Electronic Notes in Theoretical Computer Science 62 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume62.html 21 pages

Comparing Higher-Order Encodings in Logical Frameworks and Tile Logic [1]

Roberto Bruni, [a] Furio Honsell, [b] Marina Lenisa, [b] and Marino Miculan [b]

[a] *Dipartimento di Informatica, Università di Pisa Corso Italia 40, 56125 Pisa, Italy.* bruni@di.unipi.it

[b] *Dipartimento di Matematica e Informatica, Università di Udine Via delle Scienze 206, 33100 Udine, Italy.* honsell,lenisa,miculan@dimi.uniud.it

Abstract

In recent years, logical frameworks and tile logic have been separately proposed by our research groups, respectively in Udine and in Pisa, as suitable metalanguages with higher-order features for encoding and studying nominal calculi. This paper discusses the main features of the two approaches, tracing differences and analogies on the basis of two case studies: late π-calculus and lazy simply typed λ-calculus.

Introduction

A key area of research of the TOSCA project was concerned with the topic of *metalanguages* and *(computational) metamodels*, and in particular with their use as logic and semantic frameworks for the definition and analysis of languages, calculi and programming paradigms.

A metalanguage can be seen as a general specification system which allows for representing, in a uniform setting, a wide range of formal systems, referred to as the *object systems*. More precisely, a metalanguage is composed by two parts: a formal language, the *formalism*, and an informal *encoding methodology*, or *protocol*, which describes how an object system can be represented in the formalism guaranteeing suitable properties. The metalanguage has to be expressive enough to allow for a faithful translation, the *specification*, of all components of the object systems. Peculiarites and idiosincrasies of the object systems must be taken into account, e.g. context-dependent features such as binders, non-standard notions of substitution, fresh name creation, weird side conditions, etc. Still, the formalism must strive for simplicity and it should be implementable on a machine. Indeed, such a general-purpose implementation

[1] Research supported by the MURST Project TOSCA.

can be readily turned into a mechanized proof assistant for any given object system, by simply providing the specification. This is particularly useful in the case of logics and formal systems for dealing with programs and processes properties. We can factorize out most of the common features of the plethora of program and process logics, thus avoiding the daunting task of implementing a specific proof assistant or integrated environment for each of them.

Three research groups of the TOSCA project, in Milano, Pisa and Udine, have been actively involved in the area of metalanguages and metamodels developing and experimenting *cospan-span categories*, *tile logic* (TL) [14, 4] and *logical frameworks* (e.g. LF) [15, 27], respectively. We refer the reader to [13] for a comparison between the two first metamodels. In this paper, we discuss tile logic and logical frameworks. The foundations of both metamodels are by now well established, various tools based on them are available (e.g. [19, 7]), and several case studies concerning widely used calculi and programming paradigms have been carried out.

The two approaches have been developed separately and have focused on different aspects and problems of meta-representation, namely syntactic and proof-theoretical for the Logical Frameworks, and semantical and categorical for tile logic. The two approaches however share many features both from the conceptual and the methodological viewpoint. Thus, in the spirit of the TOSCA project, it is interesting to clarify these connections, allowing for bidirectional technology transfer, inheritance and reuse of techniques.

In this paper we address this issue and try to shed some light on the analogies (but also on the differences) between the two approaches by comparing the encodings in TL and LF of two paradigmatic nominal calculi: λ-calculus and π-calculus. A key technique for their translations in LF and TL is the possibility of exploiting higher-order objects as first-class citizens, so to delegate name-handling issues (name binding, capture avoiding substitution, α-conversion, name hiding, fresh name creation) to the underlying machinery, where these are dealt with automatically, in a standard way. We remark that the choice of the two case studies is instrumental in comparing TL and LF, and not in claiming the novelty of the encoding themselves. In LF this encoding protocol is named *Higher-Order Abstract Syntax* (HOAS), while in TL the analogous idea leads to *Higher-Order Tile Logic* (HOTL). A main difference between the two approaches is that in HOAS the syntax plays a primary rôle, while HOTL is strongly typed and provides the encoding with observational and operational semantics, and with categorical model of proof terms, i.e., HOTL could be called *Higher-Order Abstract Semantics.*

Synopsys. In Section 1 we present the two metalanguages we focus on, namely the Edinburgh Logical Framework (LF) and the Tile Logic (TL). Sections 2 and 3 present the π-calculus and λ-calculus case studies, respectively. For each of them, we briefly present the object system, the formalization both in LF and TL, and we discuss and compare these encoding. Final remarks and directions for future work are in Section 4.

1 The two Metalanguages and Encoding Protocols

1.1 Logical Frameworks based on Type Theory

Type Theories, such as the Edinburgh Logical Framework LF [15, 2] or the Calculus of (Co)Inductive Constructions [19] were especially designed, or can be fruitfully used, as a general logic specification language, i.e. as a Logical Framework (LF). In an LF, we can represent faithfully and uniformly all the relevant concepts of the inferential process in a logical system (syntactic categories, terms, variables, contexts, assertions, axiom schemata, rule schemata, instantiation, tactics, etc.) via the "judgements-as-types, λ-terms-as-proofs" paradigm. The key concept for representing assertions and rules is that of Martin-Löf's *hypothetico-general* judgement [22], which is rendered as a type of the dependent typed λ-calculus of the Logical Framework. The λ-calculus metalanguage of an LF supports Higher-Order Abstract Syntax (HOAS) *à la* Church, i.e., syntax where language constructors may have higher order types. In HOAS, metalanguage variables play the rôle of "object logic" variables so that schemata are represented as λ-abstractions and object-level instantiation amounts to metalanguage β-reduction. Hence, binding operators of the object language are encoded as *higher order* constants, being viewed as ranging over schemata. Thus, most of the machinery needed for handling names, such as capture-avoiding substitution, α-conversion of bound variables, generation of fresh names and *instantiation* of schemata, comes "for free" because it is safely delegated to the metalanguage [28].

Since LF's allow for higher order assertions one can treat on a par axioms and rules, theorems and derived rules, and hence encode also generalized natural deduction systems in the sense of [31].

The LF specification of a formal system is given by a *signature*, i.e. a declaration of typed constants according to the following methodology:

Theory of Syntax *Syntactic categories* of the object language are encoded as type constants; each *syntactic constructor* is encoded as a term constant of the appropriate type.

Theory of Proofs *Judgements* over the object language are encoded as constants of "typed-valued function" kind, and each *rule* is encoded as a term constant of the appropriate type.

Hence, derivations of a given assertion are represented as terms of the type corresponding to the assertion in question, so that checking the correctness of a derivation amounts just to type-checking in the metalanguage. Thus, derivability of an assertion corresponds to the inhabitation of the type corresponding to that assertion, i.e. the existence of a term of that type. It is possible to prove, informally but rigorously, that a formal system is *adequately, faithfully* represented by its encoding. This proof usually exhibit bijective maps between objects of the formal system (terms, formulæ, proofs) and the λ-terms (in canonical form) of the corresponding types in the formalization.

Encodings in LF's often provide the "normative" formalization of the system under consideration. The specification methodology of LF's, in fact, forces the user to make precise all tacit, or informal, conventions, which always accompany any presentation of a system.

Any interactive proof development environment for the type theoretic metalanguage of an LF (e.g. Coq [19], LEGO [29]), can be readily turned into one for a specific logic, once we have defined the corresponding signature. Such a generated editor allows the user to reason "under assumptions" and go about in developing a proof the way mathematicians normally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, often in top-down, goal-directed fashion.

In this paper we will adopt the Edinburgh Logical Framework LF [15]. LF is a system for deriving typing assertions of the shape $\Gamma \vdash P : Q$, whose intended meaning is "in the environment Γ, P is classified by Q". Three kinds of entities are involved, i.e. *terms* (ranged over by M, N), *types* and *typed valued functions* (ranged over by A, B), and *kinds* (ranged over by K). Types are used to classify terms, and kinds are used to classify types and typed valued functions. These entities are defined by the following abstract syntax.

$$M ::= x \mid MN \mid \lambda x : A.M$$

$$A ::= X \mid AM \mid \Pi_{x:A}.B \mid \lambda x : A.B$$

$$K ::= \mathit{Type} \mid \Pi_{x:A}.K.$$

Π is the *dependent type* constructor. Intuitively $\Pi_{x:A}.B(x)$ denotes the type of those functions, f, whose domain is A and whose values belong to a codomain *depending* on the input, i.e. $f(a) \in B(a)$, for all $a \in A$. Hence, $A \to B$ is just notation abbreviating $\Pi_{x:A}.B$, when x does not occur free in B ($x \notin FV(B)$). In the "judgements-as-types" analogy the dependent product type $\Pi_{x:A}.B(x)$ represents the assertion "for all $x \in A$, $B(x)$ holds".

Environments (ranged over by Γ, Δ) are lists of typed variables. A *signature* Σ is a particular environment which represents an object logic. Due to this different rôle of signatures, instead of $\Sigma, \Gamma \vdash P : Q$ we will write $\Gamma \vdash_\Sigma P : Q$; possibly we will drop the index $_\Sigma$ when clear from the context. We will say that a type A is *inhabited* in the environment Γ (and signature Σ) (denoted $\Gamma \vdash_\Sigma \_ : A$) if there exists a term M such that $\Gamma \vdash_\Sigma M : A$.

1.2 Tile Logic

Tile logic is a sequent calculus over a rule-based model (*tile model*) whose sequents, called tiles, have a bidimensional nature: the horizontal dimension is devoted to the *static* structure of system configurations, while the vertical dimension is devoted to the observational *dynamics* of the system. Moreover, the operational rules are designed for *open* configurations, which may interact through their interfaces. In fact, unlike rewriting logic (RL) [23], where rewrite rules can be freely instantiated and contextualized (e.g., if $f(x) \Rightarrow g(x)$ is a rule, then $C[f(t)]$ can be rewritten to $C[g(t)]$ for any context $C[\cdot]$ and any

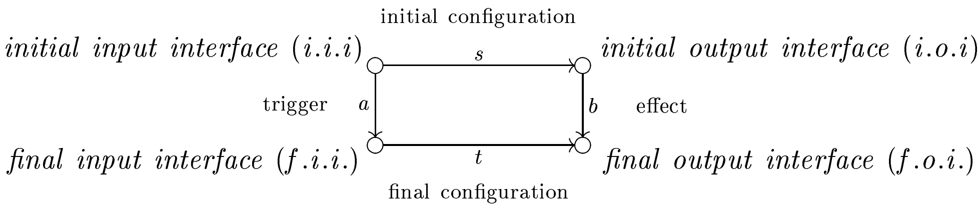

Fig. 1. A tile.

term t), tiles have the form in Figure 1 (whence the name), written as the sequent $s \xrightarrow[b]{a} t$, stating that the *initial configuration* s evolves to the *final configuration* t producing the *effect* b, but such a step is allowed only if the subcomponents of s evolve to the subcomponents of t, producing the *trigger* a (e.g., the tile $f(x) \xrightarrow[b]{a} g(x)$ can be applied to $C[f(t)]$ only if t evolves with effect a and a move of $C[\cdot]$ exists that is triggered by b). Triggers and effects are called *observations* and tile vertices are called *interfaces*.

Formally, a *tile system* is a tuple $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ where $\mathcal{H}$ and $\mathcal{V}$ are monoidal categories with the same set of objects $\mathbf{O}_{\mathcal{H}} = \mathbf{O}_{\mathcal{V}}$, N is the set of rule names and $R\colon N \to \mathbf{A}_{\mathcal{H}} \times \mathbf{A}_{\mathcal{V}} \times \mathbf{A}_{\mathcal{V}} \times \mathbf{A}_{\mathcal{H}}$ associates each name in $x \in N$ with a tile $R(x)\colon s \xrightarrow[b]{a} t$, with $s, t \in \mathcal{H}$ and $a, b \in \mathcal{V}$.

Tiles can be composed horizontally ($\_ * \_$), synchronizing an effect with a trigger; vertically ($\_ \cdot \_$), extending computations of a component; and in parallel ($\_ \otimes \_$), modeling concurrent steps. We say that a tile α is *entailed* by $\mathcal{R}$, written $\mathcal{R} \vdash \alpha$, if α can be obtained by composing basic and auxiliary tiles via $*$, $\cdot$ and $\otimes$. These compositions satisfy the laws of *monoidal double categories* [14, 24]. For $\alpha\colon s \xrightarrow[id_x]{a} id_x$ and $\beta\colon t \xrightarrow[id_y]{b} id_y$ with x the i.i.i. of β, we write $\alpha \triangleleft \beta$ as a shorthand for $(\alpha * 1^t) \cdot \beta$. Similarly, for $\alpha\colon id_x \xrightarrow[a]{id_x} s$ and $\beta\colon id_y \xrightarrow[b]{id_y} t$ with y the f.o.i. of α, we write $\alpha \triangleright \beta$ for $\alpha \cdot (1^s * \beta)$. These compositions are called *diagonal* and yield two monoidal categories.

Though the direction of the arrows in Figure 1 follows the intuitive way of composing states (from left to right) and computations (from top to down), opposites categories can be as well considered, as done e.g. in [10, 11], when this is convenient to model certain operational features, and this impacts in reversing the direction of arrows either in one dimension or in both.

TL extends the ordinary observational equivalences such as trace semantics and bisimilarity by taking $\langle\text{trigger}, \text{effect}\rangle$ pairs as labels, favoring the application of coalgebraic techniques to open systems [12]. The resulting abstract semantics are called *tile trace equivalence* and *tile bisimilarity* (these are families of equivalences indexed by sources and targets of configurations).

It is often convenient to consider tile logics whose categories of configurations and observations are freely generated by suitable signatures Σ_{H} and Σ_{V}. In particular, when the categories of configurations and observations rely on the same algebraic structure (e.g. symmetric monoidal, or cartesian), some auxiliary tiles can be added for free, which guarantee the consistency between the adjoined structure (e.g., symmetries or cartesianity), and the categorical models for such systems must be taken in the corresponding categories of, e.g., symmetric monoidal double categories or cartesian double categories [9, 4].

Fixing the auxiliary tiles and the format of the basic tiles of the system means *fixing a tile format*. Several (first order) tile formats have been e.g. defined which guarantee that tile bisimilarity is a congruence [5] (they usually exploit the format independent *tile decomposition property* [14] to prove the result). Though at present no automated verification tool is available which is based directly on TL, some prototyping has been made possible for suitable TL specification classes, via a conservative encoding in RL, as reported in [7, 8, 6].

The methodology of application of TL can be divided in three steps, which can mutually influence each other's design decisions:

Theory of Configurations Fix the signature of configurations identifying the syntactic constructors and freely adjoin the auxiliary structure needed in the category of configurations;

Theory of Observations Fix the signature of observations (e.g. ordinary LTS's labels are usually seen as unary operators) and freely adjoin the auxiliary structure needed in the category of observations;

Theory of Tiles Define the basic operational rules for the dynamics of system components, and fix the auxiliary tiles to be freely adjoined.

In this paper we focus on the higher-order version of tiles introduced in [10], where $\mathcal{H}$ and $\mathcal{V}$ are cartesian closed, the models are *cartesian closed double categories* (CCDC), and the auxiliary tiles can be conveniently represented as those in the quartet category [2] of the cartesian closed category freely generated by the signature with the same sorts as Σ_{H} and Σ_{V} but empty set of operators. Due to space limitation, we refer to [10] for more details, giving here just the basic syntax and features of HOTL. The first thing to notice is that language constructors can have higher-order types and that λ-abstraction and application are introduced automatically in $\mathcal{H}$ and $\mathcal{V}$. At the level of tiles instead we deal with object abstraction only (configuration and observation abstractions are in fact contra-variant and cannot be correctly typed in the CCDC framework). However, we can pass an argument (e.g. names in nominal calculi) which is bound in the initial configuration to the effect, providing bidimensional scope rules. The possibility of reconciling horizontal abstraction with vertical application is another interesting feature, proper of CCDC's.

Higher-order tiles are presented by typed sentences in the *double λ-notation* of [10]. To deal with the double λ-notation, type judgments $\Gamma \vdash M : \sigma$, where Γ is an environment and σ is the type of the term M in Γ, must take into account that terms are double cells and that their types, called *contour types*,

[2] The quartet category of a category $\mathcal{C}$ is the double category of commuting squares in $\mathcal{C}$.

are cell borders. Thus, a *double signature* is a 4-tuple $\Sigma = \langle \mathbb{B}, \mathbb{H}, \mathbb{V}, \mathbb{C} \rangle$, where $\mathbb{B}$ is the set of *type constants*, $\mathbb{H}$ is the set of *horizontal term constants* $h : \sigma, g : \tau, \ldots$ typed over $\mathbb{B}$ (note that σ, τ can also be higher-order types), $\mathbb{V}$ is the set of *vertical term constants* $v : \sigma, u : \tau, \ldots$ typed over $\mathbb{B}$, and $\mathbb{C}$ is the set of *tile term constants* $\alpha \square \mathbf{s}, \beta \square \mathbf{t}, \ldots$, with $\mathbf{s}$, $\mathbf{t}$, ... *closed contour types.* A generic contour type has the form $H : \tau \xrightarrow[U:\tau\to\sigma]{V:\rho} G : \rho \to \sigma$, where H and G are horizontal terms, while U and V are vertical terms. A contour type is well-typed under Γ if H, G, V and U are well-typed under Γ. A contour type is *closed* if it can be typed by an empty environment, i.e., if H, G, V and U are closed terms. Starting from the constants in Σ, more complex terms are constructed according to the syntax:

$$M ::= \alpha \mid h \mid v \mid x \mid \star \mid \langle M, M \rangle^{\varrho} \mid Proj_i^{\varrho} M \mid \lambda^{\varrho} x{:}\sigma.M \mid M @^{\varrho} M \mid M \varrho M$$

where $\alpha \in \mathbb{C}$, $h \in \mathbb{H}$, $v \in \mathbb{V}$, x is a generic variable, $\star$ is the unit, and pairings, projections, abstractions and applications (the latter denoted by $@$) are feasible in all compositional dimensions of tiles as $\varrho \in \{*, \cdot, \triangleleft, \triangleright\}$ (horizontal, vertical and diagonal, the latter in the two distinct ways previously defined). Moreover, horizontal, vertical and diagonal compositions are also given, because abstraction is restricted to objects, and hence these compositions cannot be expressed just by functional application. A generic type sentence for double λ-terms has the form $\Gamma \vdash M \square H : \tau \xrightarrow[U:\tau\to\sigma]{V:\rho} G : \rho \to \sigma$.

We refer to [10] for the description of the type system, but it is worth remarking that all variables in Γ are propagated by default around the contour of the tiles, so that names in Γ can be used in U and G without being explicitly forwarded by H and V, this makes Γ be a global environment for typing M and its contour type. When a name x in Γ is abstracted from M, then the abstraction consistently binds the occurrences of x in H, V, U and G.

2 π-calculus with Late Semantics

2.1 The object system

In this section we give the syntax of a finitary but significant fragment of the π-calculus, the *late* operational semantics, and the *strong late* bisimilarity equivalence, see [26] for more details.

Let $\mathcal{N}$ be an infinite set of names, ranged over by x, y. The set of *processes* (or *agents*) $\mathcal{P}$, ranged over by P, Q, is defined by the following abstract syntax:

$$P ::= 0 \mid \bar{x}y.P \mid x(y).P \mid \tau.P \mid (\nu x)P \mid P_1|P_2 \mid [x = y]P$$

The *input prefix* operator $x(y)$ and the *restriction* operator (νy) bind the occurrences of y in the argument P. Thus, for each process P we can define the sets of its *free names* $fn(P)$, *bound names* $bn(P)$ and *names* $n(P)$. Agents are

$$\frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}} w \notin fn((\nu z)P) \qquad \text{(IN)}$$

$$\frac{-}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \qquad \text{(OUT)}$$

$$\frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} bn(\mu) \cap fn(Q) = \varnothing \qquad \text{(PAR}_l\text{)}$$

$$\frac{P \xrightarrow{x(z)} P' \quad Q \xrightarrow{\overline{x}y} Q'}{P|Q \xrightarrow{\tau} P'\{y/z\}|Q'} \qquad \text{(COM}_l\text{)}$$

$$\frac{P \xrightarrow{x(w)} P' \quad Q \xrightarrow{\overline{x}(w)} Q'}{P|Q \xrightarrow{\tau} (\nu w)(P'|Q')} \qquad \text{(CLOSE}_l\text{)}$$

$$\frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad \text{(MATCH)}$$

$$\frac{P \xrightarrow{\mu} P'}{(\nu y)P \xrightarrow{\mu} (\nu y)P'} y \notin n(\mu) in Pisa and Udine \qquad \text{(RES)}$$

$$\frac{-}{\tau.P \xrightarrow{\tau} P} \qquad \text{(TAU)}$$

$$\frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} y \neq x, w \notin fn((\nu y)P') \qquad \text{(OPEN)}$$

Fig. 2. Late operational semantics of π-calculus.

taken up-to α-equivalence. For $X \subset \mathcal{N}$, we let $\mathcal{P}_X \triangleq \{P \in \mathcal{P} \mid fn(P) \subseteq X\}$. Capture-avoiding substitution of y in place of x in P is denoted by $P\{y/x\}$.

There is a plethora of slightly different LTS's for the late operational semantics of π-calculus, e.g. [26,25,30]. We present the original one in [26]: the relation $\xrightarrow{\mu}$ is the smallest relation on processes, satisfying the rules in Figure 2 (with the obvious rules PAR_r and COM_r omitted). There are four kinds of actions, ranged over by μ. Action τ is the *silent* move, and $\bar{x}y$ is the *free output*: $P \xrightarrow{\bar{x}y} Q$ means that P can reduce itself to Q emitting y on the channel x. Dually, the *input* $P \xrightarrow{x(z)} Q$ means that P can receive from the channel x any name w and then evolve into $Q\{w/z\}$. The *bound output* $P \xrightarrow{\overline{x}(z)} Q$ means that P can evolve into Q emitting on the channel x a restricted name z of P (name extrusion). The channel x is called the *subject*, while z, y are the *objects*. The τ and free output are *free* actions, the other two are said *bound*. The functions $fn(\cdot)$, $n(\cdot)$ and $bn(\cdot)$ are extended to actions in the obvious way.

Definition 2.1 *A relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ is a* strong late simulation *iff, for all processes P, Q, if $P \ \mathcal{S} \ Q$ then*

(i) *if $P \xrightarrow{\mu} P'$ and μ is free, then for some Q', $Q \xrightarrow{\mu} Q'$ and $P' \ \mathcal{S} \ Q'$;*

(ii) *if $P \xrightarrow{x(y)} P'$ and $y \notin fn(P, Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all $w \in \mathcal{N}$: $P'\{w/y\} \ \mathcal{S} \ Q'\{w/y\}$;*

(iii) *if $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin fn(P, Q)$, then for some Q', $Q \xrightarrow{\overline{x}(y)} Q'$ and $P' \ \mathcal{S} \ Q'$.*

$\mathcal{S}$ is a strong late bisimulation *if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong late simulations. P and Q are* strong late bisimilar,[3] *written $P \dot{\sim} Q$, if a strong late bisimulation*

[3] It is well-known that strong late bisimilarity $\dot{\sim}$ can be defined as the greatest fixed point of a suitable monotonic operator over subsets of $\mathcal{P} \times \mathcal{P}$.

$\mathcal{S}$ exists such that $P \ \mathcal{S} \ Q$.

2.2 Encoding the π-calculus in Logical Framework

In this section we present an encoding of π-calculus in LF, which follows [16], further elaborated in [18] in the Calculus of Inductive Constructions. The corresponding LF signature Σ_π appears in Figure 3.

Syntax. Following the methodology outlined in Section 1.1, for each syntactic category of names, labels and processes we introduce a specific type. Syntactic constructors are defined for labels and processes only: names have no constructor, so that the only terms which can inhabit *Name* are metalevel variables, as usual in weak HOAS encodings [18, 17]. Binders of π-calculus are rendered in LF making use of suitable abstractions. In the case of the input prefix, for instance, in order to obtain a process, both a name (the input channel) and an abstraction (a function from names to processes) are required.

Given a finite set of names X, it is easy to define an encoding function ε_X from π-calculus processes with free names in X, to LF terms of type *Proc*. Most of the π-calculus constructors are translated in the obvious way, but for input prefix $\varepsilon_X(x(y).P) = in_p(x, \lambda y : Name.\varepsilon_{X\cup\{y\}}(P))$ and restrictions: $\varepsilon_X((\nu y)P) = \nu(\lambda y : Name.\varepsilon_{X\cup\{y\}}(P))$. Accordingly, π-calculus processes will be often pretty-printed, following the notation of Section 2.1. This encoding map is an adequate and faithful representation of the syntax of π-calculus:

Proposition 2.2 (Adequacy, I) *Let $X = \{x_1, \ldots, x_n\}$ be a finite set of names. The map ε_X is a bijection between $\mathcal{P}_X$ and the normal forms of type Proc in the signature Σ_π and environment $\Gamma_X \triangleq \{x_1 : Name, \ldots, x_n : Name\}$.*

We omit the proof which follows a standard argument by induction on the syntax of terms and on the derivation of the typing judgement [15].

In view of the remarks above, the set $fn(P)$ for $P \in Proc$ is simply the set of free variables of type *Name*, occurring in the canonical (normal) form of P.

Operational semantics. The transition relation is rendered by two mutually defined inductive predicates $\longmapsto$, $\longmapsto\!\!\!\!\!\rightarrow$, which take care of transitions involving free actions and bound actions, respectively. In the latter case, the result of the transition is not a process but a *process context*, i.e. a process with a hole, conveniently represented by a function $Name \to Proc$.

Rules for $\longmapsto$ and $\longmapsto\!\!\!\!\!\rightarrow$ appear in Figure 3. Some of the rules of Figure 2 have two counterparts, according to whether they refer to free or bound transitions; the bound version of COND has been omitted together with the right versions of PAR, COM and CLOSE. Note that bound names in bound actions (the *objects* of communication) disappear in the encoding. In fact, the rôle played by the objects in bound actions is to denote which name in the resulting process is bound by the action. Since these bound names in processes are denoted by "holes" in the encoding, taking full advantage of the HOAS approach, Syntactic categories and constructors

$$\begin{array}{rlrl}
Name & : Type & 0 & : Proc \\
Label & : Type & \tau_p & : Proc \to Proc \\
Proc & : Type & in_p & : Name \to (Name \to Proc) \to Proc \\
\tau & : Label & out_p & : Name \to Name \to Proc \to Proc \\
out & : Name \to Name \to Label & | & : Proc \to Proc \to Proc \\
in & : Name \to Label & \nu & : (Name \to Proc) \to Proc \\
bout & : Name \to Label & [\ =\ ] & : Name \to Name \to Proc \to Proc
\end{array}$$

Judgements and rules

$$\begin{array}{rl}
\longmapsto & : Proc \to Label \to Proc \to Type \\
\longmapsto\!\!\!\!\rightarrow & : Proc \to Label \to (Name \to Proc) \to Type
\end{array}$$

$$\begin{array}{rl}
TAU & : \Pi_{P:Proc}.\tau_p.P \overset{\tau}{\longmapsto} P \\
OUT & : \Pi_{P:Proc}\Pi_{x,y:Name}.out_p(x,y).P \overset{out(x,y)}{\longmapsto} P \\
IN & : \Pi_{P:Name\to Proc}\Pi_{x:Name}.in_p(x).P \overset{in(x)}{\longmapsto\!\!\!\!\rightarrow} P \\
PAR_l^f & : \Pi_{P,Q,R:Proc}\Pi_{\mu:Label}.P \overset{\mu}{\longmapsto} R \to (P|Q) \overset{\mu}{\longmapsto} (R|Q) \\
PAR_l^b & : \Pi_{P,Q:Proc}\Pi_{R:Name\to Proc}\Pi_{\mu:Name\to Label}. \\
& \quad P \overset{\mu}{\longmapsto\!\!\!\!\rightarrow} R \to (P|Q) \overset{\mu}{\longmapsto\!\!\!\!\rightarrow} \lambda x : Name.\ (Rx)|Q \\
COM_l & : \Pi_{P,Q,R:Proc}\Pi_{S:Name\to Proc}. \\
& \quad \Pi_{x,y:Name}.P \overset{out(x,y)}{\longmapsto} R \to Q \overset{in(x)}{\longmapsto\!\!\!\!\rightarrow} S \to (P|Q) \overset{\tau}{\longmapsto} R|(Sy) \\
RES^f & : \Pi_{P,Q:Name\to Proc}\Pi_{\mu:Label}.\left(\Pi_{x:Name}.(Px) \overset{\mu}{\longmapsto} (Qx)\right) \to \nu(P) \overset{\mu}{\longmapsto} \nu(Q) \\
RES^b & : \Pi_{P:Name\to Proc}\Pi_{Q:Name\to Name\to Proc}\Pi_{\mu:Label}. \\
& \left(\Pi_{x:Name}.(Px) \overset{\mu}{\longmapsto\!\!\!\!\rightarrow} (Qx)\right) \to \nu(P) \overset{\mu}{\longmapsto\!\!\!\!\rightarrow} \lambda z : Name.\ \nu(\lambda x : Name.(Qxz)) \\
OPEN & : \Pi_{P,Q:Name\to Proc}\Pi_{x:Name}.\left(\Pi_{y:Name}.(Py) \overset{in(x)}{\longmapsto} (Qy)\right) \to \nu(P) \overset{bout(x)}{\longmapsto\!\!\!\!\rightarrow} Q \\
CLOSE_l & : \Pi_{P,Q:Proc}\Pi_{R,S:Name\to Proc}\Pi_{x:Name}. \\
& \quad P \overset{in(x)}{\longmapsto\!\!\!\!\rightarrow} R \to Q \overset{bout(x)}{\longmapsto\!\!\!\!\rightarrow} S \to (P|Q) \overset{\tau}{\longmapsto} \nu(\lambda z : Name.(Rz)|(Sz)) \\
COND^f & : \Pi_{P,Q:Proc}\Pi_{\mu:Label}\Pi_{x:Name}.P \overset{\mu}{\longmapsto} Q \to [x{=}x]P \overset{\mu}{\longmapsto} Q
\end{array}$$

Fig. 3. Σ_π, LF signature for the π-calculus.

their representation in the bound action is unnecessary, e.g., the transition $x(y).P \overset{x(y)}{\longmapsto} P$ is represented by $in(x)(\lambda y : Name.\varepsilon(P)) \overset{in(x)}{\longmapsto\!\!\!\!\rightarrow} \lambda y : Name.\varepsilon(P)$.

As an example, consider rule IN, where the target process is an abstraction; when an input occurs (e.g., in the COM$_l$ rule), the target is applied to the effectively received name, so that this name replaces all the occurrences of the placeholder in the target process.

The operational semantics we have presented here allows to eliminate those explicit side-conditions, which are needed in the ordinary semantics to enforce bound names to be fresh, so as to avoid name clashing. The side conditions are implicit in the higher-order nature of the type encoding of the rules in the LF presentation. Bound names remain *bound* also when the transition is performed, since abstractions are used for the target process.

It is possible to establish a precise correspondence between derivations of the semantics of π-calculus and LF judgements derivable in the signature Σ_π.

Proposition 2.3 (Adequacy II) *For any $X \subset_{\text{fin}} \mathcal{N}$, $P, Q \in \mathcal{P}_X$, $x, y \in \mathcal{N}$:*

- $P \xrightarrow{\tau} Q$ *iff* $\Gamma_X \vdash_{\Sigma_\pi} \_ : \varepsilon_X(P) \overset{\tau}{\longmapsto} \varepsilon_X(Q)$;
- $P \xrightarrow{\bar{x}y} Q$ *iff* $\Gamma_X \vdash_{\Sigma_\pi} \_ : \varepsilon_X(P) \overset{out(x,y)}{\longmapsto} \varepsilon_X(Q)$;
- $P \xrightarrow{x(y)} Q$ *iff* $\Gamma_X \vdash_{\Sigma_\pi} \_ : \varepsilon_X(P) \overset{in(x)}{\longmapsto\!\!\!\!\rightarrow} \lambda y : Name.\varepsilon_{X\cup\{y\}}(Q)$;
- $P \xrightarrow{\bar{x}(y)} Q$ *iff* $\Gamma_X \vdash_{\Sigma_\pi} \_ : \varepsilon_X(P) \overset{bout(x)}{\longmapsto\!\!\!\!\rightarrow} \lambda y : Name.\varepsilon_{X\cup\{y\}}(Q)$.

The proof of this proposition is by induction on the structure of derivations ($\Rightarrow$) and on the structure of normal forms ($\Leftarrow$).

The proposition above is *proof-irrelevant*, that is we do not consider the "proofs" corrisponding to the derivations of the transitions. This result can be strenghtened by considering also the structure of derivations of the LTS, instead of the mere transitions. From this proof-theoretical point of view, there is a compositional bijection between derivations of $P \xrightarrow{\tau} Q$ and canonical forms t such that $\Gamma_X \vdash_{\Sigma_\pi} t : \varepsilon_X(P) \overset{\tau}{\longmapsto} \varepsilon_X(Q)$; similarly for the other actions.

Strong late bisimilarity. The strong late bisimilarity can be naïvely encoded by adding to Σ_π the following constants:

$$\begin{aligned}
\dot{\sim} &: Proc \to Proc \to Type \\
\dot{\sim}_{coind} &: \Pi_{P,Q:Proc}.\Pi_{Q:Proc}.\Pi_{R:Proc\to Proc\to Type}. \\
&\quad (\Pi_{P',Q':Proc}.(R\ P'\ Q') \to (T_{\dot{\sim}}\ R\ P'\ Q')) \to (R\ P\ Q) \to (P \mathrel{\dot{\sim}} Q)
\end{aligned}$$

where $T_{\dot{\sim}} : (Proc \to Proc \to Type) \to (Proc \to Proc \to Type)$ in the coinductive rule is suitably defined as the relational operator of strong late bisimulation. However, since this definition needs an universal quantification on the kind $Proc \to Proc \to Type$, it cannot be given in LF but only in stronger theory, such as the Calculus of Constructions (CC).

Another possibility, is to introduce explicitly the type of *propositions*, *Prop*, with their logical connectives and rules for proof derivations. In this way, the quantification needed in $\dot{\sim}_{coind}$ is allowed in LF because it would range over the sort $Proc \to Proc \to Prop$. Actually, this solution would be more adherent to the encoding protocol of LF, since propositions and the logical system are, after all, syntactic components of the object system.

As a final remark, notice that, since type-checking is decidable, every relation which can be adequately represented in a Logical Framework must be

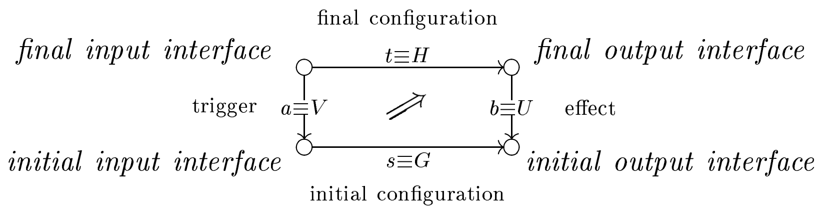

Fig. 4. A "reversed" tile.

semidecidable. This is true for the strong late bisimilarity we are considering in this paper, because we have only finite-state processes; however, it would not hold in the general case of infinite-state processes [18].

2.3 Encoding the π-calculus in Tile Logic

To represent the π-calculus in HOTL we find it convenient to reverse the direction of vertical arrows. Henceforth, according to this choice, tiles must be interpreted as in Figure 4. Indeed this allows us to use the vertical cartesian projections for dealing with name creation (since the computation grows upwards instead of downwards). The idea is then to define a signature for the π-calculus such that each proof of the transition $P \xrightarrow{\mu} Q$ is represented as a tile term $\Gamma \vdash M \square [\![Q]\!] \xrightarrow[{[\![\mu]\!]}]{id} [\![P]\!]$ (for suitable encondings of agents and actions).

The double signature $\mathcal{R}_{late}$ for π-calculus is given in Figure 5. The type constants on which the type system for the π-calculus is defined in the double λ-notation are *Name* and *Proc*. To shorten the notation we will often write n and p in place of *Name* and *Proc* respectively.

The horizontal term constants yielding the syntactic structure of π-agents are very similar to those employed in the LF encoding, but product types are used in some operators to pair arguments which operationally should be provided together (nevertheless, the curryed versions of these operators always exist). The translation of π-agents into horizontal terms (all of type *Proc*) is inductively defined on the left.

$$
\begin{aligned}
[\![0]\!] &= 0\\
[\![x(y).P]\!] &= in_p\ x\ \lambda y : n.[\![P]\!]\\
[\![\tau.P]\!] &= \tau_p\ [\![P]\!]\\
[\![\bar{x}y.P]\!] &= out_p\ \langle x, y\rangle\ [\![P]\!]\\
[\![P_1|P_2]\!] &= [\![P_1]\!]|[\![P_2]\!]\\
[\![(\nu x)P]\!] &= \nu\ \lambda x : n.[\![P]\!]\\
[\![[x = y]P]\!] &= if\ \langle x, y\rangle[\![P]\!]
\end{aligned}
$$

The vertical term constants yielding the observations of π-agents are to some extent a combination of the analogous operators in the LF encoding and the two kinds of transition relations used in that encoding.

From Figure 5 we have omitted the following tile term constants: `ResBout`, which is analogous to `ResIn`; $\texttt{ParOut}_l$, $\texttt{ParBout}_l$ and $\texttt{ParTau}_l$ (analogous to $\texttt{ParIn}_l$); $\texttt{Close}_r$, $\texttt{Com}_r$, $\texttt{ParIn}_r$, $\texttt{ParOut}_r$, $\texttt{ParBout}_r$ and $\texttt{ParTau}_r$ (analogous to their *left* counterparts with subscript $_{-l}$); `CondIn`, `CondOut` and `CondBout` (analogous to `CondTau`). The tile term constants are presented using a more linear notation than the one in Section 1.2, and designed for natural encoding

Interfaces	**Configurations**	**Observations**
n	$0 : p$	$\tau : p \to p$
p	$\tau_p : p \to p$	$out : \langle n, n\rangle \to p \to p$
	$in_p : n \to (n \to p) \to p$	$in : n \to (n \to p) \to p$
	$out_p : \langle n, n\rangle \to p \to p$	$bout : n \to (n \to p) \to p$
	$\mid \, : \langle p, p\rangle \to p$	
	$\nu : (n \to p) \to p$	
	$if : \langle n, n\rangle \to p \to p$	

Tile term constants

$$\begin{array}{rcl}
q : p & \vdash & \texttt{Tau}@^{\triangleleft} q \,\square\, \tau_p \; [q] \;\Rightarrow\; \tau \; [q] \;\, \mathbf{:}\, p \\
x, y : n, q : p & \vdash & \texttt{Output}@^{\triangleleft}\langle x, y\rangle^{\triangleleft}@^{\triangleleft} q \,\square\, out_p \; \langle x, y\rangle[q] \;\Rightarrow\; out \; \langle x, y\rangle[q] \;\, \mathbf{:}\, p \\
x : n, r : n \to p & \vdash & \texttt{Input}@^{\triangleleft} x @^{\triangleleft} r \,\square\, in_p \; x[r] \;\Rightarrow\; in \; x[r] \;\, \mathbf{:}\, p \\
x : n, r : n \to (p \times n) & \vdash & \texttt{Open}@^{\triangleleft} x @^{\triangleleft} r \,\square\, \nu[\lambda y : n.out \; \langle x, \pi_2(ry)\rangle, \pi_1(ry)] \;\Rightarrow\; bout \; x[\lambda y : n.\pi_1(ry)] \;\, \mathbf{:}\, p \\
x, y : n, r : n \to p & \vdash & \texttt{ResOut}@^{\triangleleft} x @^{\triangleleft} r \,\square\, \nu[\lambda z : n.out \; \langle x, y\rangle(rz)] \;\Rightarrow\; out \; \langle x, y\rangle[\nu(\lambda z : n.rz)] \;\, \mathbf{:}\, p \\
x : n, s : n \to n \to p & \vdash & \texttt{ResIn}@^{\triangleleft} x @^{\triangleleft} s \,\square\, \nu[\lambda z : n.in \; x(sz)] \;\Rightarrow\; in \; x[\lambda y : n.\nu(\lambda z : n.szy)] \;\, \mathbf{:}\, p \\
r : n \to p & \vdash & \texttt{ResTau}@^{\triangleleft} r \,\square\, \nu[\lambda z : n.\tau(rz)] \;\Rightarrow\; \tau[\nu r] \;\, \mathbf{:}\, p \\
x : n, r_1, r_2 : n \to p & \vdash & \texttt{Close}_l @^{\triangleleft} x @^{\triangleleft}\langle r_1, r_2\rangle^{\triangleleft} \,\square\, [in \; xr_1]_1 | [bout \; xr_2]_2 \;\Rightarrow\; \tau \; [\nu(\lambda z : n.(r_1 z)|(r_2 z)] \;\, \mathbf{:}\, p \\
x, y : n, r : n \to p, q : p & \vdash & \texttt{Com}_l @^{\triangleleft}\langle x, y\rangle^{\triangleleft}@^{\triangleleft} r @^{\triangleleft} q \,\square\, [in \; xr]_1 | [out \; \langle x, y\rangle q]_2 \;\Rightarrow\; \tau \; [(ry)|q] \;\, \mathbf{:}\, p \\
x : n, r : n \to p, q : p & \vdash & \texttt{ParIn}_l @^{\triangleleft} x @^{\triangleleft} r @^{\triangleleft} q \,\square\, [in \; xr]_1 | [q]_2 \;\Rightarrow\; in \; [\lambda y : n.ry)|q] \;\, \mathbf{:}\, p \\
x : n, q : p & \vdash & \texttt{CondTau}@^{\triangleleft} x @^{\triangleleft} q \,\square\, if \; \langle x, x\rangle[\tau \; q] \;\Rightarrow\; \tau \; [q] \;\, \mathbf{:}\, p
\end{array}$$

Fig. 5. A TL for the late π-calculus.

of interactive calculi. Thus, a contour type $H : \tau \xrightarrow[\lambda y:\tau.U':\tau\to\sigma]{V:\rho} \lambda z : \rho.G' : \rho \to \sigma$ is written $G'[V/z] \;\Rightarrow\; U'[H/y] \;\mathbf{:}\; \sigma$ where the occurrences of H and V performing substitution are written inside square brackets to separate terms that live in orthogonal dimensions. This is consistent with the choice of reversing the direction of vertical arrows as explained above. See also Figure 4 for the interpretation of the symbol $\Rightarrow$. Since τ and ρ can be the product of smaller types, all square brackets surrounding tuple elements are labelled by the element position in the tuple. For example, using infix notation, $([P]_1|[P]_2)|[P]_1$ is a context with three arguments applied to two instances of the same term P, and where the first instance is used twice. This is different from $([P]_1|[P]_2)|[P]_2$, where the second instance of P is used twice. They are both different from $([P]_1|[P]_1)|[P]_1$: In the first two cases the middle interface has two components, but only one in the third case (where the equivalent notation $([P]|[P])|[P]$ can be used). We remark that variables in the global environment Γ can be used without specifying their position in the interface.

To shorten the notation, in Figure 5 tile term constants are typed under non-empty environments, while, e.g., the constant Tau should be presented as

$$\varnothing \vdash \texttt{Tau} \,\square\, \lambda q : p.\tau_{\mathrm{p}} \; ([\lambda q' : p.q']q) \Rightarrow \lambda q : p.\tau \; ([\lambda q' : p.q']q) \;\mathbf{:}\; p \to p$$

This expression is hard to read (and others look still more complicated), but the diagonal application allows simplifying the presentation as in Figure 5.

Tiles Tau, Output, Input move the corresponding action prefix to the effect of the step. The tile Open is triggered by the horizontal abstraction of a tile that produces an ordinary output; the abstraction is w.r.t. the received name y (the name restricted by ν in the initial configuration), thus the produced effect is a bound output. Tiles `ResOut`, `ResIn` and `ResTau` propagate actions through ν when free names in the label are not restricted. Tiles $\texttt{Close}_l$ and $\texttt{Com}_l$ perform synchronous communications on channel x, w.r.t. a bound output and an ordinary output, resp. Tile $\texttt{ParIn}_l$ propagates asynchronous communications and `CondTau` checks guards before propagating actions.

Proposition 2.4 (Adequacy) *If $P \xrightarrow{\mu} Q$ then $fn(P) \vdash M \,\square\, S \xrightarrow[\llbracket\mu\rrbracket]{id} \llbracket P \rrbracket$, where $S = \llbracket Q \rrbracket$ if μ is free, while $S = \lambda y : n.\llbracket Q \rrbracket$ if μ is bound with object y.*

The proofs proceed by induction on the proof of $P \xrightarrow{\mu} Q$. In fact, a denotational mapping can be inductively defined from the proof terms of late transitions and well-typed tile terms.

Proposition 2.4 shows that late transitions of π-calculus can be adequately encoded in $\mathcal{R}_{late}$. Note however that in HOTL more operations are allowed, as e.g. abstracting channel names from proofs and analyzing partially instantiated process (with process variables), thus providing a richer LTS. For this reason, we conjecture that tile bisimilarity for $\mathcal{R}_{late}$ is finer than late bisimilarity (e.g. processes with different sets of free names cannot be tile bisimilar).

2.4 Discussion

Both encodings exploit HOAS for handling names abstraction and application. Comparing the two approaches, there is an evident similarity between the inference rules in LF and the term tile constants in $\mathcal{R}_{late}$. In fact, observations in HOTL combine *Label* constructors with the two kinds of judgements: the predicate $\xmapsto{\tau}$ corresponds to τ; the predicate $\xmapsto{out(x,y)}$ to *out* $\langle x, y\rangle$; the predicates $\overset{in(x)}{\longmapsto\!\!\!\!\rightarrow}$ and $\overset{bout(x)}{\longmapsto\!\!\!\!\rightarrow}$ to *in* x and *bout* x, respectively. To some extent, in HOTL the labels of the LTS become higher-order constructors for transitions, while in LF they are first order entities kept distinct from transition predicates.

HOTL is strongly typed and comes equipped with product types (but these can be coded in LF using the currying operation). Due to the typing rules, it might be necessary to reverse the direction of some arrows (e.g. observations, in the case of π-calculus) for encoding certain operational steps.

Encodings in LF strictly adhere to the syntactical entities of the object system. In particular, there is a 1-1 correspondence between derivations of the operational semantics in the object system and terms inhabiting the type corresponding to the transition judgement. On the other hand, in HOTL the correspondence works in one direction only, i.e. it just provides a semantic domain for transition proofs, where equivalent proofs are identified.

Concerning late bisimilarity, its naïve HOAS encoding require a type theory stronger than LF (e.g., the Calculus of Constructions); otherwise, we can still dwell in LF but a *proposition* type must be introduced. On the other hand, in HOTL late bisimilarity is replaced by the finer tile bisimilarity.

$$\frac{-}{\Gamma \vdash * : \mathtt{unit}} \; (*) \qquad \frac{\Gamma \vdash M : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N : \sigma_1}{\Gamma \vdash MN : \sigma_2} \; (\mathrm{APP})$$

$$\frac{-}{\Gamma \vdash c : \iota} \; (c) \qquad \frac{\Gamma, x : \sigma_1 \vdash M : \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_1.M : \sigma_1 \to \sigma_2} \; (\lambda)$$

$$\frac{-}{\Gamma, x : \sigma \vdash x : \sigma} \; (\mathrm{VAR})$$

Fig. 6. Typing system for the λ-calculus.

$$\frac{-}{c \xrightarrow{c} *} c \in Const \; (\mathrm{CONST}_c) \qquad \frac{\vdash P : \sigma}{\lambda x{:}\sigma.M \xrightarrow{@P} M[P/x]} \; (@)$$

$$\frac{-}{(\lambda x{:}\sigma.M)N \xrightarrow{\tau} M[N/x]} \; (\beta) \qquad \frac{M \xrightarrow{\tau} N}{MP \xrightarrow{\tau} NP} \; (\mathrm{LEFT})$$

Fig. 7. LTS for lazy operational semantics of λ-calculus.

3 The Lazy λ-calculus

3.1 The object system

In this section, we present a simply typed λ-calculus with *lazy operational semantics*. Usually, strategies on λ-calculus (and corresponding observational equivalences) are defined in terms of *reduction systems* [3,1]. Here we give an alternative presentation based on a LTS inspired by [20] (where a call-by-value typed λ-calculus is considered): we define a LTS on closed λ-terms with *weak bisimilarity* yielding *observational (contextual) equivalence* [1].

For simplicitly and for lack of space, we consider a very small language whose syntax of types *Type* and terms Λ is defined as follows:

$$Type \ni \sigma ::= \iota \mid \mathtt{unit} \mid \sigma \to \sigma \qquad \Lambda \ni M ::= c \mid * \mid x \mid MM \mid \lambda x{:}\sigma.M$$

where $x \in Var$, $c \in Const$ for *Const* a finite set of constants. As usual, λ-terms are taken up-to-α-equivalence. The typing system à la Church is in Figure 6. The environment Γ is a partial function from Var to *Type* with finite domain. We denote by Λ^σ_Γ the set of λ-terms typable with σ in Γ, e.g., $\Lambda^\sigma_\varnothing$ denotes the set of closed λ-terms typable with σ. We let $\Lambda^0 \triangleq \bigcup_\sigma \Lambda^\sigma_\varnothing$.

The transition rules of the LTS corresponding to the *lazy leftmost outermost* reduction strategy are in Figure 7. The transition relation $\xrightarrow{\alpha}$, where the label $\alpha \in \{\tau\} \cup \{@P \mid P \in \Lambda^0\} \cup Const$, is defined on $\Lambda^0 \times \Lambda^0$. Notice that the transition relation $\xrightarrow{\tau}$ is exactly the small-step lazy reduction strategy.

Definition 3.1 *Let $\stackrel{\gamma}{\Rightarrow} \subseteq \Lambda^0 \times \Lambda^0$ be defined by $\xrightarrow{\tau^*} \circ \xrightarrow{\gamma} \circ \xrightarrow{\tau^*}$, where $\xrightarrow{\tau^*}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\gamma \neq \tau$. A* weak bisimulation *is a family of symmetric relations $\{R^\sigma \subseteq \Lambda^\sigma_\varnothing \times \Lambda^\sigma_\varnothing\}_\sigma$ indexed over types such that, for all $M, N \in \Lambda^{\sigma_1}_\varnothing$, the following holds if $M \; R^{\sigma_1} \; N$ then for all $M' \in \Lambda^{\sigma_2}_\varnothing$ such that $M \stackrel{\gamma}{\Rightarrow} M'$, there exists $N' \in \Lambda^{\sigma_2}_\varnothing$ such that $N \stackrel{\gamma}{\Rightarrow} N'$ and $M' \; R^{\sigma_2} \; N'$. We let $\{\approx^\sigma_w\}_\sigma$ denote* weak bisimilarity*, i.e. the greatest weak bisimulation.*

Syntactic categories and constructors

$$
\begin{array}{rlrl}
\mathit{SType} : & \mathit{Type} & \mathtt{unit} : & \mathit{SType} \\
\mathit{Label} : & \mathit{Type} & \mathit{arr} : & \mathit{SType} \to \mathit{SType} \to \mathit{SType} \\
\mathit{Const} : & \mathit{Type} & \iota : & \mathit{SType} \\
\mathit{Term} : & \mathit{SType} \to \mathit{Type} & c : & \mathit{Const} \quad \text{(for each } c\text{)} \\
* : & \mathit{Term}(\mathtt{unit}) & \tau : & \mathit{Label} \\
\mathit{in}_T : & \mathit{Const} \to \mathit{Term}(\iota) & \mathit{in}_L : & \mathit{Const} \to \mathit{Label} \\
 & & @ : & \Pi_{\sigma:\mathit{SType}}.\mathit{Term}(\sigma) \to \mathit{Label}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{lam} : & \Pi_{\sigma_1,\sigma_2:\mathit{SType}}.(\mathit{Term}(\sigma_1) \to \mathit{Term}(\sigma_2)) \to \mathit{Term}(\sigma_1 \to \sigma_2) \\
\mathit{app} : & \Pi_{\sigma_1,\sigma_2:\mathit{SType}}.\mathit{Term}(\mathit{arr}(\sigma_1,\sigma_2)) \to \mathit{Term}(\sigma_1) \to \mathit{Term}(\sigma_2)
\end{array}
$$

Judgements and rules

$$
\longmapsto : \Pi_{\sigma_1,\sigma_2:\mathit{SType}}\mathit{Term}(\sigma_1) \to \mathit{Label} \to \mathit{Term}(\sigma_2) \to \mathit{Type}
$$

$$
\begin{array}{rl}
\mathit{CONST} : & \Pi_{c:\mathit{Const}}.\mathit{in}_T(c) \overset{\mathit{in}_L(c)}{\longmapsto} * \\
\beta : & \Pi_{\sigma_1,\sigma_2:\mathit{Stype}}\Pi_{M:\mathit{Term}(\sigma_1)\to \mathit{Term}(\sigma_2)}\Pi_{N:\mathit{Term}(\sigma_1)}. \\
 & \mathit{app}_{\sigma_1,\sigma_2}(\mathit{lam}_{\sigma_1,\sigma_2}(M), N) \overset{\tau}{\longmapsto} (MN) \\
@ : & \Pi_{\sigma_1,\sigma_2:\mathit{Stype}}\Pi_{M:\mathit{Term}(\sigma_1)\to \mathit{Term}(\sigma_2)}\Pi_{P:\mathit{Term}(\sigma_1)}.\mathit{lam}_{\sigma_1,\sigma_2}(M) \overset{@(P)}{\longmapsto} (MP) \\
\mathit{LEFT} : & \Pi_{\sigma_1,\sigma_2:\mathit{Stype}}\Pi_{M,N:\mathit{arr}(\sigma_1,\sigma_2)}\Pi_{P:\sigma_1}. \\
 & M \overset{\tau}{\longmapsto} N \to \mathit{app}_{\sigma_1,\sigma_2}(M,P) \overset{\tau}{\longmapsto} \mathit{app}_{\sigma_1,\sigma_2}(N,P)
\end{array}
$$

Fig. 8. Σ_λ, LF signature for the λ-calculus.

Definition 3.2 *Let the* lazy observational equivalence *be the family* $\{\approx^\sigma\}_\sigma$ *of relations such that, for all* $M, N \in \Lambda^\sigma_\varnothing$,

$$
M \approx^\sigma N \iff \forall C[\,] \in \Lambda^\iota_\varnothing.\forall c\ (C[M] \xrightarrow{\tau^*} c \implies C[N] \xrightarrow{\tau^*} c)\ .
$$

Using the coinductive characterization of $\{\approx^\sigma\}_\sigma$ in terms of the *applicative equivalence* [1], one can easily show that:

Theorem 3.3 *For any type* σ, *we have* $\approx^\sigma_w = \approx^\sigma$.

3.2 Encoding the lazy λ-calculus in Logical Framework

We present an encoding of the λ-calculus with lazy operational semantics in LF. The signature Σ_λ corresponding to λ-calculus is in Figure 8. Our encoding is "full HOAS," i.e. the set of variables does not have a corresponding type in LF, and the operation of substitution is delegated to the metalanguage like in [15,2]. In particular, in rules (β) and (@) the higher order is fully exploited, since term substitution is rendered via the application of the metalanguage.

One can easily define an encoding function ε for types, and a family of encoding functions $\varepsilon^\sigma_\Gamma$ double indexed over types and environments from λ-terms in Λ^σ_Γ, as defined in Section 3.1, to terms of LF in $Term(\sigma)$. In particular, $\varepsilon^{\sigma_1\to\sigma_2}_\Gamma(\lambda x{:}\sigma_1.M) = lam_{\sigma_1,\sigma_2}(\lambda x{:}\varepsilon(\sigma_1).\varepsilon^{\sigma_2}_{\Gamma,x:\sigma_1}(M))$. These encodings are a faithful representation of the syntax of λ-calculus, and a second adequacy result holds for the encoding of the LTS.

Proposition 3.4 (Adequacy I) *There is a bijection $\varepsilon : Type \to SType$ from simple types to terms in SType. Moreover, for each $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and type σ, a bijection $\varepsilon^\sigma_\Gamma$ exists between λ-terms in Λ^σ_Γ and the normal forms of Σ_λ-terms with type $Term(\sigma)$ in the environment $\{x_1 : \varepsilon(\sigma_1), \ldots, x_n : \varepsilon(\sigma_n)\}$.*

Proposition 3.5 (Adequacy II) *Let $M, N \in \Lambda^\sigma_\varnothing$, $M' \in \Lambda^{\sigma'}_\varnothing$, $Q \in \Lambda^{\sigma\to\sigma'}_\varnothing$ and $c \in Const$; then:*

- $M \xrightarrow{c} *$ *iff* $\vdash_{\Sigma_\lambda} \_ : \varepsilon^\sigma_\varnothing(M) \xmapsto{c} *$;
- $M \xrightarrow{\tau} N$ *iff* $\vdash_{\Sigma_\lambda} \_ : \varepsilon^\sigma_\varnothing(M) \xmapsto{\tau} \varepsilon^\sigma_\varnothing(N)$;
- $Q \xrightarrow{@M} M'$ *iff* $\vdash_{\Sigma_\lambda} \_ : \varepsilon^{\sigma\to\sigma'}_\varnothing(Q) \xmapsto{@(\varepsilon^\sigma_\varnothing(M))} \varepsilon^{\sigma'}_\varnothing(M')$.

Likewise π-calculus, this adequacy can be strengthened by exhibiting a compositional bijection between derivations of $P \xrightarrow{\mu} Q$ and canonical forms t such that $\vdash_{\Sigma_\pi} t : \varepsilon(P) \xmapsto{\mu} \varepsilon(Q)$.

Weak bisimilarity and observational equivalence. As for π-calculus, since the weak bisimilarity, and hence also observational equivalence, over the simply typed λ-calculus are decidable, they can be adequately encoded in LF. This does not hold for the untyped λ-calculus, where these relations are not semidecidable. Due to lack of space, we cannot describe these encoding, but they are similar to that of $\dot\sim$ for the π-calculus (Section 2.2).

3.3 Encoding the lazy λ-calculus in Tile Logic

The simply typed λ-calculus over a higher-order signature defines exactly the cartesian closed category of configurations. Thus, from the categorical viewpoint, λ-terms that are equated up to α, β and η rules yield the same abstract configuration in the initial model. Still the vertical dimension is apt for representing applicative transitions, as again labels can be λ-terms. Moreover, the auxiliary tiles of HOTL impose the coherence of horizontal and vertical application, i.e. by letting $eval_{\sigma,\tau} : \tau^\sigma \times \sigma \to \tau$ be the evaluation map, the tile $id \xrightarrow[eval_{\sigma,\tau}]{id} eval_{\sigma,\tau}$ is always present and can be composed with the vertical identity of $f \times a$ where $f : \tau^\sigma$ and $a : \sigma$ yielding $f \times a \xrightarrow[eval_{\sigma,\tau}]{id} fa$, which tells that the application of a function f to an argument a can be observed, resulting in a final configuration where f is applied to a. In order to observe the argument which the function is applied to, we must transfer horizontal constructors to the vertical side. This can be done as in Figure 9, which defines the TL $\mathcal{R}_\lambda$.

Interfaces	Configurations	Observations	Tile term constants
ι	$c : \star \to \iota$	$\hat{c} : \star \to \iota$	$\vdash \mathtt{Dyn}_c \,\square\, \hat{c}[\star] \;\Rightarrow\; c[\star] \;\mathtt{:}\, \iota$
κ	$0 : \star \to \kappa$	$\downarrow_c : \iota \to \kappa$	$\vdash \mathtt{Const}_c \,\square\, \downarrow_c [c] \;\Rightarrow\; 0[\star] \;\mathtt{:}\, \kappa$

Fig. 9. A TL for the applicative λ-calculus (with c ranging over *Const*).

This time tiles are not reversed, e.g., in $\mathtt{Const}_c$, the initial configuration is c, the effect is $\downarrow_c$, the trigger is $\star$ and the final configuration is 0 (and the i.i.i. is empty). Note that, since $\star$ is the terminal object, we cannot introduce an observation $\downarrow_c : \iota \to \star$ because this would collapse $\downarrow_c = \downarrow_{c'}$ for all $c, c' \in \mathit{Const}$ (for each object σ, a unique map exists from σ to $\star$). Thus the type constant κ, and the horizontal term constant 0, are introduced ad-hoc to model the basic observational steps, which are defined by the tiles `Const` (they end in the final configuration 0 instead of $\star$). Applicative steps are instead a combination of tiles $\mathtt{Dyn}_c$ and auxiliary tiles, as the following results show.

Proposition 3.6 *For any closed configuration M, the* TL *$\mathcal{R}_\lambda$ entails the tile $\star \xrightarrow[\hat{M}]{\star} M$, where $\hat{M}$ is the term obtained by the syntactic replacement of all c by $\hat{c}$ (i.e. we have $(\lambda x : \sigma.M)\hat{} \triangleq \lambda x : \sigma.\hat{M}$, $(MN)\hat{} \triangleq \hat{M}\hat{N}$, $\hat{\star} = \star$ and $\hat{x} = x$).*

Corollary 3.7 *For any closed configurations $M \in \Lambda^{\sigma \to \tau}_{\varnothing}$ and $N \in \Lambda^{\sigma}_{\varnothing}$, we have that $\mathcal{R}_\lambda$ entails the tile $M \xrightarrow[\star]{@\hat{N}} MN$, where $@ \triangleq \lambda x : \tau.\lambda f : \sigma \to \tau.fx$.*

Thus τ moves of the applicative transition system are mapped to vertical identities, while the two kinds of labeled steps have their counterparts in the logic. The type system takes care of allowing only correct applications.

Proposition 3.8 *Tile bisimilarity coincides with applicative bisimilarity.*

3.4 Discussion

In the "syntactically-minded" LF approach, each reduction step of λ-calculus corresponds to a rule application; this is evident in the 1-1 correspondence between derivations of the LTS semantics and terms inhabiting the corrisponding type. An advantage of this viewpoint is that the LF encoding can be easily modified to take into account other evaluation strategies, e.g. call-by-value [2]. On the other hand, the semantical approach of TL is somehow more compelling, because it hides β-reductions of the λ-calculus: two β-convertible terms are identified in the same configuration. Therefore, in order to take into account reduction strategies whose equivalences do not correspond to that of CCDC (like call-by-value), we have to adopt different encodings of terms.

In both encodings we have presented here, the object level type system is not explicitly represented, but it is delegated to the strict typing discipline of the metalanguage. More precisely, what we have encoded is an *intrinsic* type system, where only well-typed terms exist. One could also consider *extrinsic* type system or even untyped λ-calculus *tout court*. These object systems can be easily encoded in LF; for instance, the extrinsic type system can be rendered just by adding an explicit typing judgement with its derivation rules. In HOTL, the interpretation of an untyped λ-calculus would need some extra structure in the underlying CCDC, corresponding to the usual construction of *universal objects* in cartesian closed categories.

4 Final Remarks and Conclusions

Logical Frameworks are general formal logic specification languages for representing uniformly the syntactical and inferential peculiarities of arbitrary object logics/systems/calculi. When LF are expanded to a full-blown higher-order dependent constructive type theory, they allow for a smooth encoding of predicates and observational equivalence. Since these semantical encodings rely on the proof strength of the metatheory, they are, in general, weaker than the object language equivalence. On the hand, Logical Frameworks provide straightforwardly the conceptual basis for general proof tools for assisting in rigorous proofs. Summing up, LF's are syntactical/proof theoretic tools.

Tile Logics provide general categorical tools for the specification of space-time aspects of object transition systems. Tile logics can be viewed as a bidimensional/visual/graphical specification system based on a categorical model (CCDC's). Therefore, Tile Logics have a semantical, rather than a proof-theoretic flavour, in that they allow to focus directly on an observational equivalence of the object transition system. Encodings based on Tile Logics are therefore more abstract, and allow to factor out syntactical details, which are present in a Logical Framework approach. Summing up, TL's provide a syntax-free approach to operational semantics, based on transition systems.

There are various similarities between the two metamodels under consideration. The first lies in the treatment of syntax and the encoding of transition rules. Both LF and TL, being based, in effect, on a rich type theory, allow to capitalize on their higher-order features and provide efficient encodings, which delegate to the metalanguage features such as variable freshness, capture-avoiding substitution, α-conversion, name creation.

Sharp differences between the two approaches arise in the context of semantics/observational equivalence. LF is rather rigid here, allowing only to model inferences, whereas TL's are much more abstract, since they force to capture some observational equivalence at the very outset. The semantical aspect is highly intertwined in the very encoding of the proof rules. In general, tile bisimilarity is different from proof equivalence, and suitable tile formats can guarantee that tile bisimilarity is a congruence w.r.t. the language in question. On the other hand, LF's are more transparent and faithful to the object system, in that they allow to encode also formal systems, which are not presented as transition systems. We conjecture that a full equivalence between the two approaches could be reached w.r.t. the encodings of transition systems, where one wants to focus on (observe) just α-equivalence.

The comparison we have started in this work needs further elaboration. Other semantics could be considered on the two object languages we have focused on. For instance, for the λ-calculus we could consider a call-by-value evaluation strategy, whose equivalence is not immediately recovered in the metalanguages. On the π-calculus, it would be interesting to compare the encodings of early semantics [6,21], or reduction-like (i.e., unlabelled) semantics and the associated equivalences (e.g., barbed bisimulation).

Many other object systems could be considered (e.g., ambient, ν-, spi-calculus, etc.). A particularly challenging case is that of languages with polyadic binders, like polyadic π-calculus; their HOAS encoding is not trivial.

Finally, let us point out that while a syntactic encoding of TL in LF looks feasible and we leave it for future work, the semantical encoding of LF in HOTL looks difficult at present, because it requires the treatment of dependent types, not available in CCDC's.

Acknowledgements. We warmfully thanks Ugo Montanari for preliminary discussions and suggestions on part of the topics explored in this paper.

References

[1] Abramsky, S. and C.-H.L. Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993), pp. 159–267.

[2] Avron, A., F. Honsell, I. Mason and R. Pollack, *Using Typed Lambda Calculus to implement formal systems on a machine*, J. Aut. Reas. **9** (1992), pp. 309-354.

[3] Barendregt, H., "The lambda calculus: its syntax and its semantics," Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[4] Bruni, R., "Tile Logic for Synchronized Rewriting of Concurrent Systems," Ph.D. thesis, Computer Science Department, University of Pisa (1999).

[5] Bruni, R., D. de Frutos-Escrig, N. Martí-Oliet and U. Montanari, *Bisimilarity congruences for open terms and term graphs via tile logic*, in: *Proc. CONCUR 2000*, LNCS **1877** (2000), pp. 259–274.

[6] Bruni, R., J. Meseguer and U. Montanari, *Implementing tile systems: Some examples from process calculi*, in: *Proc. ICTCS'98* (1998), pp. 168–179.

[7] Bruni, R., J. Meseguer and U. Montanari, *Internal strategies in a rewriting implementation of tile systems*, in: *Proc. WRLA'98*, ENTCS **15** (1998).

[8] Bruni, R., J. Meseguer and U. Montanari, *Executable tile specifications for process calculi*, in: *Proc. FASE'99*, LNCS **1577** (1999), pp. 60–76.

[9] Bruni, R., J. Meseguer and U. Montanari, *Symmetric monoidal and cartesian double categories as a semantic framework for tile logic*, Math. Struct. in Comput. Sci. (2001), to appear.

[10] Bruni, R. and U. Montanari, *Cartesian closed double categories, their lambda-notation, and the pi-calculus*, in: *Proc. LICS'99* (1999), pp. 246–265.

[11] Bruni, R., U. Montanari and F. Rossi, *An interactive semantics of logic programming*, Theory and Practice of Logic Prog. **1(6)** (2001), pp. 647–690.

[12] Corradini, A., R. Heckel and U. Montanari, *Compositional sos and beyond: A coalgebraic view of open systems*, Theoret. Comput. Sci. (2001), to appear.

[13] Gadducci, F., P. Katis, U. Montanari, N. Sabadini and R.F.C. Walters, *Comparing cospan-spans and tiles via a Hoare-style process calculus*, in: *ENTCS 62* (2002). This volume.

[14] Gadducci, F. and U. Montanari, *The tile model*, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press (2000).

[15] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the ACM **40** (1993), pp. 143–184.

[16] Honsell, F., M. Lenisa, U. Montanari and M. Pistore, *Final semantics for the π-calculus*, in: *Proc. PROCOMET'98* (1998), pp. 225–243.

[17] Honsell, F., M. Miculan and I. Scagnetto, *An axiomatic approach to metareasoning on systems in higher-order abstract syntax*, in: *Proc. ICALP'01*, LNCS **2076** (2001), pp. 963–978.

[18] Honsell, F., M. Miculan and I. Scagnetto, *π-calculus in (co)inductive type theory*, Theoret. Comput. Sci. **253** (2001), pp. 239–285.

[19] INRIA, "The Coq Proof Assistant," (2001). `http://coq.inria.fr/`.

[20] Jeffrey, A. and J. Rathke, *Towards a theory of bisimulation for local names*, in: *Proc. LICS 1999* (1999), pp. 56–66.

[21] Lenisa, M., "Themes in Final Semantics," Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Italy (1998).

[22] Martin-Löf, P., *On the meaning of the logical constants and the justifications of the logic laws*, Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena (1985).

[23] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoret. Comput. Sci. **96** (1992), pp. 73–155.

[24] Meseguer, J. and U. Montanari, *Mapping tile logic into rewriting logic*, in: *Proc. WADT'97*, LNCS **1376** (1998), pp. 62–91.

[25] Milner, R., *The polyadic π-calculus: a tutorial*, in: *Logic and Algebra of Specification*, NATO ASI Series F **94** (1993).

[26] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes*, Inform. and Comput. **100** (1992), pp. 1–77.

[27] Pfenning, F., *The practice of Logical Frameworks*, in: *Proc. CAAP'96*, LNCS **1059** (1996), pp. 119–134.

[28] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proc. ACM SIGPLAN'88* (1988), pp. 199–208.

[29] Pollack, R., "The Theory of LEGO," Ph.D. thesis, Univ. of Edinburgh (1994).

[30] Sangiorgi, D., *A theory of bisimulation for the π-calculus*, Acta Informatica **33** (1996), pp. 69–97.

[31] Schroeder-Heister, P., *A natural extension of natural deduction*, J. Symbolic Logic **49** (1984), pp. 1284–1300.