Fundamenta Informaticae XXI (2001) 1001–1035
IOS Press

Big-step Operational Semantics Revisited

Jarosław Dominik Mateusz Kuśmierek
Institute of Informatics
Warsaw University, Poland
`jdk@google.com`

Viviana Bono
Department of Computer Science
Torino University, Italy
`bono@di.unito.it`

Abstract. In this paper we present a novel approach to big-step operational semantics. This approach stems from the observation that the typical type soundness property formulated via a big-step operational semantics is weak, while the option of using a small-step operational semantics is not always an option, because it is less intuitive to build and understand. We support our claim by using a simple language called LM, for which we present a big-step semantics expressed with the new approach, allowing one to formulate a stronger type soundness property. We prove this property for LM and we present an example of an error in the typing rules which does not violate the typical type soundness property, but does violate ours.

1. Introduction

The *big-step* operational semantics formulation (as introduced by Kahn [8] and also known as *natural* semantics) is often thought as a more intuitive choice than *small-step* semantics [10, 7, 2, 6, 1]. The superiority of such a formulation is especially visible for imperative languages with recursion and nested scopes. Moreover, a small-step semantics formulation of a language (as introduced by Plotkin [12]) in many cases requires one to define: (*i*) another language used internally in the reduction process; (*ii*) additional rules to enforce a specific order of the evaluation. Examples of small-step semantics can be found in [4, 5, 3].

Big-step semantics is used to define naturally the semantics of programs in terms of judgements that describe that a program, in a given context, yields a final value/state/..., (see, for example, [3]). This is often convenient, typically leading to a clear semantics, easy to understand and follow. However, an unfortunate consequence is that it is only possible to capture directly properties that can be formulated as “if a program evaluates to some value/state/... then ...”. For many applications this is sufficient, but, in general, this does not distinguish among computations that do not lead to successful termination of a program. In particular, it cannot distinguish looping indefinitely from getting stuck at some internal point. Consequently, properties that concern the intermediate configurations that may emerge in the course of a program evaluation are disregarded. Such properties can typically be expressed in terms of a small-step operational semantics. However, even if a small-step operational semantics is given at some appropriate level of abstraction, typically its readability suffers due to the overwhelming details. In particular, it is difficult to state that the execution of a program gets stuck, or to prove that the execution of a program does not get stuck. As a result, it is more complicated to express the subject reduction and the type soundness properties.

We introduce a novel approach to modelling type soundness with big-step operational semantics. In this approach, the big-step rules are formulated in a way it is always possible either to choose a rule deterministically, or to delay the choice of the rule by evaluating common premises of different rules, until a deterministic choice is possible. With this formulation, it is possible to define a notion of *intermediate steps* of a program execution, that permits to distinguish between looping indefinitely from getting stuck at some internal point, without the cumbersome details of a small-step semantics and without the need of introducing any explicit error value.

In order to illustrate our approach, we introduce a statically typed toy language called LM[1]. By means of LM, we show why existing formulations are weak with respect to type soundness, by introducing a small, but significant, error in the type system which is not captured within a typical big-step-based type soundness property, but it is instead captured by ours.

Then, we define the notion of intermediate steps for LM, formulate the subject reduction and the type soundness properties following the new approach, and give their proofs.

The language LM is sufficient to present our approach. In particular, LM does not feature higher-order types, because they would not change anything in the approach, since our notion of intermediate steps of a program execution does not depend on typing. However, the constructs present in the language do influence our technique; for example, LM has an explicit if-construct, which introduces naturally two separate big-step semantics rules for one form of instruction. This, in turn, may introduces non-determinism (see Section 6.4).

Indeed, the first author applied this approach to a more complex language, Magda, in his PhD thesis [9]. The approach does not change, even though Magda is object-oriented and implicitly second-order (as functions can be encapsulated in objects as virtual methods and therefore passed as parameters).

Our solution is in many respects similar to the one presented by Ager in [2] (even though it was developed independently). We present the similarities and differences in Section 6.4.

The paper is structured as follows. In Section 2 we introduce our toy language LM. In Section 3 we introduce the big-step semantics of our language and in Section 4 the type system. In Section 5, by using our language, we present the drawbacks of the classical approach to type safety with big-step operational semantics. In Section 6 we propose our new approach to computation with big-step semantics. In Section 7, we show a formulation and the proof of the subject reduction property for LM. This proof uses the model of computation presented in Section 6, in order to obtain a stronger subject reduction property than the usual one when big-step semantics is exploited. In Section 8, we formulate and prove the type soundness property for LM. In Section 9 we conclude our work.

[1]The acronym LM stands for “Little Magda”, named after the Magda language introduced in [9].

2. The LM language

In this section we present our toy language LM. This language draws inspiration from the language Magda, presented in the first author's PhD thesis [9]. Magda is a fully-fledged mixin-based object-oriented language, but, in order to present our approach to big-step operational semantics, it is enough to consider LM, for the motivations presented in the introduction.

2.1. The structure of a program

Every program in LM consists of two parts. The first part is a list of *function declarations*. The second part is an expression, called *main expression*. The execution of the program corresponds to the evaluation of the main expression.

```
Int Add (x:Int, y:Int, z:Int) = begin
  return x + y + z
end
//
Add(1, 1 + 1, 0)
```

Figure 1. A simple example in LM

A simple example of a program written in LM can be seen in Figure 1. The program consists of a function named `Add` and a main expression invoking this function. Unsurprisingly, the result of the evaluation of this program is 3.

2.2. Local variables

A function in LM can contain declarations of local variables. As one can see in the `MultiplyBy8` function on Figure 2, variable declarations are placed in the header of the function, after its name and the declaration of parameters, before the keyword `begin`. Each declaration consists of a variable name followed by a colon and the type of the variable. The type of the variable is either `Bool` or `Int`. Every local variable of type `Int` initially contains the value 0, while a `Bool` variable has the initial value `false`.

```
Int MultiplyBy8(par:Int)
  x:Int;
begin
  x := par + par;
  x := x + x;
  return x + x
end;
```

Figure 2. Local variables in LM

```
Int IterativeMultiplication(x:Int, y:Int)
  result:Int;
begin
  if (y < 0) then
    x := 0 - x;
    y := 0 - y
  end;
  while (y > 0)
    result := result + x;
    y := y - 1;
  end;
  return result
end
```

Figure 3. `Bool` type and control instructions

2.3. Control instructions and `Bool` type

The values of type `Bool` are obtained either by using the constants `true` and `false`, or by comparing two `Int`'s.

The `Bool` values are used in two control instructions: conditional `if` and loop `while`, which both begin with a condition expression of type `Bool`. The syntax of those instruction can be seen in the example in Figure 3.

2.4. Formal definition of the syntax

We present in Figure 4 the formal syntax of LM in a Backus-Naur form. To increase the readability, all keywords (terminal symbols) in the grammar are written in non-capital letters and in quotes, like `'implements'`. A general non-terminal symbol, denoting all acceptable identifiers, is denoted as `<ID>`. An additional non-terminal is `<NUMBER_LITERAL>`, which denotes any constant value representing an integer (`Int`) number. All nonterminals are written in capital letters with underscores.

2.5. Notation

We introduce now the notational conventions which will be used through the rest of the paper.

For each pair p we use $p|_1$ to denote the first element of p, and $p|_2$ to denote the second element of p. In general, for an arbitrary tuple t, we use $t|_n$ to denote the n-th element of t.

For each function or partial function f we use $f\{a \mapsto b\}$ to denote a function which, when applied to x, has value b if $x = a$ and $f(x)$, otherwise. We also use $f\{a_1 \mapsto b_1; ...; a_n \mapsto b_n\}$ to denote $f\{a_1 \mapsto b_1\}...\{a_n \mapsto b_n\}$, where all a_1 ... a_n are different elements.

We use also the convention that every function f is a set of pairs (a, b), such that $f(a) = b$.

A set of elements $\{p_1, ..., p_n\}$ is often denoted as $\overline{p}$. Additionally, a sequence of elements $(p_1, ..., p_n)$ is occasionally abbreviated as $\overrightarrow{p}$. We use notation $a \cdot \overrightarrow{b}$ to denote a sequence $\overrightarrow{b}$ with element a added at the beginning. Similarly, we use notation $\overrightarrow{b} \cdot a$ to denote a sequence $\overrightarrow{b}$ with element a added at the end. For simplicity, we often treat a sequence as the set of its values.

```
// --------------------- The program -------------------------------------
PROGRAM ::= ( FUN_DECLARATION )* EXPRESSION

// ---------------- Function declarations ---------------------------------
FUN_DECLARATION       ::= TYPE  <ID> '(' PARAMETER_DECLS ')'
                            VARIABLE_DECLARATIONS 'begin' INSTRUCTION 'end';
TYPE                  ::= 'Bool' | 'Int'
PARAMETER_DECL        ::= <ID> : TYPE
PARAMETER_DECLS       ::= [ PARAMETER_DECL (';' PARAMETER_DECL )* ]
VARIABLE_DECLARATIONS ::= ( <ID> ':' TYPE ';' )*

// -------------------- Instructions  ------------------------------------
WHILE_LOOP   ::= 'while' '(' EXPRESSION ')' INSTRUCTION 'end'
IF_COND      ::= 'if' '(' EXPRESSION ')' 'then' INSTRUCTION
                 [ 'else' INSTRUCTION ] 'end'
INSTRUCTION  ::= <ID> ':=' EXPRESSION   |
                 EXPRESSION             |
                 'return' EXPRESSION    |
                 WHILE_LOOP             |
                 IF_COND                |
                 INSTRUCTION ';' INSTRUCTION

// --------------------- Expressions -------------------------------------
ACTUAL_PARAMETERS  ::= '('  [ EXPRESSION (',' EXPRESSION )* ] ')'
FUNCTION_CALL      ::= <ID> ACTUAL_PARAMETERS

BINARY_OPERATOR    ::= '+' | '-' | '>'
EXPRESSION         ::= <NUMBER_LITERAL> | 'true' | 'false' |
                       <ID> |
                       '(' EXPRESSION ')' |
                       FUNCTION_CALL |
                       EXPRESSION BINARY_OPERATOR EXPRESSION
```

Figure 4. The formal grammar of LM

From now on, when we indicate a term t, we will always refer to one specific occurrence of t in the program.

2.6. Syntax-related definitions

In this section we introduce some syntax-related notions which will make the definitions of semantics and type checking rules easier to specify.

For $t \in \{\texttt{Bool}, \texttt{Int}\}$, we use $\mathit{InitVal}(t)$ to denote the initial value of a variable declared with the given type. This function has the following values: $\mathit{InitVal}(\texttt{Int}) = 0$ and $\mathit{InitVal}(\texttt{Bool}) = \mathit{ff}$.

We use the following notations for the sets of names occurring in a program.

- *FunNames*: the set of function identifiers declared in a program. We use the symbol *funID* or *f*, with optional indices, to denote an element of this set.
- *LocalIdentifiers*: the set of local variables and parameter names of the functions.

We define a set Decls, whose elements, denoted as D with optional indices, are partial functions from the set *FunNames* to the set of function declarations, thus the terms generated by the non-terminal `FUN_DECLARATION` defined in Figure 4. Given a program including function declarations $F_1, ..., F_n$, we calculate the related partial function D as $\mathit{DeclsVal}(F_1, ..., F_n)$, using the rules below. Notice that the side condition in the second rule makes $\mathit{DeclsVal}(\overrightarrow{F})$ undefined when $\overrightarrow{F}$ contains two declarations with the same name.

$$\mathit{DeclsVal}(\emptyset) = \emptyset$$

$$\frac{D = \mathit{DeclsVal}(\overrightarrow{F}) \qquad F = \mathit{type}\ \mathit{funID}\ \texttt{(...)}\ \texttt{...}\ \texttt{end} \qquad \mathit{funID} \notin \mathit{Dom}(D)}{\mathit{DeclsVal}(\overrightarrow{F}\ \ ;\ F) = D\{\mathit{funID} \mapsto F\}}$$

For each function name *funID* occurring in the program, we use:

- $\mathit{FunDecl}_D^{\mathit{funID}}$, to denote the declaration of a function named funID. This is equal to $D(\mathit{funID})$.
- $\mathit{FunParams}_D^{\mathit{funID}}$, to denote the sequence of declarations of function parameters in $\mathit{FunDecl}_D^{\mathit{funID}}$. Each element of this sequence is a parameter name followed by a colon and the type of the parameter.
- $\mathit{RetType}_D^{\mathit{funID}}$, to denote the return type occurring in $\mathit{FunDecl}_D^{\mathit{funID}}$.
- $\mathit{FunLocals}_D^{\mathit{funID}}$, to denote the sequence of local variable declarations in $\mathit{FunDecl}_D^{\mathit{funID}}$. Each such a declaration consists of a variable name and its type.
- $\mathit{FunInstr}_D^{\mathit{funID}}$, to denote the sequence of instructions within $\mathit{FunDecl}_D^{\mathit{funID}}$ with the following instruction added at the end: `return` v_I, where $v_I = \mathit{InitVal}(\mathit{RetType}_D^{\mathit{funID}})$. This is to ensure that every function returns the initial value of its return type if it happens not to execute any `return` instruction.

2.7. Preliminary definitions for LM semantics and typing

We define the following sets:

$$\begin{array}{rcl} \mathit{Values} & = & \mathbb{Z} \cup \{\, \mathit{tt}\,,\, \mathit{ff}\,\} \\ \mathit{Environments} & = & (\mathit{LocalIdentifiers} \rightharpoonup \mathit{Values}) \;\cup\; (\{\top\} \times \mathit{Values}) \end{array}$$

The set *Values* is the set of values of the expressions.

An element of the set *Environments* represents the (dynamic) state of the local identifiers visible in the scope of the function under execution, therefore it takes one of the forms:

- Either an association of the local identifiers with values. It has, as its domain, the names of the identifiers defined in the scope, therefore it is a partial function.
- Or a pair of the form $(\top, val)$, where val is an element of the set *Values*. This form means that the execution of the function is terminated with a `return` instruction. The value val is the result of a function evaluated by the `return` instruction.

We will use the symbol env with optional indices to denote elements of the set *Environments*.

To model the execution, we need also to represent its static context, as an element of the set *Contexts*:

$$\mathit{Contexts} \quad = \quad \mathit{FunNames} \cup \; \{\top\}$$

An element of this set is either the name of a function, the one currently under execution, or the special symbol ($\top$), denoting that the main expression of the program is currently under execution. We will use the symbol ctx with optional indices to denote elements of the set *Contexts*.

We define some additional functions which will be needed later on in the rules of the semantics and the type checking of LM.

For each $val \in$ *Values*, we use $\mathit{ValType}(val)$ to denote the type of a value, where $\mathit{ValType}(val)$ is `Bool` if $val \in \{\, \mathit{ff}\,,\, \mathit{tt}\,\}$ and `Int`, otherwise.

For each context ctx, we use $\mathit{IdTypes}_D(ctx)$ to denote a partial function defined for all identifiers declared in ctx. To every local identifier declared in ctx, $\mathit{IdTypes}_D(ctx)$ assigns the declared type of this identifier to its name. Therefore, $\mathit{IdTypes}_D(\top) = \emptyset$. In the other case, in which the context ctx refers to a function defined in D, the partial function $\mathit{IdTypes}_D(ctx)$ is defined as follows:

- When applied to a parameter names of the function referred by ctx, it returns its type as in $\mathit{FunLocals}_D^{ctx}$.
- When applied to a local variable declared within the body of a function referred by ctx, the function returns its type present in the declaration as in $\mathit{FunParams}_D^{ctx}$.

The type checking rules ensure that the set of names of local variables and function parameters are disjoint, in such way that the above definition is correct.

For any two integers v_1 and v_2, we define a predicate $\mathit{GreaterThan}(v_1, v_2)$ with the obvious meaning. We also define the functions Plus and Minus which are, respectively, the addition and the substraction between two integers.

Summarizing:

$$\begin{array}{rcl}
\mathit{ValType} & : & \mathit{Values} \to \mathit{Types} \\
\mathit{IdTypes}_D & : & \mathit{Contexts} \to (\mathit{LocalIdentifiers} \rightharpoonup \mathit{Types}) \\
\mathit{GreaterThan} & : & (\mathbb{Z} \times \mathbb{Z}) \to \{\, \mathit{ff}\,,\, \mathit{tt}\,\} \\
\mathit{Plus} & : & (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z} \\
\mathit{Minus} & : & (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}
\end{array}$$

3. Operational semantics

In this section we present the operational semantics of LM in a big-step formulation (as introduced by Kahn [8]).

Our semantics does not model any error, therefore it describes only successful executions. That is, for instance, there is no derivation for a program which attempts to add two values where one of them is not an `Int`, but a `Bool`. However, this program will not type check (see the type soundness property in Section 8).

3.1. Rules

In the semantics, we use the following forms of “evaluates to” judgments:

- Judgments which describe the evaluation of a sequence of instructions. Given: (*i*) a local environment env, denoting the values of the local variables, represented by an element of the *Environments* set; (*ii*) a static context ctx, represented by an element of the *Contexts* set; (*iii*) a set of function declarations present in the program D, represented by an element of the $Decls$ set; such judgments will model the changes in the values of local variables.

$$env, ctx, D \models \mathit{instr} \Rightarrow^{I} env'$$

- Judgments which describe how an expression evaluates to a value. The evaluation of an expression does not influence local variables, therefore the evaluation of an expression does not yield a new environment.

$$env, ctx, D \models \mathit{exp} \Rightarrow^{ex} val$$

- Judgments which describe how a program consisting of a sequence of function declarations $F_1 ... F_n$ and expression exp evaluates to a value. A judgment of this kind is called a *final judgment*.

$$\models F_1 ... F_n \;\; ; \;\; exp \Rightarrow^{P} val$$

When we refer to any of the above kinds of judgments we will use the symbol $\Rightarrow$.

Later on, we will use the notion of *partial judgment* to denote a judgment in which the value on the right-hand side of $\Rightarrow$ is unknown. A partial judgment has the form ... $\models$ ... $\Rightarrow ?$.

We use the name *premise* for each judgment of the form ... $\models$ ... $\Rightarrow$ ... occurring as an assumption in a rule. All the other assumptions are called *side-conditions*. Moreover, whenever in a rule we use a sequence of assumptions of dynamic length of the form, for instance, $env^1, ctx^1, D \models I_1 \Rightarrow^I env_1$ ... $env^n, ctx^n, D \models I_n \Rightarrow^I env_n$, we will often refer to it as one single assumption.

3.2. Program evaluation rule

The value of a program is defined as the value of its main expression.

$$\frac{\emptyset, \top, \mathit{DeclsVal}(F_1...F_n) \models exp \Rightarrow^{ex} val}{\models F_1...F_n \;\; ; \;\; exp \Rightarrow^{P} val}$$

3.3. Instruction execution rules

The rules for the execution of the instructions are rather straightforward. The only non-trivial case is the `return` instruction, which ends the execution of a function body. This termination of execution is performed by putting the special value $\top$ in the environment. As a result, all the instructions containing component instructions (like the compound instruction, `if` and `while`), after the execution of any of its components must verify if it has executed `return`. Then, depending on this, the compound instruction terminates its execution, or continues with the execution of the other component.

Assignment to a local variable. The execution of a variable assignment instruction $\mathit{VarName}$ `:=` exp; changes the value of the variable in the environment.

$$\frac{env, ctx, D \models exp \Rightarrow^{ex} val}{env, ctx, D \models \; \mathit{VarName} \; \texttt{:=} \; exp \Rightarrow^{I} env\{\mathit{VarName} \mapsto val\}}$$

Conditional instruction. The execution of a conditional instruction having the form `if` exp_1 `then` I_1 `else` I_2 `end`; begins with the evaluation of the condition expression exp_1. Then, depending on the value of exp_1, the first or the second rule is used to execute I_1 or I_2, respectively.

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} \mathit{tt} \qquad env, ctx, D \models I_1 \Rightarrow^{I} env''}{env, ctx, D \models \texttt{if} \; exp_1 \; \texttt{then} \; I_1 \; \texttt{else} \; I_2 \; \texttt{end} \Rightarrow^{I} env''}$$

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} \mathit{ff} \qquad env, ctx, D \models I_2 \Rightarrow^{I} env''}{env, ctx, D \models \texttt{if} \; exp_1 \; \texttt{then} \; I_1 \; \texttt{else} \; I_2 \; \texttt{end} \Rightarrow^{I} env''}$$

Return instruction. The execution of the `return` exp; instruction first evaluates the expression exp, and then returns an environment which contains the special element $\top$ and the value val'. This form of environment enforces the skipping of all further instructions within the same function (see below the rules for the compound and the `while` instructions). Notice that the old environment is completely discarded. This is safe because the execution of a local function is finished, thus the local variables will not be referenced anymore.

$$\frac{env, ctx, D \models exp \Rightarrow^{ex} val'}{env, ctx, D \models \texttt{return} \; exp \Rightarrow^{I} (\top, val')}$$

Compound instruction. The execution of a compound instruction of the form $I_1\,;I_2$ begins with the execution of the instruction I_1, yielding an environment env'. The continuation of the execution depends on the form of env'. If env' is not of the form $(\top, ...)$ then instruction I_2 is executed (see the first rule). If env' is of the form $(\top, ...)$, that represents the fact that a `return` instruction has been executed during the execution of I_1, then (according to the second rule) the instruction I_2 is skipped and the same environment env' is returned. Notice that, in this way, the special environment propagates upwards in the program tree up to the point of the function call, preventing all the remaining instructions in the function from being executed.

$$\frac{\begin{array}{c} env, ctx, D \models I_1 \Rightarrow^I env' \qquad env' \neq (\top, ...) \\ env', ctx, D \models I_2 \Rightarrow^I env'' \end{array}}{env, ctx, D \models I_1\,;I_2 \Rightarrow^I env''}$$

$$\frac{env, ctx, D \models I_1 \Rightarrow^I env' \quad env' = (\top, ...)}{env, ctx, D \models I_1\,;I_2 \Rightarrow^I env'}$$

Loop instruction. The execution of the `while` (exp_1) I_1 `end;` loop instruction is described by the three rules below.

Its execution begins with the evaluation of the condition expression exp_1, as specified in the first premise of each rule. If the value of exp_1 is ff, then, according to the first rule, the `while` instruction ends. If the value of exp_1 is tt then, according to the second premise of the second and third rule, the instruction I_1 is executed. When I_1 finishes and the resulting environment is a pair $(\top, val)$, then the `while` instruction finishes (see the third rule). Otherwise, according to the second rule, the `while` instruction is executed once again in the new environment.

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} \mathit{ff}}{env, ctx, D \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end} \ \Rightarrow^I env}$$

$$\frac{\begin{array}{c} env, ctx, D \models exp_1 \Rightarrow^{ex} \mathit{tt} \\ env, ctx, D \models I_1 \Rightarrow^I env'' \qquad env'' \neq (\top, val) \\ env'', ctx, D \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end} \ \Rightarrow^I env''' \end{array}}{env, ctx, D \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end} \ \Rightarrow^I env'''}$$

$$\frac{\begin{array}{c} env, ctx, D \models exp_1 \Rightarrow^{ex} \mathit{tt} \\ env, ctx, D \models I_1 \Rightarrow^I env'' \qquad env'' = (\top, val) \end{array}}{env, ctx, D \models \texttt{while (}exp_1\texttt{)}\ I_1\ \texttt{end} \ \Rightarrow^I env''}$$

3.4. Expression evaluation rules

We now introduce the expression evaluation rules.

Local identifier evaluation. The evaluation of a local identifier (a local variable or a function parameter) returns the value of the given identifier in the present environment env.

$$\frac{val = env(\mathit{VarName})}{env, ctx, D \models \mathit{VarName} \Rightarrow^{ex} val}$$

Evaluation of binary operations. These evaluation rules are straightforward.

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} val_1 \qquad env, ctx, D \models exp_2 \Rightarrow^{ex} val_2 \qquad val_3 = \mathit{Plus}(val_1, val_2)}{env, ctx, D \models exp_1 \texttt{ + } exp_2 \Rightarrow^{ex} val_3}$$

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} val_1 \qquad env, ctx, D \models exp_2 \Rightarrow^{ex} val_2 \qquad val_3 = \mathit{Minus}(val_1, val_2)}{env, ctx, D \models exp_1 \texttt{ - } exp_2 \Rightarrow^{ex} val_3}$$

$$\frac{env, ctx, D \models exp_1 \Rightarrow^{ex} val_1 \qquad env, ctx, D \models exp_2 \Rightarrow^{ex} val_2 \qquad val_3 = \mathit{GreaterThan}(val_1, val_2)}{env, ctx, D \models exp_1 \texttt{ > } exp_2 \Rightarrow^{ex} val_3}$$

Evaluation of constants. These evaluation rules are straightforward.

$$env, ctx, D \models \texttt{true} \Rightarrow^{ex} \mathit{tt} \qquad env, ctx, D \models \texttt{false} \Rightarrow^{ex} \mathit{ff}$$

$$env, ctx, D \models \texttt{0} \Rightarrow^{ex} 0 \qquad env, ctx, D \models \texttt{1} \Rightarrow^{ex} 1 \qquad \ldots$$

Function call. Following the rule below, the evaluation process of an expression of the form $\mathit{funID}\texttt{(}exp_1\texttt{,} \ldots \texttt{,} exp_n\texttt{)}$ proceeds as follows:

1. According to the first assumption, expressions exp_1, ..., exp_n denoting the actual parameters of the function are evaluated, and their values denoted as val_1, ..., val_n.
2. Then, the new environment env' in which the body of the function will be evaluated is defined (according to the second and the third assumption). The environment env' contains the values assigned to two kinds of local identifiers:
   - the names of the function parameters (par_1, ..., par_n) are determined by using the function declaration (by *FunParams*) and mapped to the values of the parameters (val_1, ..., val_n);
   - the names of the local variables declared within the function body $local_1$, ..., $local_k$ (determined using $\mathit{FunLocals}^{\mathit{funID}}$) are mapped to the initial values of their corresponding types.
3. Finally (according to the last assumption), in the environment env' and in the context containing the information about the function (funID), the body ($\mathit{FunInstr}_D^{\mathit{funID}}$) is evaluated yielding the environment $(\top, val')$. Notice that it is safe to assume that the resulting environment has the form $(\top, val')$, which means that a `return` instruction has been executed.

The value val' is the result of the function call expression.

$$\frac{\begin{array}{c} env, ctx, D \models exp_1 \Rightarrow^{ex} val_1 \quad \ldots \quad env, ctx, D \models exp_n \Rightarrow^{ex} val_n \\ (par_1 : T_1, \ldots, par_n : T_n) = \mathit{FunParams}_D^{\mathit{funID}} \qquad (local_1 : T^1, \ldots, local_k : T^k) = \mathit{FunLocals}_D^{\mathit{funID}} \\ env' = \{par_1 \mapsto val_1; \ldots; par_n \mapsto val_n; local_1 \mapsto \mathit{InitVal}(T^1); \ldots; local_k \mapsto \mathit{InitVal}(T^k)\} \\ env', \mathit{funID}, D \models \mathit{FunInstr}_D^{\mathit{funID}} \Rightarrow^{I} (\top, val') \end{array}}{env, ctx, D \models \mathit{funID}\texttt{(}exp_1\texttt{,} \ldots \texttt{,} exp_n\texttt{)} \Rightarrow^{ex} val'}$$

Each rule in our semantics has a specific structure expressing the sequential nature of the language.

Property 1. (Sequential ordering of premises)
Within each rule, the components on the left-hand side of the $\Rightarrow$ symbol in each premise are function of the components present in the left-hand side of $\Rightarrow$ symbol in all the premises occurring earlier within the same rule and of the components on the left-hand side of the $\Rightarrow$ symbol in the conclusion.

As a result, our semantics is *sequential*. To illustrate this, let us consider the following rule (which is a simplified version of the compound instruction rule):

$$\frac{env, ctx, D \models I_1 \Rightarrow^I env' \qquad env', ctx, D \models I_2 \Rightarrow^I env''}{env, ctx, D \models I_1\,;I_2 \Rightarrow^I env''}$$

In order to execute an instruction of the form $I_1\,;I_2$ in env, ctx, D, we first need to execute I_1 and then, in the resulting env', we execute I_2 to obtain env'', which is the result of the execution of $I_1\,;I_2$.

This property is very similar to the one defined for the *L-attributed grammar* by Ibraheem and Schmidt in [7] and explored deeper by Ager in [2].

Moreover, our semantics is also *deterministic*, in the sense that there do not exist two different derivation trees for one judgment. This is because for each form of a judgment there is either only one rule, or if there is more than one rule for a certain kind of judgment, then their form (together with their side-conditions) ensures that those rules are mutually exclusive. For example, consider the three rules for the `while` instruction. All those rules share a premise, which is the evaluation of the boolean condition. After the common premise, they differ in the side-conditions which depend on the result of the evaluation of the common premise. The first rule is applicable only if the boolean condition evaluates to *ff* , while the two other rules only if the boolean condition evaluates to *tt* .

3.5. Properties of the big-step semantics rules

We begin with a property enjoyed by the semantics of LM that reflects how the context is manipulated by the rules.

First of all, notice that most of the rules are such that the evaluation of an occurrence of a term t (an instruction or an expression) is defined by means of the evaluation of its subterms. Moreover, each evaluation of a subterm is performed in the same context in which t is evaluated. More precisely, within each rule used to derive the judgment $...ctx... \models t \Rightarrow ...$, in each premise of form $...ctx'... \models t' \Rightarrow ...$, if t' is a subterm of t, then $ctx' = ctx$.

The only rule in which the premises contain a term which is not a subterm of the term present in the conclusion is the function call. It is easy to see that, within this rule, the premise which contains a term which is not a subterm uses a context which is defined independently from the context used in the conclusion of the rule.The only case when this context is the same as in the conclusion is the one of a recursive function call. Moreover, the rules which have new contexts in their premises, as well as new terms which are not subterms of the terms in the conclusion, are the function call evaluation and the program evaluation rules only, where the latter generates the first context of the program. Therefore, we have the following property.

Property 2. (Structure of a derivation)
For each program P and the derivation tree of its final judgment and for each occurrence of a judgment $...ctx... \models t \Rightarrow ...$ in the derivation tree such that t is an instruction or an expression, we have that this judgment occurs in a subtree which starts from a node of the form $...ctx... \models t' \Rightarrow^I ...$, such that t' is:

- either the body of the function in which t occurs,
- or the main expression of the program in which t occurs.

We state another property of the semantics which says informally that if the evaluation of a function body terminates, then it terminates with an environment containing its value.

Property 3. (Function body evaluation)
For each environment env, context ctx, declarations D and function identifier *funID*, if we have:

$$env, ctx, D \models \textit{FunInstr}_D^{\textit{funID}} \Rightarrow^I env'$$

for some env', then env' is a pair of the form $(\top, val)$, for some value val.

In order to prove this, it is enough to see how a derivation tree for $\textit{FunInstr}_D^{\textit{funID}}$ looks like. First recall that $\textit{FunInstr}_D^{\textit{funID}}$ is a compound instruction of the form I`;` `return` v_I`;`, where $v_I = \textit{InitVal}(\textit{RetType}_D^{\textit{funID}})$. Recall also that the semantics of the compound instruction has two cases. In the first case, when the execution of I returns an environment of the form $(\top, val)$, the same environment is the result of the whole function body. In the second case, when the result of the execution of I has a different form, then the result of the whole function body is equal to the result of the execution of `return` v_I`;`, which is $(\top, v_I)$. This concludes the proof.

4. Type checking rules

In this section we introduce the typing rules.

4.1. Judgments

The type checking rules in LM are used to derive judgments of the following kinds (in the judgments, ctx is an element of the set *Contexts*, D is an element of the set $Decls$ and T is either type `Int` or type `Bool`).

A declaration of a function is type safe. This judgment says that a function declaration *funDecl* is type safe in a program containing the function declarations represented by D.

$$D \models_D \textit{funDecl} \ : \ OK$$

An instruction is type safe. This judgment says that an instruction is type safe within the context ctx of the function in which it occurs.

$$ctx, D \models_I I \ : \ OK$$

By knowing the context, we have the information about the types of the identifiers, as defined by the function $\textit{IdTypes}_D(ctx)$.

An expression is type safe and has a type T. This judgment says that within the context ctx of a function, an expression exp is type safe and has type T.

$$ctx, D \models_{ex} exp : T$$

A program is type safe. This judgment says that a program consisting of a series of function declarations $F_1, ..., F_n$ and a main expression E is type safe and has type T.

$$\models_P F_1; F_2; ...F_n; E : T$$

Similarly as for the operational semantics rules, we use two different names for all the assumptions occurring in the type checking rules. We use the name *premise* for each judgment of the form ... $\vdash_*$ ..., where $\vdash_*$ is one of the symbols $\models_I, \models_D, \models_{ex}$. All the other assumptions are called *side-conditions*.

4.2. Type checking of program and function declarations

Type checking of a program. According to the rule, a program is type safe if all function declarations occurring in it are type safe, as well as its main expression. Additionally, this rule also verifies that all the function identifiers are distinct (see the definition of $DeclsVal$).

$$\frac{D = DeclsVal(F_1...F_n) \qquad D \models_D F_1 \ : \ OK \quad ... \quad D \models_D F_n \ : \ OK \quad \top, D \models_{ex} exp : T}{\models_P F_1; F_2; ...F_n; exp : T}$$

4.2.1. Type checking of a function declaration

According to the rule, a declaration of a function *funID* is type safe if the following conditions are met:

- In the context of the declared function, the instructions in its body are type safe.
- The names of the function parameters are different from the names of the local variables.
- The names of the function parameters are distinct, as well as the names of the local variables.

$$\frac{funID, D \models_I I : OK \qquad \{v_1, ..., v_k\} \cap \{par_1, ..., par_n\} = \emptyset \qquad \forall_{i,j}(par_i = par_j \vee v_i = v_j) \Rightarrow i = j}{D \models_D T^1 \ funID \ \texttt{(} par_1 : T_1; ...; par_n : T_n \texttt{)} \ v_1 : T_{v1} ..., v_k : T_{vk} \ \texttt{begin} \ I \ \texttt{end} \ : OK}$$

4.3. Type checking of instructions

In this section we present the rules which are used to verify if the instructions are type safe.

Type checking of an assignment. An assignment instruction of the form $VarName \ \texttt{:=} \ exp$ is type safe if the assigned expression exp has the type $IdTypes_D(ctx)(VarName)$. We recall that $IdTypes_D(ctx)(VarName)$ denotes the type occurring in the declaration of variable $VarName$ in the context ctx.

$$\frac{ctx, D \models_{ex} exp : IdTypes_D(ctx)(VarName)}{ctx, D \models_I VarName \ \texttt{:=} \ exp \ : \ OK}$$

Type checking of a compound instruction. A compound instruction $I_1\,;I_2$ is type safe in a context ctx if both I_1 and I_2 are type safe in the context ctx.

$$\frac{ctx, D \models_{I} I_1 \;:\; OK \qquad ctx, D \models_{I} I_2 \;:\; OK}{ctx, D \models_{I} I_1\,;I_2 \;:\; OK}$$

Type checking of a conditional instruction. A conditional instruction having the form `if` exp_1 `then` I_1 `else` I_2 `end` is type safe in context ctx if the following conditions are met:

- Expression exp_1 (denoting the logical condition) has type `Bool`.
- Instructions I_1 and I_2 are both type safe.

$$\frac{ctx, D \models_{ex} exp_1 : \texttt{Bool} \qquad ctx, D \models_{I} I_1 \;:\; OK \qquad ctx, D \models_{I} I_2 \;:\; OK}{ctx, D \models_{I} \texttt{if}\ exp_1\ \texttt{then}\ I_1\ \texttt{else}\ I_2\ \texttt{end} \;:\; OK}$$

Type checking of a loop instruction. A loop instruction of the form `while` (exp_1) I_1 `end` is type safe if the condition exp_1 has type `Bool` and the instruction I_1 is type safe .

$$\frac{ctx, D \models_{ex} exp_1 : \texttt{Bool} \qquad ctx, D \models_{I} I_1 \;:\; OK}{ctx, D \models_{I} \texttt{while}\ (exp_1)\ I_1\ \texttt{end} \;:\; OK}$$

Type checking of a return instruction. The `return` instruction in a function is type safe if the returned expression exp has the same type of the return type of the function.

$$\frac{funID, D \models_{ex} exp : RetType_D^{funID}}{funID, D \models_{I} \texttt{return}\ exp \;:\; OK}$$

4.4. Type checking of expressions

Type checking of an identifier.

$$ctx, D \models_{ex} VarName : IdTypes_D(ctx)(VarName)$$

Type checking of constants.

$$ctx, D \models_{ex} \texttt{true} : \texttt{Bool} \qquad ctx, D \models_{ex} \texttt{false} : \texttt{Bool} \qquad ctx, D \models_{ex} \texttt{0} : \texttt{Int} \qquad ctx, D \models_{ex} \texttt{1} : \texttt{Int} \qquad \ldots$$

Type checking of binary operators.

$$\frac{ctx, D \models_{ex} exp_1 : \texttt{Int} \qquad ctx, D \models_{ex} exp_2 : \texttt{Int}}{ctx, D \models_{ex} exp_1\ \texttt{+}\ exp_2 : \texttt{Int}} \qquad \frac{ctx, D \models_{ex} exp_1 : \texttt{Int} \qquad ctx, D \models_{ex} exp_2 : \texttt{Int}}{ctx, D \models_{ex} exp_1\ \texttt{-}\ exp_2 : \texttt{Int}}$$

$$\frac{ctx, D \models_{ex} exp_1 : \texttt{Int} \qquad ctx, D \models_{ex} exp_2 : \texttt{Int}}{ctx, D \models_{ex} exp_1\ \texttt{>}\ exp_2 : \texttt{Bool}}$$

Type checking of a function call expression. According to the rule, a function call expression of the form $\mathit{funID}\texttt{(}exp_1, ..., exp_n\texttt{)}$ has type $\mathit{RetType}_D^{\mathit{funID}}$ if the following conditions are met:

- The declaration of the function funID contains the declaration of parameters with types $T^1,\ ...,\ T^n$.
- Each of the expressions $exp_1, ..., exp_n$, denoting the values of the function parameters, has the corresponding type $T^1, ..., T^n$.

$$\frac{\begin{array}{c}\texttt{(}par_1\texttt{:}T^1\texttt{,}\ ...\texttt{,}\ par_n\texttt{:}T^n\texttt{)} = \mathit{FunParams}_D^{\mathit{funID}}\\ ctx, D \models_{ex} exp_1 : T^1 \qquad ... \qquad ctx, D \models_{ex} exp_n : T^n\end{array}}{ctx, D \models_{ex} \mathit{funID}\texttt{(}exp_1, ..., exp_n\texttt{)} : \mathit{RetType}_D^{\mathit{funID}}}$$

4.5. Properties of the type system

In this section we present two properties of the structure of a typing derivation.

Notice that each rule typing a term t is *syntax driven*, which means that:

- Each premise of the rule has the form *the term t' is typeable*, where t' is a subterm of t.
- All premises of the rule contain different subterms of t and if a subterm t' is syntactically equal to a subterm t'', then these are different occurences of the same term; this implies that a term occurring in a premise is never a subterm of a term occurring in another premise.
- Each subterm of t occurs in exactly one premise of the rule.

The type system enjoys, in fact, the following property.

Property 4. (Syntax-driven type checking)
For each program P such that $\models_P P : OK$, and each occurrence of an instruction I in P, the derivation tree of $\models_P P : OK$ contains exactly one judgment of the form $ctx, D \models_{ex} I : T$, for some ctx, D. Similarly, for each occurence of an expression exp in P, the derivation tree of $\models_P P : OK$ contains exactly one judgment of the form $ctx, D \models_{ex} exp : T$, for some ctx, D.

From now on, for each program P such that $\models_P P : OK$ and each expression exp or instruction I occurring within, we will use the notion of *type checking context* of an expression exp or an instruction I to denote the context ctx in a judgment of the form $ctx, D \models_I I : OK$ or $ctx, D \models_I exp : T$ occurring within the derivation of $\models_P P : OK$. Since all type checking rules are syntax driven, we know that there is exactly one such judgment for each occurrence of a subterm in the program.

Additionally, notice that each rule used to derive a judgment of the form $ctx, D \models_I I : OK$, or $ctx, D \models_{ex} exp : T$ uses in all its premises the same context ctx. As a result, we know that each judgment for an instruction I or an expression exp contains the same context as the one present in the judgment saying that the function containing I or exp is type safe. Therefore, the following property holds.

Property 5. (Type checking context of instructions and expressions)
Every judgment of the form $ctx, D \models_{ex} exp : T$ or $ctx, D \models_I I : OK$ occurring in the derivation tree of a judgment $\models_P P : OK$ is either contained within a subtree rooted with a judgment of the form $D \models_D fun : OK$ and:

- exp (resp. I) occurs within the body of fun and

- ctx occurs in the premise of the rule to prove the judgment $D \models_D fun \ : \ OK$ (see Section 4.2.1),

or exp (resp. I) occurs within the main expression of P and $ctx = \top$.

5. The classical big-step semantics approach to type safety

We discussed in the introduction the drawbacks of the classical formulation of type soundness with big-step semantics. To better understand why such a formulation does not capture all the problems, we present the following example.

Consider a wrong typing rule for `if` instruction (see Section 4 for the correct version):

$$\frac{ctx, D \models_I I_1 \ : \ OK \qquad ctx, D \models_I I_2 \ : \ OK}{ctx, D \models_I \texttt{if} \ exp_1 \ \texttt{then} \ I_1 \ \texttt{else} \ I_2 \ \texttt{end} \ : \ OK}$$

This rule does not verify that the type of the condition expression is `Bool`. As a result, a program which uses an integer expression as condition will pass the type checking correctly. However, such a program will get stuck when reaching the point of evaluation of this condition, because the evaluation rules for the `if` instruction requires this condition to evaluate to tt or ff .

Nevertheless, the typical type soundness statement "If the program terminates, then it does so with a correct value" still holds even in the presence of the wrong rule.

This example hints that the classical notion of type soundness is not strong enough, because, as pointed out in the introduction, properties that concern the intermediate configurations that may emerge in the course of a program evaluation are disregarded by the classical approach. Therefore, it is more difficult to express the subject reduction and the type soundness of a language with a big-step operational semantics.

These issues can be solved partially by adding special values representing the notion of getting stuck and by adding rules which propagate such special values in the evaluation process (like in [3]). Then, the fact of getting stuck is modelled by the reduction of a program to a special value. However, this approach requires additional artificial rules, which often means doubling the number of rules in the semantics. Moreover, it cannot be verified easily if the special values and the additional rules cover all the cases in which the program may get stuck during its execution.

These problems have already been pointed out in the literature and there exist some proposals of solutions in [10, 7, 2, 6, 13]. The first two use a co-inductive interpretation of the big-step style semantics in order to formalize the notion of non-termination of a program. However, this also requires additional rules and uses a less intuitive interpretation of the semantics. We shall discuss the last three ones after having introduced our approach, in Section 6.4. In particular, the solution in [2] is in many respects similar to ours.

6. A novel approach to type safety

To overcome the limitations of the big-step semantics classical approach, we have developed a new approach to modelling type safety. It exploits the sequentiality of the rules (see Section 3) to discover the sequence of intermediate steps of a program execution inside the derivation tree of the program, in order to model infinite executions and to formulate statements concerning a program getting stuck, without adding special values and rules to the semantics.

6.1. Evaluation by a derivation-search algorithm

If we take a closer look at the shape of the LM semantics rules (that endows sequentiality as well as determinism, as shown in Section 3), therefore at the shape of derivation trees, it is easy to see that there exists a greedy *derivation-search* partial algorithm. By derivation-search algorithm we understand an algorithm which, given a partial judgment of the form $env, ctx, D \models E \Rightarrow^{ex} ?$, checks if there exists a value val, such that $env, ctx, D \models E \Rightarrow^{ex} val$ is derivable. The algorithm computes val, together with the derivation tree for the judgment.

The fact that the algorithm is partial means that if the program terminates (i.e., if there exists a derivation tree for the partial judgment of the form ... $\models E \Rightarrow^{ex} val$) then the algorithm will find the derivation tree and the value val. Otherwise the algorithm may loop.

Furthermore, this algorithm computes all the intermediate steps (as a sequence of *program configurations*), therefore it mimics the execution of a program. Hence, if the execution of the program is infinite, the execution of the derivation-search algorithm is also infinite.

As a result, the analysis of the sequences of program configurations (called later *traces*, see Section 6.3) will allow us to state and prove properties concerning program non-termination and to analyze the process of getting stuck, by formulating statements of the form “a program gets stuck in a certain configuration”. In particular, we will be able to state and prove a subject reduction property (see Section 7), saying that all sequences of program configurations of a type-safe program are type safe, and to state and prove the type soundness of LM, by showing that a type-safe program will not get stuck.

6.2. The derivation-search algorithm

The question answered by the big-step operational semantics is "does a program terminate and, if it does, with what value?" More precisely, the semantics is used to build a derivation for partial judgments of the form ... $\models E \Rightarrow^{ex} ?$ and to find the value of $?$.

There exists a partial deterministic greedy algorithm which finds the answer to the question above. This algorithm, given a program which terminates, builds a derivation tree for such a judgment by recursively constructing its subtrees. The algorithm is implemented as a recursive procedure (from now on, we will use the terms “algorithm” and “procedure” interchangeably). The algorithm takes as a parameter a partial judgment of the form $X \models Y \Rightarrow ?$, where $\Rightarrow$ is one of the following symbols $\{\Rightarrow^{I}, \Rightarrow^{ex}, \Rightarrow^{P}\}$ and X, Y are given. If there exists Z such that $X \models Y \Rightarrow Z$ has a derivation tree, then the procedure finds this Z, together with the derivation tree for that judgment.

The algorithm starts from the partial judgment saying that the program evaluates to an unknown value. Recursively, for each partial judgment, it chooses the applicable rule (or set of rules), which can be used to derive that kind of judgment. Notice that for most kinds of judgments there exists only one rule which can be used to derive such a judgment, which makes this choice obvious. Then, if a rule is available, the algorithm traverses all the premises of that rule (which are ordered sequentially, as said in Section 3). For each premise, a partial judgment is calculated and, for each partial judgment, the algorithm recursively calls itself in order to find the value of the $?$ occurring in that partial judgment and to build the derivation tree for that premise.

Recall that in each rule the first premise of the form $...env_1... \models t \Rightarrow (..., env_2, ...)$ is the one in which $env_1 = env$. Moreover, each premise after the first one uses, on the left-hand side of the $\models$ symbol, the environment which functionally depends on the environments and the values occurring in the previous premises and the resulting environment of the conclusion judgment of the rule is a function of the resulting environment of the last premise.

Once the premises are ordered, we execute recursively the algorithm for those premises. If the recursive execution of the algorithm returns with the derivation tree and the value of the environment used on the left-hand side of a given premise, then this environment is used, in turn, to execute the algorithm for the next premise. If the algorithm finds the derivation tree and the resulting environment for the last premise of the target rule, then it returns the tree for the target rule, with the resulting environment of the last premise used as the resulting environment of the conclusion of the rule. The algorithm terminates thanks to rules which have no premises, even though those can contain some side-conditions.

The only case left which needs additional treatment is the one in which the algorithm is executed for a judgment that can be derived using more than one rule. There exist three different forms of judgments derivable using more than one rule: the compound, `if` and `while` instructions. The algorithm deals with that three cases in the following way:

- Judgments of the form $(...,...,...) \models I_1 \ ; \ I_2 \Rightarrow^I ?$ can be derived using two rules. However, when we order the premises of both of those rules, we can see that they begin with an identical premise, concerning the I_1 instruction. Therefore, the procedure is executed recursively for that first premise, postponing the choice of the rule. Then, when this recursive call for I_1 returns with the value of the value of local environment, the procedure chooses the appropriate rule using the value of the returned environment. This choice can be performed deterministically using the mutually exclusive conditions occurring within those rules (checking whether $env' = (\top, val)$ or not).
- Judgments of the form $(...,...,...) \models$ `if` (exp) `then` $... \Rightarrow^I ?$ can be also derived using two rules and the procedure deals with them similarly as with a compound instruction. Both of the applicable rules begin with the same premise, concerning the condition expression exp. Therefore, the algorithm executes on the first premise. Once this recursive call terminates, depending on the returned value (whether it is tt or ff) and the mutually exclusive conditions, the algorithm chooses the appropriate rule.
- Judgments of the form $(...,...,...) \models$ `while` (exp) I_1 `end` $\Rightarrow^I ?$, can be derived using three rules, however we can deal with them as in the previous cases. Using twice the technique of delaying the choice of the rule until a certain premise is evaluated, the algorithm limits the set of applicable rules until only one rule is left.

  First of all, notice that all three rules applicable begin with the same premise, concerning the evaluation of the logical expression exp. Therefore, the algorithm begins with the recursive call evaluating the expression exp. When the recursive call finishes and this expression was evaluated to ff, then the first rule is applied. If the expression was evaluated to tt, then we still have the second and third rule to choose from. However, those two rules also share the second premise. Therefore the algorithm delays further the choice between those two rules and executes itself recursively for the second (common) premise to evaluate the instruction. Then, when the recursive call terminates and returns the environment, the algorithm chooses the appropriate rule using the condition referring to the environment (whether $env'' = (\top, val)$ or not).

We present a detailed definition of the algorithm using pseudo-code. For simplicity, in this definition we do not build the derivation tree itself. This procedure just checks whether such a derivation exists for a partial judgment .... $\models t \Rightarrow ?$, and, if it exists, then it also calculates the value of ? in the partial judgment[2]

[2]All the judgments in the procedure are in fact partial judgments, therefore we have decided to skip the ?.

Notice that the form of a partial judgment determines whether it describes an evaluation process of an expression, an instruction, a declaration or a program (depending on whether $\Rightarrow$ is either $\Rightarrow^{ex}$ or $\Rightarrow^{I}$). Therefore, to obtain a more intuitive and readable presentation, we have split the derivation procedure into three procedures, `SearchE`, `SearchI` and `SearchP`, responsible for expressions, instructions and programs, respectively. Moreover, each occurrence of the recursive call to the derivation-search procedure has been replaced with a call to one of the mentioned procedures. The procedure to be called at each point depends on the form of the argument of such a recursive call. Finally, the execution of the program is defined by the execution of the procedure `SearchP(`$F_1; ...F_n; E \Rightarrow^P$`)`.

A full definition of the derivation-search procedure written in pseudo-code is given in Figures 5, 6 and 7. This code contains keywords which are parts of the derivation-search procedure in pseudo-code, and also keywords and operators which are parts of the terms written in LM (which serve as values in the program in pseudo-code, and as patterns in pattern-matching). Therefore, to distinguish, we use underline to denote keywords in the pseudo-code (like `procedure`, `case`) and normal text for the code in LM. In most of the pseudo-code we exploit the pattern matching technique. We use it in the `case` statements, as well as in the assignments `:=`. As usual (like in SML[11]), we assume that if the value does not match any pattern, then the program fails with an error.

Procedures `SearchE`, `SearchI`, `SearchP` work in the following way:

- First, the pattern matching on the syntactical form of the partial judgment is performed (via the main `case` switch of each procedure) to choose the appropriate rule.
- Then, for each form of judgment, the premises of the rule are processed from left to right.
- In most cases, when there is only one rule possible for the judgment, the sequence of the instructions within the appropriate branch of the `case` contains one instruction per each premise of the rule.
- In the case of more than one applicable rule (for the three cases described above), the procedures first evaluate the premises which can be evaluated without the choice of a rule, and then the procedures choose the appropriate rule and continue with the evaluation of the other premises.

The following property holds for the algorithm.

Property 6. (Soundness and completeness of the algorithm)
For each program P, the following statements are equivalent:

- There exists a derivation tree for the judgment $\models P \Rightarrow^P val$.
- The derivation-search algorithm `SearchP(` $\models P \Rightarrow^P$ `)` terminates and returns the value val.

This property can be easily verified, because the derivation tree can be transformed in a straightforward way into the structure of the recursive calls of the derivation-search procedure and vice versa.

Similarly, for each program which has an infinite execution, this procedure will loop, too.

The derivation-search algorithm models the the execution of a program, together with its intermediate steps. Therefore we say that the instruction I_1 *executes* the instruction I_2, if during the activation of the algorithm for a partial judgment of the form ... $\models I_1 \Rightarrow^I ?$, the algorithm was activated for a partial judgment ... $\models I_2 \Rightarrow^I ?$. Similarly, we say that during the execution of program P, an instruction I was *executed in an environment* env, if the procedure was called for the partial judgment $..., env, ... \models I \Rightarrow^I ?$.

The detailed definition of the execution process based on this algorithm is present in Section 6.3.

```
procedure searchI(j)
 case j of
  $env, ctx, D \models VarName$:=$exp \Rightarrow^I$:
      $val$ := SearchE($env, ctx, D \models exp \Rightarrow^{ex}$);
      res := $env\{VarName \mapsto val\}$;
  $env, ctx, D \models$ if $exp_1$ then $I_1$ else $I_2$ end $\Rightarrow^I$:
      $val$ := SearchE($env, ctx, D \models exp_1 \Rightarrow^{ex}$);
      case $val$ of $tt$:($env''$) := SearchI($env, ctx, D \models I_1 \Rightarrow^I$);
                    $ff$: ($env''$) := SearchI($env, ctx, D \models I_2 \Rightarrow^I$);
      endcase;
      res := $env''$;
  $env, ctx, D \models$ return $exp \Rightarrow^I$:
      $val'$ := SearchE($env, ctx, D \models exp \Rightarrow^{ex}$);
      res := $(\top, val')$;
  $env, ctx, D \models I_1; I_2 \Rightarrow^I$:
      $env'$ := SearchI($env, ctx, D \models I_1 \Rightarrow^I$);
      case $env'$ of $(\top, x)$:res := $env'$;
                else: res := SearchI($env', ctx, D \models I_2 \Rightarrow^I$);
      endcase;
  $env, ctx, D \models$ while ($exp_1$) $I_1$ end $\Rightarrow^I$:
      $v$ := SearchE($env, ctx, D \models exp_1 \Rightarrow^{ex}$);
      case $v$ of $ff$:res := $env$;
                 $tt$: $env''$ := SearchI($env, ctx, D \models I_1 \Rightarrow^I$);
                     case $env''$ of $(\top, val)$:res := $env''$;
                                else: res := SearchI($env'', ctx, D \models$ while ($exp_1$) $I_1$ end $\Rightarrow^I$);
                     endcase;
      endcase;
 endcase;
 return res;
endprocedure
```

Figure 5. The derivation-search procedure for instructions

```
procedure searchE(j)
 case j of
  $env, ctx, D \models$ true $\Rightarrow^{ex}$:        res := $tt$;
  $env, ctx, D \models$ false $\Rightarrow^{ex}$:       res := $ff$;
  $env, ctx, D \models VarName \Rightarrow^{ex}$:     res := $env(VarName)$;
  $env, ctx, D \models exp_1$ + $exp_2 \Rightarrow^{ex}$: $val_1$ := SearchE($env, ctx, D \models exp_1 \Rightarrow^{ex}$);
                                $val_2$ := SearchE($env, ctx, D \models exp_2 \Rightarrow^{ex}$);
                                res := $Plus(val_1, val_2)$ ;
  $env, ctx, D \models exp_1$ > $exp_2 \Rightarrow^{ex}$: $val_1$ := SearchE($env, ctx, D \models exp_1 \Rightarrow^{ex}$);
                                $val_2$ := SearchE($env, ctx, D \models exp_2 \Rightarrow^{ex}$);
                                res := $GreaterThan(val_1, val_2)$ ;
  $env, ctx, D \models funID(exp_1,\ ...,\ exp_n) \Rightarrow^{ex}$:
    for i := 1 to n do
         $val_i$ := SearchE($env, ctx, D \models exp_i \Rightarrow^{ex}$);
    $(par_1 : T_1,\ ...,\ par_n : T_n)$ := $FunParams_D^{funID}$;
    $(local_1 : T^1,\ ...,\ local_k : T^k)$ := $FunLocals_D^{funID}$;
    $env'$ := $\{par_1 \mapsto val_1; ...; par_n \mapsto val_n; local_1 \mapsto InitVal(T^1); ...; local_k \mapsto InitVal(T^k)\}$;
    $(\top, val)$ := SearchI($env', funID, D \models FunInstr_D^{funID} \Rightarrow^I$);
    res := $val$;
 endcase;
 return res;
endprocedure
```

Figure 6. The derivation-search procedure for expressions

```
procedure SearchP(j)
 case j of
  $\models \overrightarrow{F}; E \Rightarrow^P$ : $D$ := $DeclsVal(\overrightarrow{F})$ ;
                  res := SearchE($\emptyset, \top, D \models E \Rightarrow^{ex}$) ;
 endcase;
 return res;
endprocedure
```

Figure 7. The derivation-search procedure for programs

We state now a property enjoyed by the semantics and the type system of LM. This property reflects the correspondence between: (*i*) the context in which an expression or an instruction is evaluated; and (*ii*) the context in which the type checking of an instruction is performed.

Property 7. (Context consistency)
For each type-safe program P and for each activation of the procedure occurring during the execution of the derivation-search algorithm for P such that the parameter of this activation is a partial judgment of the form ...ctx... $\models t \Rightarrow ?$, where $\Rightarrow \in \{\Rightarrow^I, \Rightarrow^{ex}\}$, we have that the context ctx is a type checking context of the expression or instruction t (see the definition in Section 4).

To prove this property, we first recall Property 2. This property seen in terms of the derivation-search algorithm can be formulated as follows: When the algorithm is executed for a partial judgment of the form ... $\models t \Rightarrow ?$ (where t is an instruction or an expression), this is initiated (directly or indirectly) by the execution of the call for the partial judgement of the function body or of the main expression in which t occurs, and the partial judgment of t uses the same context in which the function body or the main expression is evaluated.

Then consider Property 5, which says that the type checking context of an instruction or an expression is a type checking context of the function body or of the main expression in which it occurs.

Notice also that the context used to type check the body of the function is the same context used to execute the body of that function.

All the three above facts proves the context consistency property.

6.3. Execution traces

Using the definition of the derivation-search algorithm, we define the notion of *execution trace*, or *trace* for short. A trace is a sequence (finite or infinite) of *program configurations*, which represents the execution of the program.

Each *program configuration* represents a step of the program execution. It contains the values of the local variables, stored in an environment, the information about the function or main expression that is currently executed, represented by a context, and which subterm of the body or of the expression is being evaluated. A configuration has one of the following forms:

- a tuple of an environment env, a context ctx and a set of function declarations D.
- a tuple of an environment env, a context ctx, a value val, an expression e and a set of function declarations D.
- a tuple of a value val, an expression e and a set of function declarations D.
- the empty set.

We will say that a configuration C_1 is a *subconfiguration* of a configuration C_2 if $C_1 = (env, ctx, D)$ and $C_2 = (env, ctx, val, exp, D)$, or $C_1 = (val, exp, D)$ and $C_2 = (env, ctx, val, exp, D)$.

The *execution trace* of a program is a sequence of program configurations built according to the extension of the derivation-search algorithm described in the following. A variable `tr` is declared, initially equal to an empty sequence of program configurations. Then the algorithm adds two program configurations to the tail of the sequence `tr` (by the operation `tr.add(...)`). One configuration, said *opening configuration*, is added to `tr` *at the beginning* of the execution of the call for a certain rule. This configuration contains the information about the state of the program before the rule is applied and therefore it is constructed from the information present on the left-hand side of the partial judgment supplied to the procedure call as an argument. The second configuration, called *closing configuration*, is added to `tr` *at the end of the call*, i.e., at the moment when the algorithm finds the value of ? in $X \models Y \Rightarrow$?. The closing configuration is built from the value of ? and from X. We use in the closing configuration those components of X which have not been modified by the evaluation of the judgment, thus are not part of ?. For example, the expression evaluation judgment $X \models exp \Rightarrow^{ex}$? does not contain a new environment on the right-hand side of $\Rightarrow^{ex}$, therefore the closing configuration for this expression uses the environment occurring in X. Notice that this environment occurs also in the opening configuration added by this call.

In the trace being generated, between an opening configuration and a matching closing configuration, there are the configurations generated by the nested activations of the procedure. An example of a trace and its construction process is shown in Figure 10.

The form of the opening and closing configurations depends on the judgment which was the argument of the derivation-search procedure. Figure 8 shows which form of program configuration is used as opening and closing configurations for each form of judgment.

judgment	**opening configuration**	**closing configuration**
$env, ctx, D \models instr \Rightarrow^{I} env'$	(env, ctx, D)	(env', ctx, D)
$env, ctx, D \models exp \Rightarrow^{ex} val$	(env, ctx, D)	(env, ctx, val, exp, D)
$\models \overrightarrow{F}; exp \Rightarrow^{P} val$	$\emptyset$	(val, exp, D)

Figure 8. Program configurations

The extended version of the derivation-search algorithm is shown in Figure 9.

We will use the notion of *opening configuration of the term t* or *closing configuration of the term t* to denote the opening or closing configuration added to the trace by the recursive call which has as its argument a partial judgment of the form ... $\models t \Rightarrow$?. We use this notion in cases when it is clear from the context what the exact form of the judgment is.

We will also use the notion of *enclosing call* of a configuration C to denote the activation of the procedure which called the procedure which, in turn, added the configuration C to the trace. We also use the notion of *enclosing judgment* of the configuration C to denote the judgment being the parameter of the enclosing call.

Finally, depending on the behavior of the derivation-search algorithm, we have one of the following situations:

- When the algorithm terminates its execution (either successfully, or because of an error), then the value of the variable `tr` at the end of its execution is the trace of the program.
- When the algorithm has an infinite execution, then the program trace is an infinite sequence understood as the lowest upper bound of the sequence `tr`.

Informally, we say that a program *terminates successfully* when the evaluation of the main expression E of the program terminates.

Formally, there are three definitions of the notion of a program *terminating successfully* (it is easy to see that those are equivalent):

1. The trace contains the closing configuration generated by the first call of the derivation-search algorithm, which is the one responsible for the evaluation of the program. In such a case, this closing configuration is the last one in the trace.
2. A final judgment $\models F_1; ...; F_n; E \Rightarrow^P val$ can be derived for some value val.
3. The numbers of closing and opening configurations in the trace are equal.

6.4. Comparison with previous work

Our approach is in many respects similar to that of Ager [2], which presents how to translate any big-step semantics into a stack machine. Our solution does not use a stack machine, but it is based on recursive procedure calls.

However, the most important difference is that Ager's general approach generates a non-deterministic machine for each language, containing some form of judgment for which there exists more than one rule in semantics. On the other hand, even though LM's semantics contains a few judgments derivable using more than one rule, like `if` and `while`, our approach is still deterministic. This is due to the strategy of delaying the choice of the appropriate rule after some parts of it are evaluated. For example, in the case of `if`, we first evaluate the premise which is common to both of those rules, and choose the appropriate rule basing on the result of that evaluation.

We believe that our technique can be applied to all languages, except to those languages with *inherently non-deterministic* semantics, such as some process algebras, where it is not possible to state conditions for choosing among alternative rules. If a language is not inherently non-deterministic, it would suffice to formulate for it a big-step semantics in which alternative rules are selected by delaying the choice of the rule to be applied according to the result of the evaluation of common premises, as we do for our compound, conditional and loop instructions. A further step in studying our approach will be to state a meta-theory for formalizing such intuitions.

There are other works centered on the idea of tracing the intermediate steps of a program execution, in particular there is [6], which uses the concept of *partial proof semantics*, representing steps of execution using proofs with logical variables and subgoals and introducing a proof-theoretic expression of the absence of runtime type-related errors, and [13], which exploits an incremental construction of the execution tree (using a meta-language to build it),which may turn out incomplete if a program on a certain input gets stuck. However, it is not clear how these two solutions can deal with a language with constructs whose semantics has alternative rules to choose from, which is the same limit of Ager's approach.

7. Subject reduction

Using the derivation-search algorithm (see Section 6.2) and the program traces (see Section 6.3), we can formulate and prove the subject reduction property for LM, which states that each configuration in a trace of a type-safe program is type safe.

7.1. Preliminary definitions

7.1.1. Type safety of an environment

We say that an environment *env is type safe with respect to a context ctx and declarations D*, if one of the following conditions holds:

- The environment env is a function from identifier to values, its domain is equal to the domain of $IdTypes_D(ctx)$ and for each $var \in Dom(env)$ we have that $ValType(env(var)) = IdTypes_D(ctx)(var)$.
- The environment env is a pair of the form $(\top, val)$, such that $ctx = funID$ and $ValType(val) = RetType_D^{funID}$.

7.1.2. Type safety of a configuration

We say that a configuration C is *type safe* if one of the following conditions holds:

- The configuration C has the form (env, ctx, D) and the environment env is type safe with respect to the context ctx and declarations D.
- The configuration C has the form (env, ctx, val, exp, D) and:
  - the environment env is type safe with respect to the context ctx and declarations D;
  - the value val is type safe with respect to exp, ctx and D, which means that $ValType(val) = T_1$ such that $ctx, D \models_{ex} exp : T_1$.
- The configuration C has the form (val, exp, D) and:
  - the value val is type safe with respect to exp, ctx and D.
- The configuration C has the form $\emptyset$.

It is easy to see that if C_2 is type safe and C_1 is a subconfiguration of C_2, then C_1 is also type safe.

7.2. Subject reduction formulation

The subject reduction property of LM is formulated as follows.

Property 8. (Subject reduction)
For each type-safe program and a trace built by the derivation-search algorithm for that program, each configuration occurring in the trace is type safe.

In other words, the subject reduction property says that during the evaluation of a type-safe program (which is equivalent to the execution of the derivation-search procedure, *cf.* Property 6), all the local environments and all the values computed are type safe.

7.3. Subject reduction proof

The type safety of an opening configuration depends on the enclosing call of that configuration (see the definition in Section 6.3), because an opening configuration is completely determined by the components of the partial judgment passed as a parameter to the call of the procedure that adds the opening configuration to the trace.

Instead, each closing configuration C, added by an activation a of the procedure, is calculated and determined by a, or partially calculated by the recursive calls performed by a, before this configuration is added to the trace.

The proof is by induction on the sequence of configurations in a trace. We proceed by cases on the form of the judgment and the rule/call to evaluate it and we prove that:

- the opening configurations added by all the recursive calls induced by this call[3] are type safe,

[3]This call plays the role of the enclosing call.

and

- the closing configuration (if any, since a recursive call might loop infinitely) added by this call is type safe;

under the induction hypothesis that all the configurations added to the trace before are type safe.

no enclosing call (base case). There is one opening configuration in each trace for which there is no enclosing call (see the definition in Section 6.3). This configuration is the one added to the trace by the top-level call to the procedure, which has the program judgment as its parameter. It is always the first configuration in each trace and it has the form $\emptyset$, which is type safe.

program evaluation. The opening configuration added by the recursive call for the program evaluation has the form $\emptyset, \top, D$. To prove that this configuration is type safe, we have to verify if the environment is type safe. The domain of an empty environment is empty, while $\mathit{IdTypes}_D(\top)$ is also an empty set, therefore those domains coincide. Furthermore, the environment has no elements to be verified, therefore it is type safe with respect to the context.

The corresponding closing configuration, added to the trace by the program evaluation call, has the form val, exp, D and it is a subconfiguration of the closing configuration added to the trace by the recursive call evaluating the main expression judgment, therefore it is type safe by induction hypothesis.

assignment to a local variable. This rule contains one recursive call (responsible for the evaluation of the expression assigned) and the opening configuration of this judgment is the same as the opening configuration of the enclosing call. Therefore, by induction, this configuration is type safe.

This rule generates a closing configuration, which is constructed from the subconfiguration of the closing configuration generated by the evaluation of the assigned expression exp. The new closing configuration is generated from the mentioned subconfiguration by modifying the environment with respect to $\mathit{VarName}$. Therefore, since the closing configuration of the recursive call is type safe, we only have to show that assignment of val to $\mathit{VarName}$ in the new environment will maintain the type safety of that configuration. Since the closing configuration of the recursive call (evaluating val from exp) is type safe by induction, we know that $\mathit{ValType}(val)$ is equal to the static type of exp. Moreover, since the assignment is type safe, we know that $exp : \mathit{IdTypes}_D(ctx)(\mathit{VarName})$. Therefore we know that $\mathit{ValType}(val) = \mathit{IdTypes}_D(ctx)(\mathit{VarName})$.

`if` **instruction.** This judgment has two rules. Each of those two rules have two premises, therefore two recursive calls:

- The first one is responsible for the evaluation of the boolean condition. This call adds the same opening configuration as the opening configuration of the whole `if` instruction, therefore it is type safe by induction.
- The second one, in both rules, is responsible for the execution of the instructions in the `then` or `else` branch. The opening configuration of the second call is a subconfiguration of the closing one of the first recursive call (which is added earlier in the trace), so it is type safe by induction.

Both of those rules use as their closing configurations the closing configuration of their second recursive call. Therefore by induction such configurations are type safe.

`return` **instruction.** It has one recursive call, which has an opening configuration equal to the opening configuration of the enclosing call, so it is type safe by induction.

This rule generates a closing configuration constructed from the elements of the closing configuration (denoted as C) of the only recursive call (responsible for evaluation of the expression to be returned). The closing configuration of a `return` instruction contains a new environment of the form $(\top, val)$. Therefore we have to prove that the value val used in the environment fulfills all the necessary conditions.

The value val occurs in the configuration C. Therefore by the type safety of this configuration we know that $ValType(val)$ is the type of exp. Moreover, by the fact that this `return` instruction is type safe, we know that $exp : RetType_D^{ctx}$. As a result, we know that the type $ValType(val)$ is equal to $RetType_D^{ctx}$, which concludes the proof for this case.

compound instruction. It has two recursive calls. The first one has the same opening configuration as the enclosing call, which has been added earlier in the trace, so it is type safe by induction. The second recursive call (present only in the first rule) supplies the opening configuration, which is equal to the closing configuration of the first call, which has been also added in the trace earlier, so is type safe by induction.

The closing configuration added by the evaluation of the compound instruction depends on the rule used. The first rule uses the closing configuration of the second recursive call, while the second one uses the closing configuration of the first recursive call. In both cases this is a closing configuration which already has been added to the trace, so it is type safe by induction.

expression evaluation. It has one recursive call with the same opening configuration as the enclosing call, so it is also type safe.

The evaluation of that judgment generates a closing configuration which is a subconfiguration of the closing configuration of the recursive call, therefore this one is also type safe.

`while` **instruction.** It has three recursive calls. The first recursive call evaluates the boolean expression and uses the same opening configuration as the enclosing call, so is type safe by induction. Then, if that expression evaluated to tt then the second recursive call, responsible for the execution of the instruction of the loop, is executed. This second recursive call uses as its opening configuration the one which is a subconfiguration of the closing configuration of the first call, so it is also type safe by induction. Then, if that instruction has not executed a `return` instruction (therefore the environment does not contain the symbol $\top$), then the third recursive call is executed, which repeats the execution of the loop. That call, however, has an opening configuration which is equal to the closing configuration of the second call, so this configuration is type safe by induction.

All of the rules for `while` instruction generate a closing configuration which is either equal, or is a subconfiguration of a closing configuration of the last recursive call. Therefore in every case such a closing configuration is also type safe.

constant. There are four constants: `true`, `false`, `0` and `1`. The evaluation of each of those does not perform any recursive calls, therefore it never plays the role of an enclosing call and so there is no opening configuration to be considered.

The evaluation of each of those constants generates a closing configuration which is built from the opening configuration of the same call (which is type safe by induction), with the addition of a value val and an expression exp. Therefore we just have to show that val is type safe with respect to ctx, exp and D. However, it is easy to see that $ValType(v)$, for $v \in \{\texttt{true}, \texttt{false}, 0, 1\}$ is equal to the type of each of those expressions.

local identifier. The evaluation of such expressions does not perform any recursive calls, therefore it never plays the role of an enclosing call.

The closing configuration generated for such a judgment is the environment of the opening configuration, which is type safe by induction. We have just to show that the returned val, which is the value of env for the given identifier, has a type. In fact, by the type safety of the opening configuration, we know that $ValType(env(VarName))$ is equal to $IdTypes_D(ctx)(VarName)$.

function call. The evaluation of a function call performs a sequence of calls responsible for the evaluation of the parameter values and a call evaluating the body of the function.

The recursive calls responsible for the evaluation of the parameter values use the opening configuration of the function call, thus they are type safe by induction.

The only non-trivial case is the last recursive call, which is the call evaluating the instructions of the function body in the new environment env' constructed in this rule. This is non trivial since the environment here is a function constructed from scratch, therefore we have to prove that it is type safe. In order to prove that env' (which is a function, not a pair of the form $(\top, val)$) is type safe with respect to the context $funID$, first we have to show that its domain matches the definition of environment type safety, and then that its values fulfill all the requirements.

The domain of the environment env' is constructed from two parts: $FunParams_D^{funID}$ and $FunLocals_D^{funID}$. Therefore, the domain of env' is equal to the domain of the $IdTypes_D(funID)$ (see the definition of $IdTypes$ in Section 3). As a result, the domain fulfills the requirements of type safety.

The values of variables in env' also fulfill the conditions of type safety, because:

- The local variables are assigned with initial values, therefore they are trivially type safe.
- The function parameters are assigned with val_1, ..., val_n, which are the respective values of the expressions exp_1, ..., exp_n calculated by the recursive calls. For each such val_i, we have to show that it has the type T^i (where T^i is the declared type of the corresponding function parameter).
  Since the closing configuration of the call evaluating val_i is type safe, we know that $ValType(val_i)$ is the static type of exp_i. Moreover, thanks to the fact that this function call expression is type safe, we know that $exp_i : T^i$, which concludes the proof.

The closing configuration generated by the evaluation of a function call expression has the form $(env,\ ctx,\ val,\ funID(...)\ , D)$. This configuration is type safe for the following reasons:

- The environment env and context ctx are part of the opening configuration of that call, thus env is type safe with respect to ctx by induction.
- The value val is type safe with respect to exp, ctx and D because the closing configuration generated by the evaluation of the function instructions is type safe (by induction), therefore we know that the type $ValType(val)$ is equal to $RetType_D^{funID}$.

binary operators: +, -, >. The evaluation of each of the operators performs two recursive calls. The first recursive call, to build the related opening configuration, uses the opening configuration of the enclosing call, thus is type safe by induction. The second recursive call, to build the related opening configuration, uses a subconfiguration of the closing configuration of the first recursive call, thus is also type safe by induction.

The closing configurations of the binary operators use the environment and the context which are part of the opening configuration of their calls, thus they are type safe by induction. We only have to prove that for the value val being a part of the closing configuration, $ValType(val)$ is equal to the type of the expression.

In the case of the first two operators, the *Plus* and *Minus* functions always return elements of $\mathbb{Z}$, thus they are of type `Int` and the type of such expressions is always `Int` (according to their typing rules), thus such a configuration is always type safe. In the case of the > operator, the result of *GreaterThan* function is either *ff*, or *tt*, and both of them are of type `Bool`, which is the type of every conditional expression, which concludes this case of the proof.

8. Type soundness

In this section we formulate and prove the type soundness property. We state it using the notion of program trace (see Section 6.3) and prove it using the subject reduction property (see Section 7).

8.1. Type soundness formulation

Informally, the type soundness property says that if the program is type safe then during its execution nothing can "go wrong", that is, the execution cannot get stuck because it reached a value of an unexpected type.

Formally, the type soundness property is defined in the following way.

Property 9. (Type soundness)
For each type safe program, one of the following two situations occurs:

- The program terminates successfully (see the definition in Section 6.3).
- The trace generated by the derivation-search procedure is infinite.

8.2. Type soundness proof

We prove the following property which is equivalent to the type soundness defined above.

Property 10. (Type soundness reformulated)
If the trace generated by the derivation-search procedure of a type safe program is finite, then the program terminates successfully.

We prove this property by contradiction. We show that if the program does not terminate successfully, then there does not exist a configuration that is the last one in the trace, i.e., there is always another configuration following, which means that the execution does not get stuck.

In this proof, we use the notion of a *configuration added unconditionally*, or that a recursive call is performed *unconditionally*, to say that it is performed: (*i*) without checking further conditions, and (*ii*) without performing additional calculations which could fail. For example, we say that a recursive call is performed unconditionally when a partial judgment being its parameter consists of an environment and a context calculated before by other recursive calls or supplied as a parameter.

Notice that, for a given configuration C, in order to prove that C is not the last configuration in the trace, it is enough to show that the next configuration is added unconditionally.

Instead, whenever the next configuration after C is not added unconditionally, we prove that C is not the last one in the following way:

- If there is an additional condition which is checked in the semantics rules and thus in the derivation-search procedure, then we prove that this condition is fulfilled and the execution can proceed and add successive configurations to the trace.
- If the environment, context or value being a part of the partial judgment or returned value is calculated by a function, then we show that such environment, context, value is always defined.

We proceed by cases on the form of the judgment and the rule/call to evaluate it and we prove that:

- the opening configuration C added by the call for the judgment is not the last one.

- the closing configuration C added by every recursive call performed by the call for the judgement is not the last one. We analyze each recursive call (corresponding to a premise of the rule) separately.

To prove that the execution cannot get stuck in the given configuration we exploit the type safety of all the configurations in a trace (using Property 8).

This way we cover all configurations in a trace, except for the closing one added by the top-level activation of the procedure (which is not a recursive call). However, this closing configuration is indeed the last one, and it represents a successful program termination.

In all other cases, we will use C to refer to the configuration for which we prove that it cannot be the last one.

assignment to a local variable. The execution of the derivation-search procedure for a variable assignment starts from unconditionally performing the recursive call to evaluate the assigned expression (which in turn adds the subsequent opening configuration), therefore the opening configuration added by the assignment cannot be the last configuration.

On the recursive call for the expression. When the recursive call responsible for the evaluation of the assigned expression ends, then it always does so with the value of the desired form. Therefore, in this case the new environment of the form ($env\{\, VarName \ \mapsto\ val\}$) is always defined, then there will always be a next configuration that is the closing configuration of this assignment.

`if` **instruction.** In the case of an `if` instruction, the execution of the procedure starts unconditionally from the execution of the recursive call to evaluate the boolean expression, so the opening configuration C cannot be the last configuration.

On the recursive call for the boolean condition. The recursive call for an expression evaluation returns a value val. The fact that the closing configuration generated by that call is type safe ensures that the type $ValType(val)$ is equal to the type of the expression. Moreover, by the fact that the conditional instruction is type safe, we know that this expression has type `Bool`. Therefore we know that the type $ValType(val)$ is `Bool`.

On the other hand, we know that `Bool` type has only two values: ff and tt.

Therefore we know that one of the two rules can be applied, so there is always a next configuration which is the opening configuration added by the recursive call for the instruction I_1 or I_2 (depending on the value val).

On the recursive call for the instruction. When the recursive call for instruction in the body of `if` instruction adds its closing configuration and finishes, then the `if` instruction also finishes unconditionally and adds its own closing configuration.

`return` **instruction.** The evaluation of `return` instruction starts unconditionally from the evaluation of the returned expression, so C cannot be the last configuration.

On the recursive call for the expression. When the recursive call for the expression to be returned adds its closing configuration and finishes, the call for the `return` instruction also adds unconditionally its closing configuration and finishes, so the closing configuration added by the recursive call cannot be the last one.

compound instruction. The execution of a compound instruction starts unconditionally from the execution of the first instruction, so C cannot be the last configuration in the trace.

On the recursive call for the first instruction. When the first instruction finishes, we can have one of the two cases. In the first case, the environment returned by the evaluation of the first instruction is a pair, and then the execution of the compound instruction terminates unconditionally , adding to the trace a closing configuration identical to the one of the first instruction.

In the second case (when the environment is not a pair, but a function), the second concatenated instruction is executed, so the configuration which follows is an opening configuration added by the recursive call responsible for the execution of the second instruction.

On the recursive call for the second instruction. When the execution of the second instruction finishes by adding its closing configuration, then the compound instruction also finishes, adding the identical closing configuration to the trace.

expression evaluation. The execution of an expression evaluation instruction starts unconditionally from the evaluation of the expression, so C cannot be the last configuration in the trace.

When this recursive call finishes, then the whole instruction finishes unconditionally as well, therefore the closing configuration of the recursive call cannot be the last one as well.

`while` **instruction.** The execution of `while` loop instruction starts unconditionally from the evaluation of the boolean condition, which in turn starts from adding its opening configuration to the trace, therefore C cannot be the last one.

On the recursive call for the boolean condition expression. When the evaluation of the boolean condition finishes, it finished with either a tt or ff value (as in the case of `if` instruction). Then, depending on that value, we have two cases. In the case of ff, the `while` instruction finishes, which means that the closing configuration for the `while` instruction is added. In the case of the tt value, the next step is the execution of the loop body, which starts unconditionally from the addition of its opening configuration.

On the recursive call for the body of the loop instruction. When the body of the loop finishes, we have one of the two cases.

In the first case, when the returned environment is a pair (representing the execution of `return` instruction), the `while` instruction finishes unconditionally and adds its closing configuration to the trace.

In the second case, when the returns environment is not a pair, the loop is executed unconditionally once again, thus it adds its opening configuration to the trace after C.

On the recursive execution of the loop. When the recursive execution of the loop adds its closing configuration, then the loop execution also finishes and adds its closing configuration.

constant. All three constants evaluate unconditionally to a closing configuration, so the opening configuration cannot be the last one.

The evaluation of a constant does not have recursive calls, thus it is never an enclosing call of any other call.

local identifier. The opening configuration of a local identifier evaluation is always followed by the closing configuration. To see that the closing configuration is always defined it is enough to show that $env(\mathit{VarName})$ is always defined. The fact that the expression is type safe ensures that $\mathit{VarName}$ is in the domain of $\mathit{IdTypes}_D(ctx)$. The subject reduction ensures the fact that the opening configuration is type safe, which in turn ensures that the domain of env is equal to the domain of $\mathit{IdTypes}_D(ctx)$ which finally ensures that $env(\mathit{VarName})$ is defined.

Similarly as with constant evaluation, the evaluation of a local identifier also does not perform any recursive calls.

function call. We have two cases. In the first case, when the function has at least one parameter, the evaluation of the function call starts unconditionally from the evaluation of the first parameter, which adds the subsequent configuration. The second case (when the function does not have any parameter) requires additional explanations, since the judgment being the parameter of the recursive call for the evaluation of the function body contains a new environment, which is build from scratch, and a term, which is not a subterm of the term part of the judgment passed as the parameter to the current call (for the evaluation of the function call). Therefore we have to prove that they are always defined. This reasoning applies also in the presence of parameters, when the parameter to be evaluated is the last one.

In fact, with or without parameters, we know that the function call $funID(...)$ is type safe. This means that *RetType*^{funID} is defined. Therefore also *FunLocals*^{funID} as well as *FunParams*^{funID} are defined. Additionally, val_1 ... val_n are part of closing configurations which are earlier in the trace, thus they are defined, which implies that the newly constructed environment env' is defined. The fact that *funID* is a valid function identifier implies also that *FunInstr*^{funID} is defined. All that ensures that all elements of the partial judgement newly constructed are defined.

On the recursive call for the evaluation of a parameter value. When the evaluation process of a parameter value finishes, then we have two cases, depending on whether the parameter was the last one.

If this is not the last parameter, then the next parameter is evaluated unconditionally and its opening configuration is added after C.

In the case of the last parameter, the next configuration is the opening configuration added by the evaluation of the function body in a new environment. The evaluation of the function body will always start by adding its opening configuration, because the partial judgment passed to that call is always defined. In fact, this is proved in the same way of the case when the function does not have any parameters (see above).

On the recursive call for evaluation of the function body. When the process of the execution of the function body finishes, the whole execution of the function call also finishes, assuming that the body evaluate to a value of the form $(\top, val)$. This is always the case, as a direct consequence of Property 3. Therefore the next configuration will be always the closing one added by the function call recursive call.

binary operators: `+`, `-`, `>`. The evaluation of all expressions built using the binary operators starts unconditionally from evaluating their first argument, which always adds a configuration to the trace.

On the recursive call for the first operand. When the evaluation of the first operand finishes, then the evaluation of the second operand starts unconditionally. Therefore, the closing configuration added by the evaluation of the first operand cannot be the last one.

On the recursive call for the second operand. When the evaluation of the second operand finishes, then the closing configuration of the binary expression is added and it uses the environment and the context used in the opening configuration, however it presents a newly calculated value. Therefore we have to verify that this value is always defined. In the case of all three operators, the result is a function which takes two values in $\mathbb{Z}$.

Moreover, we know that the program is type safe, thus the expressions being the arguments of the binary operators have type `Int`. Moreover, thanks to the subject reduction, we know that the closing configurations generated by the evaluation of those arguments are type safe, thus their values have type `Int` (which is the type of those expressions). And we know that, if $\mathit{ValType}(\mathit{val}) = \mathtt{Int}$, then $\mathit{val} \in \mathbb{Z}$ (see the definition of $\mathit{ValType}$ in Section 3), which concludes the proof.

9. Conclusions

In this paper, we tried to join the best of two worlds, big-step semantics and small-step semantics, by giving as primary semantics the big-step semantics, thus preserving readability and abstraction, and then extracting from it a notion of computation steps for programs, thus allowing the user to state and check properties otherwise naturally captured by the small-step operational semantics, as explained in Section 1.

The key is a partial algorithm to search for the derivation tree of the judgments of the big-step semantics, which mimics the expected process of evaluation of a program. Along with the derivation tree, it is easy to record all program intermediate steps that arise in the search process, thus yielding a trace of the computation of the program. The notion of intermediate steps (*program configurations*) of a program execution (*trace*) is then exploited to state and prove the type soundness property for a toy language LM, in an adequately strong version.

This paper is only a first step towards the formalization of the novel approach. Indeed, a meta-theory to formalize its strength and properties would be a desirable result and this is a research we intend to pursue. Informally, we can already claim that our approach cannot be applied to truly non-deterministic sets of big-step rules (like the ones present in some process algebras).

Acknowledgments. The authors would like to thank the anonymous referees. In particular, we would like to thank the referee who made suggestions on how to improve the intuitive presentation. We would also like to thank Kim Bruce, who was the external reviewer of the PhD thesis of the first author, Andrzej Tarlecki, who was the internal reviewer, and Pawel Urzyczyn, who was co-advisor together with the second author. Last but not least, the second author would like to thank Jerzy Tiuryn, to whom this work is dedicated, for scientific inspiration and help in establishing a long-lasting collaboration with the University of Warsaw.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., NJ, USA, 1996.

[2] M. S. Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 245–261. Springer-Verlag, 2005.

[3] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle$_{II}$. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

[4] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.

[5] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998.

[6] C. A. Gunter and D. Rémy. A Proof-Theoretic Assesment of Runtime Type Errors. AT&T Bell Laboratories Technical Memo 11261-921230-43TM, 1993.

[7] H. Ibraheem and D. A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and “higher-order” derivations. In *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*. Elsevier, 1997.

[8] G. Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[9] J. Kuśmierek. *A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming*. PhD thesis, Warsaw University, Departmanet of Informatics, 2010.

[10] X. Leroy and H. Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008.

[11] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[12] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus, 1981.

[13] A. Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. *Electr. Notes Theor. Comput. Sci.*, 10, 1997.

```
tr := list();

procedure searchI(j)
 $env, ctx, D \models instr \Rightarrow^{I}$ := j;
 tr.add( ($env$, $ctx$, $D$));     /* the opening configuration is added to the trace */
 case j of
  ....
 endcase;
 tr.add( (res, $ctx$, $D$)); /* the closing configuration is added to the trace */
 return res;
endprocedure

procedure searchE(j)
 $env, ctx, D \models exp \Rightarrow^{ex}$ := j;
 tr.add( ($env$, $ctx$, $D$));
 case j of
  ....
 endcase;
 tr.add( ($env$, $ctx$, res, $exp$, $D$) );
 return res;
endprocedure

procedure searchP(j)
 $\models \overrightarrow{F}; exp \Rightarrow^{P}$ := j;
 tr.add($\emptyset$);
 case j of
  ....
 endcase;
 tr.add( (res, $exp$, $D$) );
 return res;
endprocedure
```

Figure 9. Program trace construction using the derivation-search algorithm

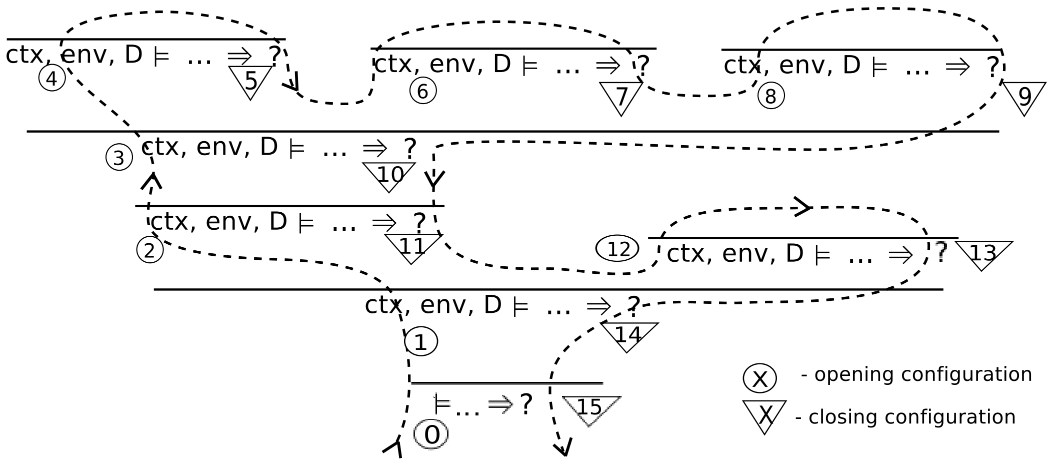

Figure 10. An example of a program trace