.5. For any environment $\Gamma = (x_1 : \tau_1, \ldots, x_n : \tau_n)$, we write $\Gamma^\times$ for $\tau_1 \times \ldots \times \tau_n$.

LEMMA C.6. *For any environment* Γ*,* $dom(\Gamma) = dom(\Gamma^\star)$ *and* $x : \tau \in \Gamma$ *implies* $x : \tau^\star \in \Gamma^\star$*.*

C.2 Proof of Theorem 4.7

THEOREM 4.7. *For any environments* Γ *and* Ω*, any imperative expression* e *and any imperative sequence* $\{s\}_{\vec{x}}$ *we have:*

— $\Gamma;\Omega \vdash e : \tau$ *implies* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$
— $\Gamma;\Omega \vdash \{s\}_{\vec{x}} : \mathbf{seq}$ *implies* $\Gamma^\star, \Omega^\star \vdash \{s\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times$

PROOF. By mutual induction on the typing derivation (and by cases on the translation rule used). Let us begin with expressions.

—(T.NUM)

$$\Gamma;\Omega \vdash \bar{n} : int$$

Indeed:

$$\Gamma^\star, \Omega^\star \vdash S^n(0) : int$$

is clearly derivable.

—(T.VAR)

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

Indeed:

$$\Gamma^\star, \Omega^\star \vdash y : \tau^\star$$

since by Lemma C.6, $y : \tau \in \Gamma, \Omega$ implies $y : \tau^\star \in \Gamma^\star, \Omega^\star$.

—(T.PROC)

$$\frac{\vec{z} \not= \emptyset \qquad \Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{seq}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\vec{y} : \mathbf{in}\ \vec{\sigma}; \vec{z} : \mathbf{out}\ \vec{\tau})\{s\}_{\vec{z}} : \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})}$$

by induction hypothesis on $\Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{seq}$, we obtain $\Gamma^\star, (\vec{y} : \vec{\sigma}^\star), (\vec{z} : \vec{\tau}^\star) \vdash \{s\}^\star_{\vec{z}} : (\vec{\tau}^\star)^\times$, and by Lemma 4.2 $\vdash \delta(\tau_i) : \tau_i^\star$. By Lemma C.3 we have $\Gamma^\star, (\vec{y} : \vec{\sigma}^\star), \Omega^\star \vdash \delta(\tau_i) : \tau_i^\star$. By Lemma C.2, we then get $\Gamma^\star, (\vec{y} : \vec{\sigma}^\star), \Omega^\star \vdash \{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}] : (\vec{\tau}^\star)^\times$. We can then build the following typing derivation :

$$\frac{\Gamma^\star, (\vec{y} : \vec{\sigma}^\star), \Omega^\star \vdash \{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}] : (\vec{\tau}^\star)^\times}{\Gamma^\star, \Omega^\star \vdash \mathbf{fn}\ (\vec{y} : \vec{\sigma}^\star) \Rightarrow \{s\}^\star_{\vec{z}} : (\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times}$$

—(T.SEQ-II)

$$\frac{\vec{x} \subset \Omega \quad \Gamma; \Omega \vdash c : \mathbf{comm} \quad \Gamma; \Omega \vdash \{s_2\}_{\vec{x}} : \mathbf{seq}}{\Gamma; \Omega \vdash \{c; s_2\}_{\vec{x}} : \mathbf{seq}}$$

Recall that the translation of $\{c; s_2\}_{\vec{x}}$ depends on c (see remark 4.5). We continue thus the proof by cases on c using two additional hypothesis : $\vec{x} \subset \Omega$ and $\Gamma^\star, \Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times$ (obtained by induction hypothesis with $\Gamma; \Omega \vdash \{s_2\}_{\vec{x}} : \mathbf{seq}$). For simplicity, we display only the typing derivation of c which completes the typing derivation of the translated sequence.

—(T.SEQ-I)

$$\frac{\vec{x} \subset \Omega}{\Gamma; \Omega \vdash \{\}^\star_{\vec{x}} : \mathbf{seq}}$$

Indeed:

$$\Gamma^\star, \Omega^\star \vdash \vec{x} : (\Omega^\star_{|\vec{x}})^\times$$

since by Lemma C.6, $\vec{x} \subset \Omega$ implies $\vec{x} \subset \Omega^\star$.

—(T.ASSIGN)

$$\frac{y : \tau \in \Omega \quad \Gamma; \Omega \vdash e : \tau}{\Gamma; \Omega \vdash y := e : \mathbf{comm}}$$

By induction hypothesis on $\Gamma; \Omega \vdash e : \tau$, we obtain $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$. By Lemma C.3, we get $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$. We remark that since $y : \tau \in \Omega$, by Lemma C.6, we get $(y : \tau^\star) \in \Omega^\star$ and we have $\Omega^\star, (y : \tau^\star) = \Omega^\star$. We can then build the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star \quad \Gamma^\star, \Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star, \Omega^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.INC)

$$\frac{y : \mathit{int} \in \Omega}{\Gamma; \Omega \vdash \mathbf{inc}(y) : \mathbf{comm}}$$

We remark that since $y : \mathit{int} \in \Omega$, by Lemma C.6, we get $(y : \mathit{int}) \in \Omega^\star$ and we have $\Omega^\star, (y : \mathit{int}) = \Omega^\star$. We can then build the following typing derivation :

$$\frac{\dfrac{\Gamma^\star, \Omega^\star \vdash y : \mathit{int}}{\Gamma^\star, \Omega^\star \vdash \mathbf{succ}(y) : \mathit{int}} \quad \Gamma^\star, \Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star, \Omega^\star \vdash \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.DEC) Similar to the previous case.

—(T.CALL)

$$\frac{\Gamma;\Omega \vdash p : \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau}) \quad \Gamma;\Omega \vdash e_i : \sigma_i \quad z_j : \tau_j \in \Omega}{\Gamma;\Omega \vdash p(\vec{e};\vec{z}) : \mathbf{comm}}$$

By induction hypothesis, we obtain $\Gamma^\star,\Omega^\star \vdash e_i^\star : \sigma_i^\star$ and $\Gamma^\star,\Omega^\star \vdash p^\star : (\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times$. By Lemma C.3, we get $\Gamma^\star,\Omega^\star \vdash e_i^\star : \sigma_i^\star$ and $\Gamma^\star,\Omega^\star \vdash p^\star : (\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times$. Note that since $z_j : \tau_j \in \Omega$, by Lemma C.6, we get $(\vec{z} : \vec{\tau}^\star) \in \Omega^\star$ and then $\Omega^\star, (\vec{z} : (\vec{\tau}^\star)^\times) = \Omega^\star$. We can then build the following typing derivation :

$$\frac{\Gamma^\star,\Omega^\star \vdash p^\star : (\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times \quad \Gamma^\star,\Omega^\star \vdash \vec{e}^\star : \vec{\sigma}^\star}{\Gamma^\star,\Omega^\star \vdash p^\star\ \vec{e}^\star : (\vec{\tau}^\star)^\times}$$

and finally

$$\frac{\Gamma^\star,\Omega^\star \vdash p^\star\ \vec{e}^\star : (\vec{\tau}^\star)^\times \quad \Gamma^\star,\Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star,\Omega^\star \vdash \mathbf{let}\ \vec{z} = p^\star\ \vec{e}^\star\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.COMM)

$$\frac{\vec{z} \subset \Omega \quad \Gamma;\Omega_{|\vec{z}} \vdash \{s_1\}_{\vec{z}} : \mathbf{seq}}{\Gamma;\Omega \vdash \{s_1\}_{\vec{z}} : \mathbf{comm}}$$

By induction hypothesis, we obtain $\Gamma^\star,\Omega^\star \vdash \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times$. By Lemma C.3 we get $\Gamma^\star,\Omega^\star \vdash \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times$. We can then build the following typing derivation :

$$\frac{\Gamma^\star,\Omega^\star \vdash \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times \quad \Gamma^\star,\Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star,\Omega^\star \vdash \mathbf{let}\ \vec{z} = \{s_1\}^\star_{\vec{z}}\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.VAR)

$$\frac{\vec{z} \subset \Omega \quad \Gamma;\Omega \vdash e : \tau \quad \Gamma;y : \tau,\Omega_{|\vec{z}} \vdash \{s_1\}_{\vec{z}} : \mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{var}\ y : \tau := e\ \{s_1\}_{\vec{z}} : \mathbf{comm}}$$

By induction hypothesis, we obtain $\Gamma^\star,\Omega^\star,(y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z},y} : ((\Omega,(y : \tau))^\star_{|\vec{z},y})^\times$ and $\Gamma^\star,\Omega^\star \vdash e^\star : \tau^\star$. By Lemma C.3 we get $\Gamma^\star,\Omega^\star,(y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z},y} : ((\Omega,(y : \tau))^\star_{|\vec{z},y})^\times$ and $\Gamma^\star,\Omega^\star \vdash e^\star : \tau^\star$. By Lemma C.2, we then obtain $\Gamma^\star,\Omega^\star \vdash \{s_1\}^\star_{\vec{z}}[e^\star/y] : ((\Omega,(y : \tau))^\star_{|\vec{z},y})^\times$. Since $(\Omega,(y : \tau))^\star_{|\vec{z},y} = \Omega^\star_{|\vec{z}},(y : \tau^\star)$. We can then build the following typing derivation, using Lemma C.3 on the variable y to correctly type $\{s_2\}^\star_{\vec{x}}$:

$$\frac{\Gamma^\star,\Omega^\star \vdash \{s_1\}^\star_{\vec{z}}[e^\star/y] : (\Omega^\star_{|\vec{z}})^\times \quad \Gamma^\star,\Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star,\Omega^\star \vdash \mathbf{let}\ \vec{z} = \{s_1\}^\star_{\vec{z}}[e^\star/y]\ \mathbf{in}\ \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.CST)

$$\frac{\vec{z} \subset \Omega \quad \Gamma;\Omega \vdash e : \tau \quad \Gamma,y : \tau;\Omega_{|\vec{z}} \vdash \{s_1\}_{\vec{z}} : \mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{cst}\ y = e\ \{s_1\}_{\vec{z}} : \mathbf{comm}}$$

By induction hypothesis, we obtain $\Gamma^\star,\Omega^\star,(y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : ((\Omega)^\star_{|\vec{z}})^\times$ and $\Gamma^\star,\Omega^\star \vdash e^\star : \tau^\star$. By Lemma C.3 we get $\Gamma^\star,\Omega^\star,(y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : ((\Omega)^\star_{|\vec{z}})^\times$

and $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$. We can then build the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star \qquad \Gamma^\star, \Omega^\star, (y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times}{\Gamma^\star, \Omega^\star \vdash \textbf{let } y = e^\star \textbf{ in } \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times}$$

and finally

$$\frac{\Gamma^\star, \Omega^\star \vdash \textbf{let } y = e^\star \textbf{ in } \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times \quad \Gamma^\star, \Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star, \Omega^\star \vdash \textbf{let } \vec{z} = (\textbf{let } y = e^\star \textbf{ in } \{s_1\}^\star_{\vec{z}}) \textbf{ in } \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

—(T.FOR)

$$\frac{\vec{z} \subset \Omega \quad \Gamma; \Omega \vdash e : \tau \quad \Gamma, y : \tau; \Omega_{|\vec{z}} \vdash \{s_1\}_{\vec{z}} : \textbf{seq}}{\Gamma; \Omega \vdash \textbf{for } y \textbf{ in } 1..e \ \{s_1\}_{\vec{z}} : \textbf{comm}}$$

By induction hypothesis, we obtain $\Gamma^\star, \Omega^\star, (y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : ((\Omega)^\star_{|\vec{z}})^\times$ and $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$. By Lemma C.3 we get $\Gamma^\star, \Omega^\star, (y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : ((\Omega)^\star_{|\vec{z}})^\times$ and $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$. We can then build the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star \quad \Gamma^\star, \Omega^\star \vdash \vec{z} : (\Omega^\star_{|\vec{z}})^\times \quad \Gamma^\star, \Omega^\star, (y : \tau^\star) \vdash \{s_1\}^\star_{\vec{z}} : (\Omega^\star_{|\vec{z}})^\times}{\Gamma^\star, \Omega^\star \vdash \textbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s_1\}^\star_{\vec{z}}) : (\Omega^\star_{|\vec{z}})^\times}$$

and finally

$$\frac{\Gamma^\star, \Omega^\star \vdash \textbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s_1\}^\star_{\vec{z}}) : (\Omega^\star_{|\vec{z}})^\times \quad \Gamma^\star, \Omega^\star \vdash \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}{\Gamma^\star, \Omega^\star \vdash \textbf{let } \vec{z} = \textbf{rec}(e^\star, \vec{z}, \lambda\vec{z}.\lambda y.\{s_1\}^\star_{\vec{z}}) \textbf{ in } \{s_2\}^\star_{\vec{x}} : (\Omega^\star_{|\vec{x}})^\times}$$

□

Contents

1 Introduction **1**

2 Gödel System T **4**
2.1 Example: Ackermann function 6

3 The higher-order Loop language **6**
3.1 Syntax 6
3.2 Type system 7
3.3 Natural semantics 9
3.4 Transition semantics 11

4 Lock-step simulation **13**
4.1 Translation 13
4.2 Example: Ackermann function 14
4.3 Simulation theorem 16

5 Expressiveness **19**

6 Applications **24**
6.1 Characterizing elementary functions 24
6.2 The minimum problem 25

A Ultimate Obstinacy **28**

B Subject reduction for Loop$^\omega$ **30**
B.1 Substitution and renaming 30
B.2 Preliminary lemmas 30
B.3 Proof of Theorem 3.18 31

C Translation $(\ )^\star$ is type-preserving **34**
C.1 Preliminary lemmas 34
C.2 Proof of Theorem 4.7 34