Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

April 7, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\cong$	CCS weak observational congruence (see Section 11.7.2)
$\approx_d$	CCS dynamic bisimilarity (see Definition 11.17)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\text{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DTMC	Discrete Time Markov Chain (see Definition 14.14)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.5)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.5)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.13)
PI	set of partial injective functions on natural numbers (see Problem 5.12)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.14)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Part III
HOFL: a higher-order functional language

This part focuses on models for sequential computations that are associated to HOFL, a higher-order declarative language that follows the functional style. Chapter 7 presents the syntax, typing and operational semantics of HOFL, while Chapter 9 defines its denotational semantics. The two are related in Chapter 10. Chapter 8 extends the theory presented in Chapter 5 to allow the construction of more complex domains, as needed by the type-constructors available in HOFL.

Chapter 7
Operational Semantics of HOFL

Typing is no substitute for thinking. (Richard Hamming)

Abstract In the previous part of the book we have introduced and studied an imperative language called IMP. In this chapter we move our attention to functional languages. In particular, we introduce HOFL, a simple higher-order functional language which allows for the explicit construction of infinitely many types. We overview Church and Curry type theories. Then, we present a *lazy* operational semantics, which corresponds to a *call-by-name* strategy, namely actual parameters are passed to functions without evaluating them. This view is contrasted with the *eager* evaluation semantics, which corresponds to a *call-by-value* strategy, where all actual parameters are evaluated before being passed to functions. The operational semantics evaluates (well-typed) terms to suitable canonical forms.

7.1 Syntax of HOFL

We start by introducing the plain syntax of HOFL. Then we discuss the type theory and define the well-formed terms. Finally we present the operational semantics of well-formed terms, which reduces terms to their canonical form (when it exists).

In IMP there are only three types: *Aexp* for arithmetic expressions, *Bexp* for boolean expressions and *Com* for commands. Since IMP does not allow to construct other types explicitly, these types are directly embedded in its syntax. HOFL, instead, allows one to define a variety of types, so we first present the grammar for *pre-terms*, then we introduce the concept of typed terms, namely the well-formed sentences of HOFL. Due to the context-sensitive constraints induced by the types, it is possible to see that well-formed terms could not be defined by a syntax expressed in a context-free format. We assume a set of variables *Var* is given.

Definition 7.1 (HOFL: syntax). The following productions define the syntax of HOFL pre-terms:

$$t \quad ::= \quad x \mid n \mid t_0 + t_1 \mid t_0 - t_1 \mid t_0 \times t_1 \mid \textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \mid (t_0, t_1) \mid \textbf{fst}(t) \mid \textbf{snd}(t) \mid \lambda x.\, t \mid (t_0\ t_1) \mid \textbf{rec}\, x.\, t$$

where x is a variable and n an integer.

Besides usual variables x, constants n and arithmetic operators $+, -, \times$, we find: a conditional construct **if** t **then** t_0 **else** t_1 that reads as **if** $t = 0$ **then** t_0 **else** t_1; the constructs for pairing terms (t_0, t_1) and for projecting over the first and second component of a pair $\textbf{fst}(t)$ and $\textbf{snd}(t)$; function abstraction $\lambda x.\, t$ and application $(t_0\ t_1)$; and recursive definition $\textbf{rec}\, x.\, t$. Recursion allows to define recursive terms, namely $\textbf{rec}\, x.\, t$ defines a term t that can contain variable x, which in turn can be replaced by its recursive definition $\textbf{rec}\, x.\, t$.

We call *pre-terms* the terms generated by the syntax above, because it is evident that one could write ill-formed terms, like applying a projection to an integer instead of a pair ($\textbf{fst}(1)$) or summing an integer to a function ($1 + \lambda x.\, x$). To avoid these constructions we introduce the concepts of *type* and *typed term*.

7.1.1 Typed Terms

Definition 7.2 (HOFL types). A HOFL type is a term constructed by using the following grammar:

$$\tau \quad ::= \quad \mathit{int} \quad \mid \quad \tau_0 * \tau_1 \quad \mid \quad \tau_0 \to \tau_1$$

We let $\mathscr{T}$ denote the set of all types.

We allow constant type *int*, the pair type $\tau_0 * \tau_1$ and the function type $\tau_0 \to \tau_1$. Using these productions we can define infinitely many types, like $(\mathit{int} * \mathit{int}) \to \mathit{int}$ for functions that take as argument a pair of integers and return an integer, and $\mathit{int} \to (\mathit{int} * (\mathit{int} \to \mathit{int}))$ for functions that take an integer and return an integer in pair with a function from integers to integers.

Now we define the rule system which allows to say if a pre-term of HOFL is well-formed (i.e., if we can or not associate a type expressed in the above grammar to a given pre-term). The predicates we are interested in are of the form $t : \tau$, expressing that the pre-term t is well-formed and has type τ. We assume variables are typed, i.e., that a function $\widehat{(\cdot)} : \mathit{Var} \to \mathscr{T}$ is given, which assigns a unique type to each variable.

$$\frac{}{x : \widehat{x}}$$

The rule for variables assign to each variable x its type $\widehat{x}$.

$$\frac{}{n : \mathit{int}} \qquad \frac{t_0 : \mathit{int} \quad t_1 : \mathit{int}}{t_0 \text{ op } t_1 : \mathit{int}}\ \text{op} \in \{+, -, \times\} \qquad \frac{t : \mathit{int} \quad t_0 : \tau \quad t_1 : \tau}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 : \tau}$$

The rules for arithmetic expressions assign type *int* to each integer n and to each expression built using $+, -, \times$, whose arguments must be of type *int* too. The rule for conditional expressions **if** t **then** t_0 **else** $t_1 : \tau$ requires the condition t to be of type *int* and the the two branches t_0 and t_1 to have the same type τ, which is also the type of the conditional expression.

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

The rule for pairing says that the type of a term (t_0, t_1) is the pair type $\tau_0 * \tau_1$, where t_i has type τ_i for $i = 0, 1$. Vice versa, for projections it is required that the argument t has pair type $\tau_0 * \tau_1$ for some τ_0 and τ_1, and the result has type τ_0 when the first projection is used or τ_1 when the second projection is used.

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.\, t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

The rule for function abstraction assigns to $\lambda x.\, t$ the functional type $\tau_0 \to \tau_1$, where τ_0 is the type of x and τ_1 is the type of t. In the case of function application $(t_1\ t_0)$, it is required that t_1 has functional type $\tau_0 \to \tau_1$ for some types τ_0 and τ_1, where τ_0 is also the type of t_0. Then, the result has type τ_1.

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.\, t : \tau}$$

The last rule handles recursion: it check that the type τ of the defining expression t is the same as the type of the recursively defined name x; if so, then τ is also the type of the recursive expression **rec** $x.\, t$.

Definition 7.3 (Well-Formed Terms of HOFL). Let t be a pre-term of HOFL, we say that t is *well-formed* if there exists a type τ such that $t : \tau$.

Note that our type system is very simple. Indeed it does not allow to construct useful types, such as recursive, parametric, dependent, polymorphic or abstract types. These limitations imply that we cannot construct many useful terms. For instance, while it is easy to express the types for lists of integer numbers of fixed length (using the type pairing operator $*$) and functions that manipulate them, in our type system lists of integer numbers of variable length are not typable, because some form of recursion should be allowed at the level of types to express them.

7.1.2 Typability and Typechecking

As we said in the last section we will give semantics only to well-formed terms, namely terms which have a type in our type system. Therefore we need an algorithm to say if a term is well-formed. In this section we will present two different solutions to the typability problem, introduced by Church and by Curry, respectively.

7.1.2.1 Church Type Theory

In Church type theory we explicitly associate a type to each variable and deduce the type of each term by structural recursion (i.e., by using the inference rules in a bottom-up fashion).

In this case, we sometimes annotate directly the bounded variables with their type, like in $\lambda x : int.\ x + x$ or **rec** $f : int \to int.\ \lambda x : int.\ fx$.

Example 7.1 (Factorial with Church types). Let $x : int$ and $f : int \to int$ in the pre-term:

$$fact \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1)))$$

So we can type *fact* and all its subterms as below:

$$\dfrac{\dfrac{\widehat{f} = int \to int \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{\dfrac{\widehat{f} = int \to int}{f : int \to int} \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{x - 1 : int}}{f(x-1) : int}}{(x \times (f(x-1))) : int}}{\dfrac{\widehat{x} = int \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{(x \times (f(x-1))) : int}}{\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1))) : int}}}{\dfrac{f : int \to int \quad \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1))) : int \to int}{fact : int \to int}}$$

More concisely, we write:

$$fact \stackrel{\text{def}}{=} \textbf{rec}\ \underset{int \to int}{f}.\lambda\ \underset{int}{x}.\textbf{if}\ \underset{int}{x}\ \textbf{then}\ \underset{int}{1}\ \textbf{else}\ \underset{int}{x} \times (\underset{int \to int}{f}\ (\underbrace{\underset{int}{x} - \underset{int}{1}}_{int}))\ \underbrace{}_{int}\ \underbrace{}_{int}\ \underbrace{}_{int}\ \underbrace{}_{int \to int} : int \to int$$

7.1.2.2 Curry Type Theory

In Curry style, we do not need to explicitly declare the type of each variable. Instead we use the inference rules to calculate type equations (i.e., equations which have types as variables) whose solutions define all the possible type assignments for the term. This means that the result will be a set of types associated to the typed term. The surprising fact is that this set can be represented as all the instances of a single type term with variables, where one instance is obtained by freely replacing each variable with any type. We call this term with variables the *principal type* of the term. This construction is made by using the rules in a goal-oriented fashion, as we have done in Example 7.5.

Example 7.2 (Identity). Let us consider the identity function:

$$\lambda x.\ x$$

By using the type system we have:

$$\lambda x.\ x : \tau \quad \nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1} \quad x : \tau_2$$
$$\nwarrow_{\widehat{x}=\tau_2} \quad \square$$

So we have $\widehat{x} = \tau_1 = \tau_2$ and the principal type of $\lambda x.\ x$ is $\tau_1 \to \tau_1$. Now each solution of the type equation will be an identity function for a specified type. For example if we set $\tau_1 = int$ we have $\tau = int \to int$, but if we set $\tau_1 = int * (int \to int)$ we have $\tau = (int * (int \to int)) \to (int * (int \to int))$.

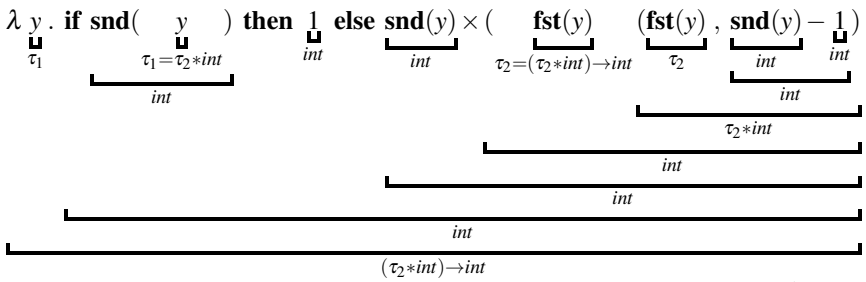
Example 7.3 (Non-typable term of HOFL). Let us consider the following function, which computes the factorial without using recursion.

begin
$$fact(f,x) \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \times f(f, x-1)$$
$$fact(fact, 3)$$
end

The first instruction defines *fact* as a function that takes two arguments (e.g., a function f and an integer x) and returns 1 if $x = 0$ and returns $x \times f(f, x-1)$ otherwise. The second instruction invokes *fact* by passing *fact* as a first argument and the number 3 as second argument. Since $3 \neq 0$, the invocation will trigger the calculation $3 \times fact(fact, 2)$ and so on. It can be translated to HOFL as follows:

$$fact \stackrel{\text{def}}{=} \lambda y.\ \textbf{if } \textbf{snd}(y) = 0 \textbf{ then } 1 \textbf{ else } \textbf{snd}(y) \times \textbf{fst}(y)(\textbf{fst}(y), \textbf{snd}(y) - 1)$$

We can try to infer the type of *fact* as follows:



We derive $\mathbf{fst}(y) : \tau_2$ and $\mathbf{fst}(y) : (\tau_2 * int) \to int$. Thus we have $\tau_2 = (\tau_2 * int) \to int$ which has no solution.

We recall the unification algorithm from Section 2.1.4 that can be used to solve general systems of type equations as well. We recall it here, in compact form, to address explicitly the unification of terms that denote types. The idea is that types are terms built over a suitable signature. In the case of HOFL, the signature just consists of the constant *int* and two binary operators $*$ and $\to$ and variables are usually denoted as τ's. We start from a system of type equations like:

$$\begin{cases} t_1 = t'_1 \\ t_2 = t'_2 \\ \dots \\ t_k = t'_k \end{cases}$$

and then we apply iteratively in any order the following steps:

1. We eliminate all the equations like $\tau = \tau$ for τ a type variable.
2. For each equation of the form $f(u_1, \dots, u_n) = f'(u'_1, \dots, u'_m)$:[1]

   if $f \neq f'$: then the system has no solutions and we stop.
   if $f = f'$: then $n = m$ so we must have:

   $$u_1 = u'_1, u_2 = u'_2, \dots, u_n = u'_n$$

   and thus we replace the original equation with these.

3. For each equation of the type $\tau = t$ with $t \neq \tau$:

   if τ appears in t: then the system has no solutions.
   if τ does not appear in t: we replace each occurrence of τ with t in all the other equations.

Eventually, either the system is recognized as unsolvable, or all the variables in the original equations are assigned to solution terms. Note that the order of the step executions can affect the complexity of the algorithm but not the solution. The best

[1] In our case f and f' can be taken from $\{int, *, \to\}$.

execution strategies yield a complexity linear or quasi linear with the size of the original system of equations.

Example 7.4. Let us now apply the algorithm to the Example 7.3: We have the type equation

$$\tau_2 = (\tau_2 * int) \to int$$

1. We cannot apply step 1 of the algorithm, because the equation does not express a trivial equality.
2. We cannot apply step 2 either, because the left-hand side of the equation consists of a variable and not of an operator applied to some subterms, as required.
3. Step 3 can be applied and it fails, because the type variable τ_2 appears in the right hand side.

Here we show another interesting term which is not typable.

Example 7.5 (Non-typable terms). Let us define a pre-term t which, when applied to the argument 0, should define the list of all even numbers:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2)))$$

Intuitively, the term $t\ 0$ takes the value 0 and place it in the first position of a pair, whose second component is the term t itself applied to $0+2=2$, so recursively $t\ 0$ should represent the infinite list of all even numbers:

$$t\ 0 \equiv (0, (t\ 2)) \equiv (0, (2, (t\ 4))) \equiv \cdots \equiv (0, (2, (4, \ldots)))$$

Let us show that this term is not typable:

$$\begin{array}{rl}
t = \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2))) : \tau & \nwarrow_{\widehat{p}=\tau}\ \lambda x.\ (x, (p\ (x+2))) : \tau \\
\nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1} & (x, (p\ (x+2))) : \tau_2 \\
\nwarrow_{\tau_2=\tau_3*\tau_4} & x : \tau_3,\ (p\ (x+2)) : \tau_4 \\
\nwarrow_{\widehat{x}=\tau_3} & (p\ (x+2)) : \tau_4 \\
\nwarrow & p : \tau_5 \to \tau_4,\ (x+2) : \tau_5 \\
\nwarrow_{\widehat{p}=\tau_5\to\tau_4} & (x+2) : \tau_5 \\
\nwarrow_{\tau_5=int} & x : int \\
\nwarrow_{\widehat{x}=int} & \square
\end{array}$$

So we have:

$$\widehat{x} = \tau_1 = \tau_3 = int \qquad \tau_2 = (\tau_3 * \tau_4) = (int * \tau_4) \qquad \tau = (\tau_1 \to \tau_2) = (int \to (int * \tau_4)) \qquad \tau_5 = int$$

From which:

$$\widehat{p} = \tau = (int \to (int * \tau_4)) \qquad \text{and} \qquad \widehat{p} = (\tau_5 \to \tau_4) = (int \to \tau_4)$$

Thus it must be the case that

$$int * \tau_4 = \tau_4$$

which is absurd, because it is not possible to unify τ_4 with a composed term containing an occurrence of τ_4. The above argument is represented more concisely below:

$$t = \textbf{rec}\ p.\quad \lambda\, \underset{int}{x}\,.\ (\underset{int}{x}, (\ \underset{int\to\tau_4}{p}\quad (\underbrace{\underset{int}{x}\ \underset{int}{2}}_{int})))$$
$$\underbrace{\qquad\qquad}_{\tau_4}$$
$$\underbrace{\qquad\qquad\qquad}_{int*\tau_4}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{(int\to(int*\tau_4)) = (int\to\tau_4) \Rightarrow \tau_4 = (int*\tau_4)}$$

So we have no solutions, and the term is not a well-formed term.

7.2 Operational Semantics of HOFL

In Section 6.1 we have defined the concepts of free variables and substitution for the λ-calculus. Now we define the same concepts in the case of HOFL, which will be necessary to define its operational semantics.

Definition 7.4 (Free variables). We define the set of free-variables of HOFL terms by structural recursion, as follows:

$$\begin{aligned}
\mathrm{fv}(n) &\stackrel{\text{def}}{=} \varnothing \\
\mathrm{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\
\mathrm{fv}(t_0\ \mathrm{op}\ t_1) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \cup \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}((t_0, t_1)) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{fst}(t)) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\textbf{snd}(t)) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\lambda x.\, t) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \backslash \{x\} \\
\mathrm{fv}((t_0\ t_1)) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{rec}\ x.\, t) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \backslash \{x\}
\end{aligned}$$

Finally as done for λ-calculus we define the substitution operator on HOFL.

Definition 7.5 (Capture-avoiding substitution). Capture avoiding substitution $[{}^{t}/_{x}]$ of x with t is defined by structural recursion over HOFL terms as follows:

$$
\begin{aligned}
n[t/x] &= n \\
y[t/x] &\stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(t_0 \text{ op } t_1)[t/x] &\stackrel{\text{def}}{=} t_0[t/x] \text{ op } t_1[t/x] \quad \text{with op} \in \{+,-,\times\} \\
(\textbf{if } t' \textbf{ then } t_0 \textbf{ else } t_1)[t/x] &\stackrel{\text{def}}{=} \textbf{if } t'[t/x] \textbf{ then } t_0[t/x] \textbf{ else } t_1[t/x] \\
(t_0,t_1)[t/x] &\stackrel{\text{def}}{=} (t_0[t/x], t_1[t/x]) \\
\textbf{fst}(t')[t/x] &\stackrel{\text{def}}{=} \textbf{fst}(t'[t/x]) \\
\textbf{snd}(t')[t/x] &\stackrel{\text{def}}{=} \textbf{snd}(t'[t/x]) \\
(t_0\ t_1)[t/x] &\stackrel{\text{def}}{=} (t_0[t/x]\ t_1[t/x]) \\
(\lambda y.\ t')[t/x] &\stackrel{\text{def}}{=} \lambda z.\ (\ t'[z/y][t/x]\ ) \quad \text{with } z \notin \text{fv}(\lambda y.\ t') \cup \text{fv}(t) \cup \{x\} \\
(\textbf{rec } y.\ t')[t/x] &\stackrel{\text{def}}{=} \textbf{rec } z.\ (t'[z/y][t/x]) \quad \text{with } z \notin \text{fv}(\textbf{rec } y.\ t') \cup \text{fv}(t) \cup \{x\}
\end{aligned}
$$

Note that in the last two rules we perform α-conversion $[z/y]$ of the bound variable y with a fresh identifier z before the substitution. This ensures that the free occurrences of y in t, if any, are not bound accidentally after the substitution. As discussed in Section 6.1, the substitution is well-defined if we consider the terms up to α-conversion (i.e., up to the renaming of bound variables). Obviously, we would like to extend these concepts to typed terms. So we are interested in understanding how substitution and α-conversion interact with typing. We have the following results:

Theorem 7.1 (Substitution Respects Types). *Let $x,t : \tau$ and $t' : \tau'$. Then, we have*

$$t'[t/x] : \tau'$$

Proof. We leave as an exercise to prove the property by induction on the derivation, after having proved that for the special case where $t = z : \tau$ we have $t'[z/y] : \tau'$ with a derivation that has the same length as the derivation of $t' : \tau'$. □

We are now ready to present the operational semantics of HOFL. Unlike IMP, the operational semantics of HOFL is a simple manipulation of terms. This means that the operational semantics of HOFL defines a method to calculate the *canonical form* of a given term of HOFL. In particular, we focus on *closed terms* only, i.e., terms t with no free variables ($\text{fv}(t) = \varnothing$). Canonical forms are particular closed terms, which we will assume to be the results of calculations (i.e., as ordinary values). For each type we fix the set of terms in canonical form by taking a subset of terms which reasonably represent the notion of values for that type.

As shown in the previous section, HOFL has three type constructors: the constant *int*, and the binary operators $*$ for pairs and $\to$ for functions. Terms which represent the integers provide the obvious canonical forms for the integer type. For pair types we take any pair of terms as canonical form: note that this choice is arbitrary; for example we could have taken instead pairs of terms that are themselves in canonical

form. We will explain later the rationale of our choice. Finally, since HOFL is a higher-order language, functions are values. So is quite natural to take all abstractions as canonical forms for the arrow type.

Definition 7.6 (Canonical forms). Let us define a set C_τ of canonical forms for each type τ as follows:

$$\frac{}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.\, t : \tau_0 \to \tau_1 \quad \lambda x.\, t \text{ closed}}{\lambda x.\, t \in C_{\tau_0 \to \tau_1}}$$

We now define the rules of the operational semantics; these rules define an evaluation relation:

$$t \to c$$

where t is a well-formed closed term of HOFL and c is its canonical form.

For terms that are already in canonical form according to Definition 7.6 we let:

$$\frac{}{c \to c}$$

For clarity, the above rule offers a concise representation to the otherwise verbose rules:

$$\frac{}{n \to n} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \to (t_0, t_1)} \qquad \frac{\lambda x.\, t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.\, t \to \lambda x.\, t}$$

Next, we give the rules for arithmetic expressions.

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{\text{op}}\, n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

For the arithmetic operators the semantics is obviously the simple application of the correspondent meta-operator as well as in IMP. Only, here we distinguish between HOFL syntactic operators and meta-operators by underlying the latter. For instance, we have $1 + 2 \to 3$, since $1 \to 1$, $2 \to 2$ and $1 \underline{+} 2 = 3$.

We recall that for the conditional statement, since we have no boolean values, we use the convention that **if** t **then** t_0 **else** t_1 stands for **if** $t = 0$ **then** t_0 **else** t_1, so the premise $t \to n \neq 0$ means the test is false and $t \to 0$ means the test is true.

Let us now consider the pairing. Obviously, since we consider pairs as canonical values, we do not have to add further rules for simple pairs. We have instead two rules for projections:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1}$$

The rules are obviously similar: the canonical form of t is computed, which must be of the form (t_0, t_1), because t must have pair type for the projection to be applicable and $\mathbf{fst}(t)$ typable. Note however that t_0 and t_1 need not be in canonical form. So only the canonical form of the component indicated by the projection operator is computed, with the other component discarded.

Function abstraction is handled by the axiom for terms already in canonical form, as in the case of pairing. For function application, we show two rules, according to two different evaluation strategies, called *lazy* and *eager*. In the lazy operational semantics, we do not evaluate the canonical forms of the parameters when passing them to the function body. The lazy semantics will be our primary focus in the rest of this part of the book concerned with HOFL.

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(lazy)}$$

We remark that the in the second premise of the rule, we replace with t_0 each occurrence of x in t_1', i.e., we replace each instance of x with a copy of the (non evaluated) parameter t_0 and not with its canonical form.

For the sake of discussion let us consider the *eager* alternative to this rule.

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(eager)}$$

Unlike the lazy semantics, the eager semantics evaluates the parameters only once and before the substitution. Note that these two types of evaluation are not equivalent. If the evaluation of the argument does not terminate, and it is not needed, the lazy rule will guarantee convergence, while the eager rule will diverge. Vice versa, according to the lazy semantics, if the argument is actually needed it may be later evaluated several times (every times it is used).

Finally, we have a last rule for recursive terms:

$$\frac{t[{}^{\mathbf{rec}\ x.\ t}/_x] \to c}{\mathbf{rec}\ x.\ t \to c}$$

To evaluate the canonical form of $\mathbf{rec}\ x.\ t$ we first plug in t the recursive definition itself in place of every occurrence of x and then compute the canonical form.

Example 7.6. Let us consider the term $t \stackrel{\text{def}}{=} \lambda x.\, 0 + x$. Clearly the term t is closed and typable, with $t : \mathit{int} \to \mathit{int}$. It is already in canonical form and we have in fact:

$$t \to c \ \nwarrow_{c=\lambda x.\ 0+x} \Box$$

Example 7.7. Let us consider the term $t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ 0 + x$. Clearly the term t is closed and typable, with $t : \mathit{int}$. We show that the term has no canonical form, in fact:

$$\begin{array}{rl} t \to c & \nwarrow\ (0+x)[{}^{t}/_{x}] \to c \\ & =\ 0+t \to c \\ & \nwarrow_{c=c_1 \pm c_2}\ 0 \to c_1,\ t \to c_2 \\ & \nwarrow_{c_1=0}\ t \to c_2 \\ & \nwarrow\ \ldots \end{array}$$

Let us see an example which illustrates how rules are used to evaluate a function application.

Example 7.8 (Factorial). Let us consider the well-formed factorial function seen in the Example 7.1:

$$\mathit{fact} \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (f(x-1))$$

It is immediate to see that *fact* is closed and we know it has type $\mathit{int} \to \mathit{int}$. So we can calculate its canonical form by using the last rule seen and the axiom for terms in canonical form:

$$\frac{\lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\mathit{fact}(x-1))}{\mathit{fact} \to \lambda x.\ \textbf{if}\, x\, \textbf{then}\, 1\, \textbf{else}\, x \times (\mathit{fact}(x-1))}$$

We can apply this function to a specific value and calculate the canonical form of the result. For example, we see what is the canonical form c of the (closed and typable) term $(\mathit{fact}\ 2) : \mathit{int}$

$$
\begin{array}{rl}
(\mathit{fact}\ 2) \to c & \nwarrow\ \mathit{fact} \to \lambda x'.\, t',\quad t'[^2/_{x'}] \to c \\
& \nwarrow\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x'.\, t', \\
& \quad t'[^2/_{x'}] \to c \\
\nwarrow_{x'=x,\ t'=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ 2\ \textbf{then}\ 1\ \textbf{else}\ 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow^* & 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow_{c=c_1 \underline{\times} c_2} & 2 \to c_1,\quad (\mathit{fact}(2-1)) \to c_2 \\
\nwarrow^*_{c_1=2} & \mathit{fact} \to \lambda x''.\, t'',\quad \underbrace{t''[^{2-1}/_{x''}] \to c_2}_{\text{note that } 2-1 \text{ is not evaluated}} \\
\nwarrow_{x''=x,\ t''=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ (2-1)\ \textbf{then}\ 1 \\
& \textbf{else}\ (2-1) \times (\mathit{fact}((2-1)-1)) \to c_2 \\
\nwarrow & 2-1 \to n,\quad n \neq 0, \\
& (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow_{n=n_1 \underline{-} n_2} & 2 \to n_1,\quad 1 \to n_2,\quad n_1 \underline{-} n_2 \neq 0, \\
& \quad (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^*_{n_1=2,\ n_2=1} & (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^*_{c_2=c_3 \underline{\times} c_4} & 2-1 \to c_3,\quad \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^*_{c_3=1} & \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^* & \textbf{if}\ (2-1)-1\ \textbf{then}\ 1 \\
& \textbf{else}((2-1)-1) \times (\mathit{fact}(((2-1)-1)-1)) \to c_4 \\
\nwarrow & (2-1)-1 \to 0,\quad 1 \to c_4 \\
\nwarrow^*_{c_4=1} & \square
\end{array}
$$

So we have

$$c = c_1 \underline{\times} c_2 = 2 \underline{\times} (c_3 \underline{\times} c_4) = 2 \underline{\times} (1 \underline{\times} 1) = 2$$

Example 7.9 (Lazy vs eager evaluation). The aim of this example is to illustrate the difference between lazy and eager semantics. Let us consider the term

$$t \stackrel{\text{def}}{=} ((\lambda x : \mathit{int}.\ 3)(\textbf{rec}\ y : \mathit{int}.\ y)) : \mathit{int}$$

also written more concisely as

$$t \stackrel{\text{def}}{=} (\lambda x.\ 3)\, \textbf{rec}\ y.\ y$$

assuming $\widehat{x} = \widehat{y} = \mathit{int}$. It consists of the constant function $\lambda x.\ 3$ applied to a diverging term **rec** $y.\ y$ (i.e., a term with no canonical form).

- **Lazy evaluation**
  Lazy evaluation evaluates a parameter only if needed: if a parameter is never used in a function or in a specific instance of a function it will never be evaluated. Let us show our example:

$$\begin{array}{lll} ((\lambda x.\ 3)\,\mathbf{rec}\ y.\ y) \to c & \nwarrow & \lambda x.\ 3 \to \lambda x.\ t, \quad t[^{\mathbf{rec}\ y.\ y}/_x] \to c \\ & \nwarrow_{t=3} & 3[^{\mathbf{rec}\ y.\ y}/_x] \to c \\ & \nwarrow_{c=3} & \square \end{array}$$

So although the argument **rec** $y.\ y$ as no canonical form the application can be evaluated.

- **Eager evaluation**
  On the contrary in the eager semantics this term has no canonical form since the parameter must be evaluated before the application, leading to a diverging computation:

$$\begin{array}{lll} ((\lambda x.\ 3)\,\mathbf{rec}\ y.\ y) \to c & \nwarrow & \lambda x.\ 3 \to \lambda x.\ t, \quad \mathbf{rec}\ y.\ y \to c_1, \quad t[^{c_1}/_x] \to c \\ & \nwarrow_{t=3} & \mathbf{rec}\ y.\ y \to c_1, \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \mathbf{rec}\ y.\ y \to c_1 \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \ldots \end{array}$$

  So the evaluation does not terminate.

However if the parameter of a function is used n times, the parameter would be evaluated n times (at most) in the lazy semantics and only once in the eager case.

We conclude this chapter by presenting a theorem that guarantees that

1. if a term can be reduced to a canonical form then it is unique (determinacy);
2. the evaluation of the canonical form preserves the type assignments (type preservation).

Theorem 7.2. *Let t be a closed and typable term.*

1. *For any canonical form c, c', if $t \to c$ and $t \to c'$ then $c = c'$*
2. *For any canonical form c and type τ, if $t \to c$ and $t : \tau$ then $c : \tau$*

Proof. Property 1 is proved by rule induction, taking the predicate

$$P(t \to c) \stackrel{\text{def}}{=} \forall c'.\ t \to c' \Rightarrow c = c'$$

We show only the case of the application rule, the remainder of the proof of the theorem, including the proof of Property 2, is left as an exercise (see Problem 7.11). We have the rule:

$$\frac{t_1 \to \lambda x.\ t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We assume the inductive hypotheses:

- $P(t_1 \to \lambda x.\ t_1') \stackrel{\text{def}}{=} \forall c'.\ t_1 \to c' \Rightarrow \lambda x.\ t_1' = c'$
- $P(t_1'[^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall c'.\ t_1'[^{t_0}/_x] \to c' \Rightarrow c = c'$

We want to prove:

$$P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall c'.\ (t_1\ t_0) \to c' \Rightarrow c = c'$$

As usual, we assume the premise of the implication:

$$(t_1\ t_0) \to c'$$

From it, by goal reduction:

$$(t_1\ t_0) \to c' \quad \nwarrow \quad t_1 \to \lambda x'.\ t_1'', \quad t_1''[{}^{t_0}/_{x'}] \to c'$$

Then we have by the first inductive hypothesis:

$$\lambda x.\ t_1' = \lambda x'.\ t_1''$$

i.e., $x = x'$ and $t_1' = t_1''$. Then $t_1''[{}^{t_0}/_{x'}] = t_1'[{}^{t_0}/_{x}]$ and by the second inductive hypothesis we have $c = c'$. □

Problems

7.1. Let $x, y, w : \mathit{int}$, and $f : \mathit{int} \to (\mathit{int} \to \mathit{int})$. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ (\lambda y.\ (y + w))\ \textbf{else}\ (f\ w)$$

1. Compute the term $t[{}^{(f\ x\ y)}/_{w}]$.
2. Compute the term $t[{}^{(f\ x\ y)}/_{x}]$.

Hint: You are allowed to introduce additional (typed) variables if needed.

7.2. Is it possible to assign a type to the HOFL pre-term below? If yes, compute its principal type.

$$\textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ f(\ \textbf{fst}(x)\ ,\ (\textbf{fst}(x)\ \textbf{snd}(x)))$$

7.3. A *list of positive numbers* is defined by the following syntax, where $n \in \mathbb{N}, n > 0$:

$$L\ ::=\ (n, 0)\ \mid\ (n, L)$$

For instance the list with 3 followed by 5 is represented by the term $(3, (5, 0))$.

1. Define a HOFL term t (closed and typable) such that the application $(t\ L)$ to a list L of 3 elements returns the last element of the list.
2. Is it possible to find a closed and typable HOFL term which returns the last element of a generic list?

7.4. Given the two HOFL terms

$$t_1 \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x+3$$
$$t_2 \stackrel{\text{def}}{=} \lambda z.\ \mathbf{fst}(z)+3$$

1. Compute their types.
2. Prove that, given the canonical form $c : \tau$, the two terms

$$((t_1\ 1)\ c) \qquad \text{and} \qquad (t_2\ (1,c))$$

yield the same canonical form.

7.5. Let us consider the HOFL term

$$\mathit{map} \stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ ((f\ \mathbf{fst}(x)),(f\ \mathbf{snd}(x)))$$

Show that *map* is a typable term and give its principal type. Then, compute the canonical form of the term

$$\mathit{map}\ (\lambda x.\ 2\times x)\ (1,2)$$

7.6. Determine the type of the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ ((\lambda y.\ \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0)\ x).$$

Then compute its operational semantics.

7.7. Recall the definition of *binomial coefficients* $\binom{n}{k}$ from Problem 4.13:

$$\binom{n}{0} \stackrel{\text{def}}{=} 1 \qquad \binom{n}{n} \stackrel{\text{def}}{=} 1 \qquad \binom{n+1}{k+1} \stackrel{\text{def}}{=} \binom{n}{k} + \binom{n}{k+1}.$$

where $n,k \in \mathbb{N}$ and $0 \le k \le n$. Consider the corresponding HOFL program:

$$\begin{aligned} t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda n.\ \lambda k.\ \ & \mathbf{if}\ k\ \mathbf{then}\ 1 \\ & \mathbf{else\ if}\ n-k\ \mathbf{then}\ 1 \\ & \qquad \mathbf{else}\ ((f\ (n-1))\ k)+((f\ (n-1))\ (k-1)). \end{aligned}$$

Compute its type and evaluate the canonical form of the term $((t\ 2)\ 1)$.

7.8. Consider the Fibonacci sequence already found in Problem 4.14

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1)+F(n)s.$$

where $n \in \mathbb{N}$.

1. Write a well-formed, closed HOFL term $t : \mathit{int} \to \mathit{int}$ to compute F.

2. Compute the operational semantics of $(t\ 2)$

7.9. Check if the HOFL pre-term

$$\lambda x.\ \lambda y.\ \lambda z.\ \textbf{if}\ z\ \textbf{then}\ (y\ x)\ \textbf{else}\ (x\ y).$$

is typable, in which case give its type.

7.10. Let us consider the HOFL pre-term $t = \lambda x.\ (x\ x)$. Prove that it is not typable. Try to compute anyway the canonical form of the application $(t\ t)$. Given that any well-typed term without recursion has a canonical form, argue why the given term is not typable.

7.11. Complete the proof of Theorem 7.2.

7.12. Suppose we extend HOFL with the inference rule:

$$\frac{t_1 \to 0}{t_1 \times t_2 \to 0}$$

Prove that the determinism property

$$\forall t, c_1, c_2.\ t \to c_1\ \wedge\ t \to c_2 \Rightarrow c_1 = c_2$$

is still valid. What if also the inference rule below is added?

$$\frac{t_2 \to 0}{t_1 \times t_2 \to 0}$$