DiSIT, Computer Science Institute
Università del Piemonte Orientale “A. Avogadro”
Viale Teresa Michel 11, 15121 Alessandria
http://www.di.unipmn.it

A Provenly Correct Compilation of Functional Languages into Scripting Languages

P. Giannini, A. Shaqiri (paola.giannini@mfn.unipmn.it, albert.shaqiri@di.unimi.it)

TECHNICAL REPORT TR-INF-2014-12-02-UNIPMN
(December 2014)

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2014-01 *An Intelligent Swarm of Markovian Agents*, A. Bobbio, B. Dario , C. Davide, M. Gribaudo, S. Marco, June 2014.

2013-01 *Minimum pattern length for short spaced seeds based on linear rulers (revised)*, L. Egidi, G. Manzini, July 2013.

2012-04 *An intensional approach for periodic data in relational databases*, A. Bottrighi, A. Sattar, B. Stantic, P. Terenziani, December 2012.

2012-03 *Minimum pattern length for short spaced seeds based on linear rulers*, L. Egidi, G. Manzini, April 2012.

2012-02 *Exploiting VM Migration for the Automated Power and Performance Management of Green Cloud Computing Systems*, C. Anglano, M. Canonico, M. Guazzone, April 2012.

2012-01 *Trace retrieval and clustering for business process monitoring*, G. Leonardi, S. Montani, March 2012.

2011-04 *Achieving completeness in bounded model checking of action theories in ASP*, L. Giordano, A. Martelli, D. Theseider Dupré, December 2011.

2011-03 *SAN models of a benchmark on dynamic reliability*, D. Codetta Raiteri, December 2011.

2011-02 *A new symbolic approach for network reliability analysis* , M. Beccuti, S. Donatelli, G. Franceschinis, R. Terruggia, June 2011.

2011-01 *Spaced Seeds Design Using Perfect Rulers*, L. Egidi, G. Manzini, June 2011.

2010-04 *ARPHA: an FDIR architecture for Autonomous Spacecrafts based on Dynamic Probabilistic Graphical Models*, D. Codetta Raiteri, L. Portinale, December 2010.

2010-03 *ICCBR 2010 Workshop Proceedings*, C. Marling, June 2010.

2010-02 *Verifying Business Process Compliance by Reasoning about Actions*, D. D'Aprile, L. Giordano, V. Gliozzi, A. Martelli, G. Pozzato, D. Theseider Dupré, May 2010.

2010-01 *A Case-based Approach to Business Process Monitoring*, G. Leonardi, S. Montani, March 2010.

2009-09 *Supporting Human Interaction and Human Resources Coordination in Distributed Clinical Guidelines*, A. Bottrighi, G. Molino, S. Montani, P. Terenziani, M. Torchio, December 2009.

2009-08 *Simulating the communication of commands and signals in a distribution grid*, D. Codetta Raiteri, R. Nai, December 2009.

A Provenly Correct Compilation of Functional Languages into Scripting Languages

Paola Giannini[a,1,*], Albert Shaqiri[b]

[a]Computer Science Institute, DiSIT, Università del Piemonte Orientale, viale Teresa Michel 11, 15121 Alessandria, Italy
[b]Dipartimento di Informatica, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy

Abstract

In this paper we consider the problem of translating core `F#`, a typed functional language including mutable variables, into scripting languages such as JavaScript or Python. In previous work, we abstracted the most significant characteristics of scripting languages in an intermediate language (`IL` for short), which is an imperative language with definition of names (variables and functions) done in blocks and where a definition of a name does not have to statically precede its use. We define a big-step operational semantics for core `F#` and for `IL` and formalise the translation of `F#` expressions into `IL`. The main contribution of the paper is the proof of correctness of the given translation, which is done by showing that the evaluation of a well-typed `F#` program converges to a primitive value if and only if the evaluation of its translation into `IL` converges to the same value. For this proof is crucial the type soundness of core `F#` which is proved by giving a coinductive formalization of the divergence predicate and proving that well-typed expressions either converge to a value or diverge and so they are never stuck.

Keywords: scripting languages, functional languages, intermediate language, translation

1. Introduction

Programming in JavaScript (or any other dynamically typed language) optimizes the programming time, but can cause problems when big applications are created. The absence of type checking, may cause unexpected application behaviour followed by onerous debugging, and introduce serious difficulties in the maintenance of medium to large applications. For this reason dynamically typed languages are used mostly for prototyping and quick scripting.

*Principal corresponding author
Email addresses: `giannini@di.unipmn.it` (Paola Giannini), `albert.shaqiri@di.unimi.it` (Albert Shaqiri)
[1]Partly funded by "Progetto MIUR PRIN CINA Prot. 2010LHT4KM".

To deal with these problems, in previous work, [7] and [9], we proposed using dynamically typed languages as "assembly languages" to which we translate the source code from `F#` which is statically typed. In this way, we take advantage of the `F#` type checker and type inference system, as well as other `F#` constructs and paradigms such as pattern matching, classes, discriminated unions, namespaces, etc., and we may use the safe imperative features introduced via `F#` mutable variables. There are also the advantages of using an IDE such as Microsoft Visual Studio (code organization, debugging tools, IntelliSense, etc.).

To provide translation to different target languages we introduced an intermediate language, `IL` for short. This is useful, for instance, for translating to Python that does not have complete support for functions as first class concept, or for translating to JavaScript, using or not libraries such as jQuery.

In this paper we prove the correctness of the compilers produced. To do that we formalize the dynamic semantics of the languages `F#`and `IL`, the type-checking for `F#` and give a formal definition of the translation from the source language `F#` to `IL`. Finally, we prove that the translation preserves the dynamic semantics of `F#` expressions. The language `IL` is imperative and untyped, and has some of the characteristics of the scripting languages that makes them flexible, but difficult to check, such as blocks in which definition and use of variables may be interleaved, and in which use of a variable may precede its definition. (`IL` is partly inspired by IntegerPython, see [17].) Therefore, the proof of correctness of the translation from the source language `F#` to `IL` already covers most of the gap from functional to scripting languages.

In order to facilitate the proof of correctness, instead of the small-step semantics we introduced in [7], we give a big-step semantics to both languages. Then we define an equivalence between values and an equivalence between runtime configurations and show that equivalent configurations produce equivalent values and also that divergence is preserved. Since the big-step semantics of `F#` does not distinguish between non-terminating computations and computations that "go wrong" we give a coinductive characterization of divergent computations, and show that well-typed `F#` expressions either converge to a value or diverge.

The paper is organized as follows: in Section 2 we formalize the fragment of `F#` which is our source language and state its main properties. In Section 3 we introduce `IL` by first highlighting some of the design choices made for the language and then presenting its syntax and operational semantics. In Section 4 we briefly recall some of the challenges of the translation, widely discussed in [7] and [9], then we formalize the translation giving the translation of `F#` expressions to `IL` constructs. Finally, we prove that an `F#` program converges to a primitive value if and only if its `IL` translation converges to the same value. In the two appendices we give the proof of the lemmas of type preservation and progress for `F#` expressions, stated in Section 2, which are the results from which the soundness of the type system for `F#` is proved.

This paper is an extended and completely revised version of [7] and [9], whose aim was to introduce `IL` and show the challenges of the translation. As already mentioned, in this paper we give a different definition of the operational semantics of both languages and formalize and prove the correctness of the translation from the source language `F#` to the target language `IL`. We also formalize and prove the soundness of the type system for the dynamic semantics of `F#`.

2. Core `F#`

The syntax for core `F#` language is presented in Fig. 1. We included constructs, such as `let`, `let mutable`, and `let rec` that are used in the practice of programming and that raise challenges in the translation to dynamic languages. We also did not introduce imperative features through reference types, but through mutable variables, as this is closer to the imperative style of programming. We present a simply typed version of `F#`. However, our translator does not depend on the presence of types, since it uses `F#` type inference.

e	$::=$	$x \mid n \mid \texttt{tr} \mid \texttt{fls} \mid F \mid e{+}e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ e \mid e, e$	
		$\mid \texttt{let}\ [\texttt{mutable}]\ x{:}T{=}e\ \texttt{in}\ e \mid \texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e \mid x{<\!-}e$	expression
F	$::=$	$\texttt{fun}\ x{:}T{-\!>}e$	function
T	$::=$	$\texttt{int} \mid \texttt{bool} \mid T \rightarrow T$	type

Figure 1: Syntax of core `F#`

In the grammar for expressions, in Fig. 1, the square brackets "$[\dots]$" delimit an optional part of the syntax, we use x, y, z for variable names, and the overbar sequence notation is used according to [10]. For instance: "$\overline{x}{:}\overline{T}{=}\overline{F}$" stands for "$x_1{:}T_1{=}F_1 \cdots x_n{:}T_n{=}F_n$". The empty sequence is denoted by "$\emptyset$". For an `F#` expression e the *free variables of* e, $FV(e)$, are defined in Fig. 2. Note that, in the `let rec` construct, the occurrences of variables in $\overline{w}$ in $\overline{F}$ are bound. An expression e *is closed* if $FV(e) = \emptyset$. We assume equality of expressions up to α-equivalence. With $e[x := e']$ we denote the result of *substituting x with e' in e* with renaming of bound variables if needed.

- $FV(x) = \{x\}$,
- $FV(n) = FV(\texttt{tr}) = FV(\texttt{fls}) = \emptyset$,
- $FV(\texttt{fun}\ x{:}T{-\!>}\{e\}) = FV(e) - \{x\}$,
- $FV(e_1{+}e_2) = FV(e_1\ e_2) = FV(e_1, e_2) = FV(e_1) \cup FV(e_2)$,
- $FV(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) = FV(e_1) \cup FV(e_2) \cup FV(e_3)$,
- $FV(\texttt{let}\ [\texttt{mutable}]\ x{:}T{=}e_1\ \texttt{in}\ e_2) = FV(e_1) \cup (FV(e_2)) - \{x\})$,
- $FV(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e) = (FV(e) \bigcup_{1 \le i \le m} FV(F_i)) - \{\overline{w}\}$,
- $FV(x{<\!-}e) = FV(e) \cup \{x\}$.

Figure 2: Free variables of expressions

The `let rec` construct introduces mutually recursive functions. The `let` construct (followed by an optional `mutable` modifier) binds the variable x to the value resulting from the evaluation of the expression on the right-hand-side of $=$ in the evaluation of the body of the construct. In the (concrete syntax) of the examples, as in `F#`, "," and "`in`" are substituted by "carriage return" without indentation.

When the `let` construct is followed by `mutable` the variable introduced is mutable. Only mutable variables may be used on the left-hand-side of an assignment. This restriction is enforced by the type system of the language. The type system enforces also the restriction that the body of a function cannot contain free mutable variables, even though it may contain bound mutable variables. We did not model the fact that top level mutable variables could be refereed in functions, however, our compiler, handles such variables.

We present a simply and fully typed version of `F#`. Function parameters as well as variables defined in the `let`, `let mutable`, and `let rec` constructs are annotated with their type. This is just to simplify the proof of the correctness of the translation. As already mentioned, our translator takes as input the standard polymorphically typed `F#`.

A type environment Γ is defined by:

$$\Gamma ::= x{:}T, \Gamma \mid x{:}T!, \Gamma \mid \emptyset$$

that is Γ associates variables with types, possibly followed by !. If the type is followed by ! this means that the variable was introduced with the `mutable` modifier. Let $\dagger$ denote either ! or the empty string, and let $dom(\Gamma) = \{x \mid x{:}T \dagger \in \Gamma\}$. We assume that for any variable x, in Γ there is at most an associated type. We say that the *expression* e *has type* T *in the environment* Γ if the judgment

$$\Gamma \vdash e : T$$

is derivable from the rules of Fig. 3. In the rules of Fig. 3, with $\Gamma[\Gamma']$ we denote the type environment such that $dom(\Gamma[\Gamma']) = dom(\Gamma) \cup dom(\Gamma')$ and:

- if $x{:}\tau \dagger \in \Gamma'$ then $x{:}\tau \dagger \in \Gamma[\Gamma']$, and
- if $x{:}\tau \dagger \in \Gamma$ and $x \notin dom(\Gamma')$, then $x{:}\tau \dagger \in \Gamma[\Gamma']$.

In the following we describe the most interesting rules.
Consider rule (TyAbs), to type the body of a function we need assumptions on its free variables and formal parameter. From the definition of $\Gamma[\Gamma']$ we have that, in the environment in which the function is defined, $\Gamma[\Gamma']$, there can be mutable variables, as long as they are not needed to type the body of the function. Moreover, the body of the function could contain bound mutable variables.
In the rule (TyLet), in typing e_2 the variable x is bound to the type of the expression e_1, and in the rule (TyLetMut) in typing e_2 the variable y is bound to the type of the expression e_1 followed by !, so that inside e_2 the variable y may be used on the left-hand-side of an assignment (see rule (TyAssign)).
Finally in rule (TyRec), the variables $\overline{w}$ are bound to the types $\overline{T}$, both in the typing of the body e of the construct, and also in the typing of their definitions $\overline{F}$. Moreover, the type of the function definition, F_k, associated with w_k must be T_k $(1 \leq k \leq m)$.

In this paper, we give a big-step semantics for the language with an explicitly *typed evaluation stack* for keeping the bindings of the immutable variables. The definition of values and stacks is given in Fig. 4. Function values contain their definition stack, i.e., are *closures*. For recursive functions the definition stack of the functions F_i should include the definition F_i itself. To break circularity we use the `let rec` construct for the expression part of the value of mutually recursive functions.

$$\frac{x{:}T\,\dagger \in \Gamma}{\Gamma \vdash x : T}\;\text{(TyVar)} \qquad \Gamma \vdash n : \texttt{int}\;\text{(TyNum)} \qquad \Gamma \vdash \texttt{tr}, \texttt{fls} : \texttt{bool}\;\text{(TyBool)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash e_1 {+} e_2 : \texttt{int}}\;\text{(TySum)} \qquad \frac{\begin{array}{c}\Gamma'[x{:}T'] \vdash e : T \\ \forall y,\, T''\; y{:}T''! \notin \Gamma'\end{array}}{\Gamma[\Gamma'] \vdash \texttt{fun}\; x{:}T'\texttt{->}e : T' \to T}\;\text{(TyAbs)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \;\; \Gamma \vdash e_1 : T \;\; \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\; e\; \texttt{then}\; e_1\; \texttt{else}\; e_2 : T}\;\text{(TyIf)} \qquad \frac{\Gamma \vdash e_1 : T' \to T \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1\; e_2 : T}\;\text{(TyApp)}$$

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1, e_2 : T}\;\text{(TySeq)} \qquad \frac{\Gamma \vdash e_1 : T' \quad \Gamma[x{:}T'] \vdash e : T}{\Gamma \vdash \texttt{let}\; x{:}T'{=}e_1\; \texttt{in}\; e_2 : T}\;\text{(TyLet)}$$

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma[y{:}T'!] \vdash e : T}{\Gamma \vdash \texttt{let mutable}\; y{:}T{=}e_1\; \texttt{in}\; e_2 : T}\;\text{(TyMut)} \qquad \frac{\Gamma[\overline{w}{:}\overline{T}] \vdash \overline{F} : \overline{T} \;\; \Gamma[\overline{w}{:}\overline{T}] \vdash e : T}{\Gamma \vdash \texttt{let rec}\; \overline{w}{:}\overline{T}{=}\overline{F}\; \texttt{in}\; e : T}\;\text{(TyRec)}$$

$$\frac{\Gamma \vdash e : T \quad x{:}T! \in \Gamma}{\Gamma \vdash x \texttt{<-} e : T}\;\text{(TyAssign)}$$

Figure 3: Typing rules for core `F#`

v	$::=$	$n \mid \texttt{tr} \mid \texttt{fls} \mid (\texttt{fun}\; x{:}T\texttt{->}e, \sigma) \mid (\texttt{let rec}\; \overline{w}{:}\overline{T}{=}\overline{F}\; \texttt{in}\; F_i, \sigma)$	value
σ	$::=$	$x{:}T{\mapsto}v, \sigma \mid \emptyset$	stack

Figure 4: `F#` values

The domain of a stack $dom(\sigma)$ is the set of variables x such that $x{:}T{\mapsto}v \in \sigma$. For stacks we use the override notation $\sigma[\sigma']$ as for type environment.

The set of free variables of a value $v = (e, \overline{x}{:}\overline{T}{\mapsto}\overline{v})$ is defined by: $FV(v) = (FV(e) - \{\overline{x}\}) \cup FV(\overline{v})$.

The standard α-conversion of functional languages, i.e.,

$$\texttt{fun}\; x{:}T\texttt{->}e =_\alpha \texttt{fun}\; z{:}T\texttt{->}e[x := z] \quad z \notin FV(e)$$

is extended to values with the transitive closure of the following rule

$$(e, \overline{x}{:}\overline{T}{\mapsto}\overline{v}) =_\alpha (e[x_i := z], [x_1 \cdots x_{i-1}\, z\, x_{i+1} \cdots x_n{:}\overline{T}{\mapsto}\overline{v}]) \quad z \notin FV(e, \sigma) \cup \{\overline{x}\}$$

Equality of values will be considered up to α-conversion.

The lookup function that follows, given a variable and a stack, returns the value associated to the variable in the stack, if any. It handles recursive function values, by returning their body and a stack in which the name of the mutually recursive function are bound to their recursive definition. That is, unrolling the `let rec` construct once.

Definition 1. *The* lookup function*, $lkp(\sigma)$, is defined by:*

- *$lkp(x, \sigma[x{:}T{\mapsto}v]) = (F_k, \sigma[w_i{:}T_i{\mapsto}(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)]_{1\leq i\leq m})$, if $v = (\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$*
- *$lkp(x, \sigma[x{:}T{\mapsto}v]) = v$, if v is not a recursive function,*
- *$lkp(x, \sigma[x'{:}T{\mapsto}v]) = lkp(x, \sigma)$ if $x \neq x'$.*

The *type environment associated with σ, $env(\sigma)$*, is defined by:

$$env(\emptyset) = \emptyset \qquad env(\sigma[x{:}T{\mapsto}v]) = env(\sigma)[x{:}T]$$

Our core `F#` language has imperative features, so for the definition of the operational semantics we need a *store* which is a mapping between locations and values:

$$l_1 \mapsto v_1, \ldots, l_n \mapsto v_n$$

With $\rho[x \mapsto v]$ we denote the mapping defined by: $\rho[x \mapsto v](x) = v$ and $\rho[x \mapsto v](y) = \rho(y)$ when $x \neq y$.

The *runtime configurations* are triples "runtime expression, stack, store", $\langle e \mid \sigma \mid \rho\rangle$, where the *runtime expressions*, including locations (generated by the evaluation of mutable variables definitions) are defined by adding the clauses:

$$e ::= \cdots \mid l \mid l{<}{-}e$$

to the grammar of expression of Fig. 1. In Fig. 5 we give the *rules for the evaluation relation, $\Downarrow$,* that given a runtime configuration produces a pair "value, store", which is the result of the evaluation of the expression. In particular, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$ means that the evaluation of e with the stack σ, in the store ρ produces the value v, and modifies the store to be ρ'.

Evaluation of a variable produces the value returned by the lookup function applied to the evaluation stack, rule (VAR-F). Evaluation of an integer or a boolean value produces the value itself, rule (PR-VAL-F), whereas to produce a function value from a function expression, in rule (FN-VAL-F), we associate with the function its definition stack. (For recursive functions this is done with rule (LETREC-F).) Evaluation of a location, rule (LOC-F), produces the value associated to the location in the store. The evaluation of the sum expression evaluates the operands of the expression, and then returns n, which is the numeral corresponding to the sum of the values of such operands n_1 and n_2. For the conditional expression we first evaluate the condition and then return the evaluation of the `then` or `else` branch depending on the (boolean) value of the condition. For an application, rule (APP-F), we first evaluate the expression on the left, which result must be a function value, then we evaluate the actual parameter, and then return the result of the evaluation of the function body. The evaluation stack for the body is the definition stack of the function on which we add the association between the formal parameter x and the value of the actual parameter. Similarly for (LET-F), where the definition stack of the expression is the current evaluation stack. Instead, for a mutable variable, rule (LETMUT-F) , a new location l is generated, added to the store with the initial value given by the evaluation of the expression associated with y, and the occurrences of y in the body of the construct are substituted with such location. Between these occurrences there are the variables on the left-hand-side of assignments. Indeed, since

$$\frac{\mathit{lkp}(x,\sigma) = v}{\langle x \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle}\ (\textsc{Var-F}) \qquad \frac{e = n \lor e = \mathtt{tr} \lor e = \mathtt{fls}}{\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle e \mid \rho\rangle}\ (\textsc{Pr-Val-F})$$

$$\frac{}{\langle F \mid \sigma \mid \rho\rangle \Downarrow \langle (F,\sigma) \mid \rho\rangle}\ (\textsc{Fn-Val-F}) \qquad \frac{\rho(l) = v}{\langle l \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle}\ (\textsc{Loc-F})$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle n_1 \mid \rho_1\rangle \quad \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle n_2 \mid \rho_2\rangle \quad \tilde{n} = \tilde{n}_1 +^{\mathtt{int}} \tilde{n}_2}{\langle e_1{+}e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle n \mid \rho_2\rangle}\ (\textsc{Sum-F})$$

$$\frac{\begin{array}{c}\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle \land \\ (\,(\, v = \mathtt{tr} \land \langle e_1 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle\,) \lor (\, v = \mathtt{fls} \land \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle\,)\,)\end{array}}{\langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_2\rangle}\ (\textsc{If-F})$$

$$\frac{\begin{array}{c}\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\mathtt{fun}\ x{:}T\texttt{->}e, \sigma') \mid \rho_1\rangle \quad \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle \\ \langle e \mid \sigma'[x{:}T{\mapsto}v] \mid \rho_2\rangle \Downarrow \langle v' \mid \rho_3\rangle\end{array}}{\langle e_1\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v' \mid \rho_3\rangle}\ (\textsc{App-F})$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle \quad \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_2 \mid \rho_2\rangle}{\langle e_1, e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v_2 \mid \rho_2\rangle}\ (\textsc{Seq-F})$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle \quad \langle e_2 \mid \sigma[x{:}T{\mapsto}v] \mid \rho_1\rangle \Downarrow \langle v' \mid \rho_2\rangle}{\langle \mathtt{let}\ x{:}T{=}e_1\ \mathtt{in}\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v' \mid \rho_2\rangle}\ (\textsc{Let-F})$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle \quad \langle e_2[y := l] \mid \sigma \mid \rho_1[l \mapsto v]\rangle \Downarrow \langle v' \mid \rho_2\rangle \quad l \notin \mathit{dom}(\rho_1)}{\langle \mathtt{let\ mutable}\ y{:}T{=}e_1\ \mathtt{in}\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v' \mid \rho_2\rangle}\ (\textsc{LetMut-F})$$

$$\frac{\langle e \mid \sigma[w_i{:}T_i{\mapsto}(\mathtt{let\ rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \mathtt{in}\ F_i, \sigma)]_{1 \le i \le n} \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle}{\langle \mathtt{let\ rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \mathtt{in}\ e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle}\ (\textsc{LetRec-F})$$

$$\frac{\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle}{\langle l \texttt{<-} e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1[l \mapsto v]\rangle}\ (\textsc{Ass-F})$$

Figure 5: Big-step operational semantics for core `F#`

in well-typed expressions variables on the left-hand-side of assignments were always introduced by `let mutable`, when an assignment is evaluated, rule (Assign-F), we have a configuration: $\langle l \texttt{<-} e \mid \rho\rangle$ which is evaluated by changing the value of the location l to be result of the evaluation of e. The evaluation of `let rec`, rule (Letrec-F), produces the result of the evaluation of the body e with an evaluation stack, σ', which is the current stack, σ, to which we add the associations between the names of the recursively defined functions, w_i, and the function values, $(\mathtt{let\ rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \mathtt{in}\ F_i, \sigma)$, so that the evaluation of an occurrence of w_i in e will produce the evaluation of F_i with the stack

σ', as it should be.

The typing rules in Fig. 3 are for the (source) expression language, so they do not include a rule for locations. To type runtime expressions we need a store environment Σ assigning types to locations. The type judgment is:

$$\Gamma \mid \Sigma \vdash e : T$$

and the typing rule for locations and assignment to a location are:

$$\Gamma \mid \Sigma \vdash l : \Sigma(l) \quad \text{(TyLocF)} \qquad \frac{\Gamma \mid \Sigma \vdash e : T \quad \Sigma(l) = T}{\Gamma \mid \Sigma \vdash l \texttt{<-} e : T} \text{(TyAssignLoc)}$$

All the other rules are obtained by putting $\Gamma \mid \Sigma$ on the left-hand-side of "$\vdash$" in the typing rules of Fig. 3, except for rule (TyAbs) which becomes:

$$\frac{\Gamma'[x{:}T] \mid \emptyset \vdash e : T' \qquad \forall y, T''\ y{:}T''! \notin \Gamma'}{\Gamma[\Gamma'] \mid \Sigma \vdash \texttt{fun}\ x{:}T \texttt{->} e : T \to T'} \text{(TyAbs)}$$

In the following we define well-typed configurations, and prove the soundness result, which is derived from a big-step semantics version of the Subject Reduction and Progress lemmas that follow. The soundness of the type system is essential for the proof of correctness of our translation.

In order to define well-typed runtime configurations we have to define well-typed values, stacks and stores. The definition of well-typed values and stacks are mutually recursive, however, there is no circularity, since stacks contain types, and are subterm of value function terms.

Definition 2. 1. *An* `F#` *value* v has type T, $\models v{:}T$,
   (a) *if* $v = n$, *then* $T = \texttt{int}$
   (b) *if* $v = \texttt{tr}$ *or* $v = \texttt{fls}$, *then* $T = \texttt{bool}$
   (c) *if* $v = (\texttt{fun}\ x{:}T_1 \texttt{->} e, \sigma)$, *then for some* T_2, *we have* $T = T_1 \to T_2$, $env(\sigma) \mid \emptyset \vdash \texttt{fun}\ x{:}T_1 \texttt{->} e : T_1 \to T_2$ *and* $\models \sigma\diamond$
   (d) *if* $v = (\texttt{let rec}\ \overline{x}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)$, *then* $T = T_i$, $\models \sigma\diamond$ *and for all* j, $1 \leq j \leq m$, $env(\sigma)[\overline{x}{:}\overline{T}] \mid \emptyset \vdash F_j : T_j$.
2. *A stack* σ is well-typed, $\models \sigma\diamond$, *if for all* $x{:}T {\mapsto} v \in \sigma$ *we have that* $\models v{:}T$.
3. *An* `F#` *store* ρ *is* well-typed with respect to a store environment Σ, $\Sigma \models \rho$, *if* $dom(\rho) = dom(\Sigma)$ *and for all* $l \in \rho$ *we have that*
   (a) $\models \rho(l){:}\Sigma(l)$ *and*
   (b) *if* $\Sigma(l) = T_1 \to T_2$ *for some* T_1 *and* T_2, *then* $\rho(l) = (\texttt{fun}\ x{:}T_1 \texttt{->} e', \sigma')$ *for some* e' *and* σ'.
4. *The* `F#` *configuration* $\langle e \mid \sigma \mid \rho \rangle$ is well-typed, with respect to Σ, $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$, *if*
   (a) $env(\sigma) \mid \Sigma \vdash e : T$ *for some* T
   (b) $\models \sigma\diamond$ *and*
   (c) $\Sigma \models \rho$.

Evaluation of well-typed configurations preserves well-typed stores, and the type of expressions.

Lemma 3 (Type Preservation). *Let $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$, then $\models v{:}T$ where $env(\sigma) \mid \Sigma \vdash e : T$, and $\Sigma' \models \rho'$ for some $\Sigma' \supseteq \Sigma$. Moreover, if $T = T_1 \to T_2$, then $v = (\texttt{fun}\ x{:}T_1\texttt{->}e', \sigma')$ for some e' and σ'.*

Proof:
The proof of the lemma is given in AppendixA. $\square$

The big-step semantics is convenient for the proof of correctness of the translation, however, it has the disadvantage of not distinguishing between non terminating programs and programs that "get stuck". In our previous papers, [7] and [9], we had defined a small-step semantics which insures that well-typed programs do not "get stuck". Here, following [12], we give a coinductive characterization of the *divergence judgment* $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ and show that well-typed configurations either diverge or converge to a value. This insures that programs do not "get stuck".

The divergence judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ is defined in Fig. 6.

The rules are quite obvious. The evaluation of a sum expression, rule (Sum-$\Uparrow$), diverges when either the evaluation of the first operand of the expression diverges, or when this evaluation converges to a numeral and the evaluation of the second operand diverges. The evaluation of a conditional expression, rule (If-$\Uparrow$), diverges when either the evaluation of the condition diverges, or if this evaluation converges to either `tr` or `fls` and the evaluation of the corresponding branch diverges. For an application, rule (App-$\Uparrow$), the evaluation diverges if either the evaluation of the expression on the left diverges, or if this evaluation converges to a function value and the evaluation of the actual parameter diverges, or if also the evaluation of the actual parameter produces a value, then the evaluation of the function body with the definition stack of the function on which we add the association between the formal parameter x and the value of the actual parameter diverges. Evaluation of a sequence of expressions, rule (Seq-$\Uparrow$), diverges if either the evaluation of the first expression diverges of if this converges to any value and the evaluation of the second expression diverges. A `let` or `let mutable` expressions diverges if either the evaluation of the expression associated with the defined variable diverges or if this evaluation converges and the evaluation of the body of the construct diverges. The evaluation of a `let rec` expression, rule (Letrec-$\Uparrow$), diverges if it diverges the evaluation of its body with an evaluation stack, σ', which is the current stack, σ, on which we add the associations between the names of the recursively defined functions, w_i, and the function values, $(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)$. The evaluation of an assignment, rule (Ass-$\Uparrow$), diverges if it diverges the evaluation of its right-hand-side.

The following lemma expresses the progress property for big step semantics, which says that well-typed configurations do not get stuck. Note that, the "either or" has to be interpreted as an "exclusive or", as the lemma states. It is important to prove this, since the relations of convergence to a value and the one of divergence are specified by two separate sets of rules that could be both applicable to a given configuration.

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow \quad \vee \quad (\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle n_1 \mid \rho_1\rangle \quad \wedge \quad \langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow\ )}{\langle e_1{+}e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Sum-}\Uparrow)$$

$$\frac{\begin{array}{c}\langle e \mid \sigma \mid \rho\rangle\Uparrow \quad \vee \quad (\ (\ \langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathtt{tr} \mid \rho_1\rangle \ \wedge\ \langle e_1 \mid \sigma \mid \rho_1\rangle\Uparrow\ ) \quad \vee \\ (\ \langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathtt{fls} \mid \rho_1\rangle \ \wedge\ \langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow\ )\ )\end{array}}{\langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{If-}\Uparrow)$$

$$\frac{\begin{array}{c}\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow \ \vee\ (\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\mathtt{fun}\ x{:}T\mathtt{->}e, \sigma') \mid \rho_1\rangle \ \wedge\ \langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow\ ) \vee \\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\mathtt{fun}\ x{:}T\mathtt{->}e, \sigma') \mid \rho_1\rangle \wedge \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle \wedge \langle e \mid \sigma'[x{\mapsto}v] \mid \rho_2\rangle\Uparrow\end{array}}{\langle e_1\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{App-}\Uparrow)$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow \quad \vee \quad (\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle \quad \wedge \quad \langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow\ )}{\langle e_1, e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Seq-}\Uparrow)$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow \ \vee\ (\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle \wedge \langle e_2 \mid \sigma[x{:}T{\mapsto}v] \mid \rho_1\rangle\Uparrow\ )}{\langle \mathtt{let}\ x{:}T{=}e_1\ \mathtt{in}\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Let-}\Uparrow)$$

$$\frac{\begin{array}{c}\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow \quad \vee \\ \langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle \quad \wedge \quad \langle e_2[y := l] \mid \sigma \mid \rho_1[l \mapsto v]\rangle\Uparrow \quad l \notin dom(\rho_1)\end{array}}{\langle \mathtt{let\ mutable}\ y{:}T{=}e_1\ \mathtt{in}\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{LetMut-}\Uparrow)$$

$$\frac{\langle e \mid \sigma[w_i{:}T_i{\mapsto}(\mathtt{let\ rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \mathtt{in}\ F_i, \sigma)]_{1\leq i\leq n} \mid \rho\rangle\Uparrow}{\langle \mathtt{let\ rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \mathtt{in}\ e \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{LetRec-}\Uparrow)$$

$$\frac{\langle e \mid \sigma \mid \rho\rangle\Uparrow}{\langle l\mathtt{<-}e \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Ass-}\Uparrow)$$

Figure 6: Coinductive characterization of divergence

Lemma 4 (Progress). *Let $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. Either $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$ for some value v and store ρ', or $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. Moreover, it is not the case that $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$ for some value v and store ρ' and $\langle e \mid \sigma \mid \rho\rangle\Uparrow$.*

Proof:
The proof of the lemma is given in AppendixB. $\square$

Soundness of the type system for the big step semantics is expressed by the following theorem. Also in this theorem the "either or" has to be interpreted as an "exclusive or".

Theorem 5 (Soundness). *Let $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ and $env(\sigma) \mid \Sigma \vdash e : T$. Either $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$, where $\models v{:}T$ and $\Sigma' \models \rho'$ for some $\Sigma' \supseteq \Sigma$, or $\langle e \mid \sigma \mid \rho\rangle\Uparrow$.*

Proof:
Let $\langle e \mid \sigma \mid \rho \rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho \rangle \diamond$ and $env(\sigma) \mid \Sigma \vdash e : T$. From Lemma 4, either $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho' \rangle$ for some value v and store ρ', or $\langle e \mid \sigma \mid \rho \rangle \Uparrow$. In case $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho' \rangle$ for some value v and store ρ', from Lemma 3, we have that $\models v{:}T$ and $\Sigma' \models \rho'$ for some $\Sigma' \supseteq \Sigma$. $\square$

Let $Loc(e)$ be the set of locations occurring in the expression e. An F# *program* is a well-typed closed expression e such that $Loc(e) = \emptyset$. The *initial configuration* associated with a program is $\langle e \mid \emptyset \mid [\,] \rangle$. From Theorem 5, either $\langle e \mid \emptyset \mid [\,] \rangle \Downarrow \langle v \mid \rho \rangle$, where v has the same type of e, and $\Sigma \models \rho$ for some Σ or $\langle e \mid \emptyset \mid [\,] \rangle \Uparrow$.

3. Intermediate language

3.1. Design Choices

In designing the intermediate language, IL, our goals where, on one side to have a language close enough to the structure of the dynamic languages target of the translation, namely JavaScript and Python and on the other to allow us to give a simple enough translation, which could be formally proved to preserve the semantics of the original functional language, F#.

Our IL is an imperative language with three syntactic categories: expressions, statements and blocks. We introduce the distinction between expressions and statements as many target languages do. This facilitates the translation process and prevents some errors while building the intermediate abstract syntax tree, see [3] for a similar choice. The block structure is inspired by IntegerPython, see [17]. Variables are statically scoped, in the sense that, if there is a definition of the variable x in a block, all the free occurrences of x in the block refer to this definition. However, we can have occurrences of x preceding its definition. As in Python and JavaScript closures are expressions, but sequence of expressions are not expressions.

Many F# constructs can be directly mapped to JavaScript (or Python), but when this is not the case we obtain a semantically equivalent behavior by using the primitives offered by the target language. E.g., in F# a sequence of expressions is itself an expression, while in JavaScript and Python it is a statement.

Citing the example from [7], suppose we want to translate a piece of code that calculates a fibonacci number, binds the result to a name and also stores the information if the result is even or odd, see left-side of Fig. 7. The JavaScript translation can be easily accommodated by wrapping the sequence into a function, and calling the function, as the JavaScript program on the left-side of Fig. 8 shows (line $5 \div 11$). However, the same cannot be done in Python as its support for closures is partial. In particular, in Python we have to define a temporary function, say `temp`, in the global scope and to execute it we have to call `temp` in the place where the original sequence should be. However, variables such as `even` will be out of the scope of their definition and this would make the translation wrong. To obtain a behavior semantically equivalent, we have to pass to `temp` the variable `even`, by reference, since it may be modified in the body of `temp`. The resulting translation is show on the right-side of Fig. 8. Note that, this problem is not present in JavaScript where the closure is defined and called in the scope of `even`.

We abstract from this specific problem and consider the more general problem of moving "open code" from its context, replacing it with an expression having the same behavior and, taking inspiration from work on dynamic binding, see [15] and recent work by one of the author, we define a pair of boxing/unboxing constructs, that we call: `code`, and `exc`. The construct `code` wraps "open code" (in this case a sequence of expressions) providing the information on the environment needed for its execution, that is the mutable and immutable variables occurring in it. This construct defines a value, similar to a function closure. The construct `exc` is used to execute the code contained in `code`. To do this it must provide values for the immutable variables, in our example the variable z and bindings for the mutable variables to variables in the current environment, since when executing the code we have to modify the variable `even`.

With these constructs, the `F#` code on the left-side of Fig. 7 would be translated into the IL code on the right-side of Fig. 7.

```
let z=7
let mutable even = false
let x =
   let rec fib x =
      if x < 3 then 1
      else
         fib(x - 1) + fib(x - 2)
   let temp = fib z
   even <- (temp % 2 = 0)
   temp
x
```

```
{ def y = code(
    { def fib = fun x ->
          if x < 3 then 1
          else
           (fib(x-1)+fib(x-2));
      def temp = fib u;
      w := temp % 2 = 0;
      temp },
   w->EV,  u);
  def z = 7;
  def even = false;
  def x = exc(y, EV->even, z);
  x }
```

Figure 7: `F#` program and the corresponding `IL` translation

```
(function() {
  var z = 7;
  var even = false;
  var x = (function () {
    var fib = function (x) {
      if (x < 3) return 1;
      else
        return fib(x-1)+fib(x-2)};
    var temp = fib(z);
    even = (temp % 2) == 0;
    return temp })();
  return x })();
```

```
def temp1(w, z):
  def temp2(w, fib, x):
    if (x < 3):  return 1
    else: return fib(x-1)+fib(x-2)
  fib = lambda x: temp2(w, fib, x)
  temp = fib(z)
  w.value = ((temp % 2) == 0)
  return temp
def __main__():
  z = 7
  even = False
  wrapper1 = ByRef(even)
  x = temp1(wrapper1, z)
  even = wrapper1.value
  return x
__main__();
```

Figure 8: JavaScript and Python translations

3.2. Syntax and Semantics of `IL`

In the syntax of `IL`, given in Fig. 9, there are three (main) syntactic categories: *blocks*, *statements*, and *expressions*.

$$\begin{array}{lcll}
bl & ::= & \{se\} & \text{block} \\
se & ::= & st \mid e \mid se; se & \text{sequence} \\
st & ::= & x \texttt{<-} e \mid \texttt{def}\ x\texttt{=}e & \text{statement} \\
e & ::= & x \mid n \mid \texttt{tr} \mid \texttt{fls} \mid \texttt{fun}\ x\texttt{->}bl \mid e\texttt{+}e \mid \texttt{if}\ e\ \texttt{then}\ bl\ \texttt{else}\ bl \mid e\ e & \\
 & & \mid \texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \texttt{exc}(e, \overline{Y} \mapsto \overline{y}, \overline{e}) & \text{expression} \\
v & ::= & n \mid \texttt{tr} \mid \texttt{fls} \mid \texttt{fun}\ x\texttt{->}bl \mid \texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}) & \text{value}
\end{array}$$

Figure 9: Syntax of `F#`

Blocks are sequences of statements or expressions, enclosed in brackets. In our translation we flatten the nested structure of `let` constructs so we need blocks in which definitions and expressions/statements may be intermixed. Moreover, since we do not have a specific `let rec` construct use of a variable may precede its definition, e.g., when defining mutually recursive (or simply recursive) functions. Statements may be either assignments or variable definitions. Our compiler handles many more statements, but these are enough to show the ideas behind the design of `IL`. Forward definition in a block are permitted, E.g.,

$$\{\texttt{def f} = \texttt{fun y}\texttt{->}\ \{\texttt{x}\}; \texttt{def x} = \texttt{5}; \texttt{f 2}\}$$

correctly returns 5, whereas the following code would produce a runtime error:

$$\{\texttt{def x} = \texttt{7}; \texttt{if (x} > \texttt{3) then}\ \{\texttt{def f} = \texttt{fun y}\texttt{->}\ \{\texttt{x}\}; \texttt{f 2}; \texttt{def x} = \texttt{5}; \texttt{3}\}\ \texttt{else}\ \{\texttt{4}\}\}$$

since when `f` is called the variable `x`, defined in the inner block, has not yet been assigned a value. Instead, if `x` was not defined in the inner block, like in the following

$$\{\texttt{def x} = \texttt{7}; \texttt{if (x} > \texttt{3) then}\ \{\texttt{def f} = \texttt{fun y}\texttt{->}\ \{\texttt{x}\}; \texttt{f 2}\}\ \texttt{else}\ \{\texttt{4}\}\}$$

the block would return 7, since `x` is bound in the enclosing block. This is also the behavior in JavaScript and Python. The construct `code` is used to move a block, bl, outside its definition context. To produce a closed term, the *mutable variables* free in bl, $\overline{y}$, are unbound by associating them to *global names* $\overline{Y}$ not subject to renaming. The variables $\overline{x}$, instead, are the *immutable variables* free in bl, i.e., they are not modified by the execution of bl. The metavariables, X, Y, Z are used to denote names. Values are integers, booleans, functions (as for `F#`) and code.

In Fig. 10, we define the *free variables of an expression*, $FV(e)$, and the *free variables of a block*, $FV(bl)$. Since we may have forward definitions to define the free variable of a block we first define the variables occurring in a sequence of statements or expressions, se and the variable defined in se. Note that, assignment does not define a variable.

The operational semantics of `IL` is given by defining evaluation relations, for the syntactic categories of `IL`. In particular, we define $\Downarrow_{bl}$ for blocks, $\Downarrow_{sq}$ for sequences, $\Downarrow_{st}$ for statements and $\Downarrow_{ex}$ for expressions. Configurations will be pairs, "$\langle C \mid \rho\rangle$ ",which

- $FV(e)$, is defined by
  - $FV(x) = \{x\}$
  - $FV(n) = FV(\texttt{tr}) = FV(\texttt{fls}) = \emptyset$
  - $FV(\texttt{fun}\, x\texttt{->}\{bl\}) = FV(bl) - \{x\}$
  - $FV(e_1\texttt{+}e_2) = FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$
  - $FV(\texttt{if}\ e\ \texttt{then}\ \{bl_1\}\ \texttt{else}\ \{bl_2\}) = FV(bl_1) \cup FV(bl_2) \cup FV(e)$
  - $FV(\texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}))) = FV(\{se\}) - \{\overline{x}, \overline{y}\}$
  - $FV(\texttt{exc}(e, \overline{Y} \mapsto \overline{y}, \overline{e})) = FV(e) \cup \{\overline{y}\} \cup FV(\overline{e})$
- $Var(se)$, is defined by
  - $Var(x\texttt{<-}e) = FV(e) \cup \{x\}$
  - $Var(\texttt{def}\ x\texttt{=}e) = Var(e) = FV(e)$
  - $Var(se_1; se_2) = Var(se_1) \cup Var(se)$
- $def(se)$ is defined by
  - $def(x\texttt{<-}e) = def(e) = \emptyset$
  - $def(\texttt{def}\ x\texttt{=}e) = \{x\}$
  - $def(se_1; se_2) = def(se_1) \cup def(se_2)$
- $FV(\{se\}) = Var(se) - def(se)$

Figure 10: Free variables of expressions and blocks

first component is a syntactic construct: block, sequence, statement, or expression and the second a store. As for `F#` we have to add to the syntax of expressions locations, l, as they are generated during the evaluation of blocks.The syntax of the runtime language is generated by adding the the clauses for expressions and statements that follows.

$$\begin{array}{lll} st & ::= & \cdots \mid l\texttt{<-}e \mid \texttt{def}\ l\texttt{=}e \\ e & ::= & \cdots \mid l \mid \texttt{code}(bl, \overline{l} \mapsto \overline{Y}, \overline{x}) \mid \texttt{exc}(e, \overline{Y} \mapsto \overline{l}, \overline{e}) \end{array}$$

The rules of the operational semantics are defined in Fig. 11.

The first rule, (BLOCK), defines the evaluation of a block. We first allocate the locations for the variables defined in the block and then return the result of the evaluation of the sequence of statements or expressions of the block. The function $def(se)$ mapping a sequence to the set of variables defined in it is defined in Fig. 10. We want to model forward declarations in blocks and the fact that the evaluation of an access to a variables before it is assigned a value is not permitted. To this extent, the initial value of the locations is set to undefined, ?, so that an access to this location before the evaluation of an assignment or a definition for the corresponding variable would be stuck. Note that, *this will never happen for* `IL` *programs which are translation of* `F#` *programs*. After this initial allocation a closed block will not contain free variables. The evaluation for sequence of statements or expressions is obvious.
Rules (ASS) and (DEF) define the evaluation of a statement. They both modify the location on the left-hand-side to the value resulting from the evaluation of the expression

$$\frac{\langle se[\overline{x} := \overline{l}] \mid \rho[\overline{l} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v \mid \rho'\rangle \qquad \{\overline{x}\} = \mathit{def}(se) \qquad \{\overline{l}\} \cap \mathit{dom}(\rho) = \emptyset}{\langle \{se\} \mid \rho\rangle \Downarrow_{bl} \langle v \mid \rho'\rangle}\ \text{(Block)}$$

$$\frac{\langle se_1 \mid \rho\rangle \Downarrow_{sq} \langle v_1 \mid \rho_1\rangle \qquad \langle se_2 \mid \rho_1\rangle \Downarrow_{sq} \langle v_2 \mid \rho_2\rangle}{\langle se_1; se_2 \mid \rho\rangle \Downarrow_{sq} \langle v_2 \mid \rho_2\rangle}\ \text{(Seq)}$$

$$\frac{\langle st \mid \rho\rangle \Downarrow_{st} \langle v \mid \rho'\rangle}{\langle st \mid \rho\rangle \Downarrow_{sq} \langle v \mid \rho'\rangle}\ \text{(St)} \qquad \frac{\langle e \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho'\rangle}{\langle e \mid \rho\rangle \Downarrow_{sq} \langle v \mid \rho'\rangle}\ \text{(Expr)}$$

$$\frac{\langle e \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho_1\rangle}{\langle l \texttt{<-} e \mid \rho\rangle \Downarrow_{st} \langle v \mid \rho_1[l \mapsto v]\rangle}\ \text{(Ass)} \qquad \frac{\langle e \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho_1\rangle}{\langle \texttt{def}\ l{=}e \mid \rho\rangle \Downarrow_{st} \langle v \mid \rho_1[l \mapsto v]\rangle}\ \text{(Def)}$$

$$\frac{}{\langle v \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho\rangle}\ \text{(Val)} \qquad \frac{\rho(l) = v}{\langle l \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho\rangle}\ \text{(Loc)}$$

$$\frac{\langle e_1 \mid \rho\rangle \Downarrow_{ex} \langle n_1 \mid \rho_1\rangle \qquad \langle e_2 \mid \rho_1\rangle \Downarrow_{ex} \langle n_2 \mid \rho_2\rangle \qquad \tilde{n} = \tilde{n}_1 +^{\texttt{int}} \tilde{n}_2}{\langle e_1 {+} e_2 \mid \rho\rangle \Downarrow_{ex} \langle n \mid \rho_2\rangle}\ \text{(Sum)}$$

$$\frac{\begin{array}{c}\langle e \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho_1\rangle \\ (v = \texttt{tr} \wedge \langle bl_1 \mid \rho_1\rangle \Downarrow_{bl} \langle v \mid \rho_2\rangle) \vee (v = \texttt{fls} \wedge \langle bl_2 \mid \rho_1\rangle \Downarrow_{bl} \langle v \mid \rho_2\rangle)\end{array}}{\langle \texttt{if}\ e\ \texttt{then}\ bl_1\ \texttt{else}\ bl_2 \mid \rho\rangle \Downarrow_{ex} \langle v \mid \rho_2\rangle}\ \text{(If)}$$

$$\frac{\begin{array}{c}\langle e_1 \mid \rho\rangle \Downarrow_{ex} \langle \texttt{fun}\ x\texttt{->}bl \mid \rho_1\rangle \qquad \langle e_2 \mid \rho_1\rangle \Downarrow_{ex} \langle v \mid \rho_2\rangle \\ \langle bl[x := l] \mid \rho_2[l \mapsto v]\rangle \Downarrow_{bl} \langle v' \mid \rho_3\rangle \qquad l \notin \mathit{dom}(\rho_2)\end{array}}{\langle e_1\ e_2 \mid \rho\rangle \Downarrow_{ex} \langle v' \mid \rho_3\rangle}\ \text{(App)}$$

$$\frac{\begin{array}{c}\langle e \mid \rho\rangle \Downarrow_{ex} \langle \texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \rho_1\rangle \qquad \langle \overline{e} \mid \rho_1\rangle \Downarrow_{ex} \langle \overline{v} \mid \rho_2\rangle \\ bl' = (bl[\overline{x} := \overline{l}'])[y_i := l_j \mid Y_i = Z_j\ \ 1 \leq i \leq n] \quad \overline{Y} \subseteq \overline{Z} \\ \langle bl' \mid \rho_2[\overline{l}' \mapsto \overline{v}]\rangle \Downarrow_{bl} \langle v' \mid \rho_3\rangle \quad \{\overline{l}'\} \cap \mathit{dom}(\rho_2) = \emptyset\end{array}}{\langle \texttt{exc}(e, \overline{Z} \mapsto \overline{l}, \overline{e}) \mid \rho\rangle \Downarrow_{ex} \langle v' \mid \rho_3\rangle}\ \text{(Code)}$$

Figure 11: Big-step operational semantics for IL

on the right-hand-side. So, after this, the value of l is no longer undefined.
The remaining rules define the evaluation of expressions. Rule (Val) returns the value and rule (Loc) the value contained in the location. The rules for $+$ and `if` are obvious.

In rule (App), we first evaluate the expression on the left-hand-side, that must result in a function, then the actual parameter of the function is evaluated. A location is allocated in the memory, assigning to it its value and the location is substituted for the formal parameter in the body of the function. Note that, being in an imperative language, the formal parameter could be modified in the body of the function, however, this change would not be visible in the calling environment, since the location is new. Finally, the body of the function is evaluated. In the evaluation of the `exc` construct, rule (Code), the first argument has to evaluate to a `code` construct. The body of the construct is bl and the names of its unbindings $\overline{Y}$ should be a subset of the names of the rebindings provided by `exc`, which are $\overline{Z}$. The execution proceeds with the evaluation of the expressions in $\overline{e}$, whose value should be associated to the variables $\overline{x}$. The notation $\langle \overline{e} \mid \rho \rangle \Downarrow_{ex} \langle \overline{v} \mid \rho' \rangle$ stands for

$$\langle e_1 \mid \rho_0 \rangle \Downarrow_{ex} \langle v_1 \mid \rho_1 \rangle \quad \langle e_2 \mid \rho_1 \rangle \Downarrow_{ex} \langle v_2 \mid \rho_2 \rangle \cdots \langle e_n \mid \rho_{n-1} \rangle \Downarrow_{ex} \langle v_n \mid \rho_n \rangle$$

where $\rho_0 = \rho$ and $\rho_n = \rho'$. As for application, new locations $\overline{l}'$ are allocated in the store, associated with the values $\overline{v}$ and substituted for the variables $\overline{x}$ in bl. Instead, the unbound variables $\overline{y}$ are substituted with the locations associated via the correspondence of the names in $\overline{Y}$ and $\overline{Z}$. So through assignment to the (local) variables $\overline{y}$ the variable in the execution environment are modified.

Let C be an `IL` syntactic construct, $Loc(C)$, is the set of locations occurring in C. We define well-formed configurations.

Definition 6. 1. *An* `IL` *store* ρ *is* location closed *if for all* $l \in dom(\rho)$*, we have that* $FV(\rho(l)) = \emptyset$ *and* $Loc(\rho(l)) \subseteq dom(\rho)$*.*

2. *The* `IL` *configuration* $\langle C \mid \rho \rangle$ *is* well-formed *if* ρ *is location closed,* $FV(C) = \emptyset$ *and* $Loc(C) \subseteq dom(\rho)$*.*

The operational semantics of Fig. 11 preserves well-formed configurations, as the following theorem states.

Theorem 7. *Let* $\langle C \mid \rho \rangle$ *be well-formed, if* $\langle C \mid \rho \rangle \Downarrow_{\_} \langle v \mid \rho' \rangle$*, then* $\langle v \mid \rho' \rangle$ *is well-formed.*

Proof:
By induction on the derivation of $\Downarrow_{\_}$. We consider only the rules, (Block), (App) and (Code). For all the other rules the result follows from the induction hypotheses.

Rule (Block) From the fact that $\langle \{se\} \mid \rho \rangle$ is well-formed, ρ is location closed, $FV(\{se\}) = \emptyset$ and $Loc(\{se\}) \subseteq dom(\rho)$. From $FV(\{se\}) = \emptyset$ and definition of Var in Fig. 10 we have that $Var(se) \subseteq def(se) = \{\overline{x}\}$. Therefore, $FV(se[\overline{x} := \overline{l}]) = \emptyset$ and $Loc(se[\overline{x} := \overline{l}]) \subseteq dom(\rho[\overline{l} \mapsto ?])$. Moreover, $\rho[\overline{l} \mapsto ?]$ is location closed. So $\langle se[\overline{x} := \overline{l}] \mid \rho[\overline{l} \mapsto ?] \rangle$ is well-formed.
Let $\langle se[\overline{x} := \overline{l}] \mid \rho[\overline{l} \mapsto ?] \rangle \Downarrow_{sq} \langle v \mid \rho' \rangle$, by induction hypothesis $\langle v \mid \rho' \rangle$ is well-formed.

Rule (App) From the fact that $\langle e_1\ e_2 \mid \rho \rangle$ is well-formed, ρ is location closed, $FV(e_1) = \emptyset$, $FV(e_2) = \emptyset$, $Loc(e_1) \subseteq dom(\rho)$ and $Loc(e_2) \subseteq dom(\rho)$. Therefore, $\langle e_1 \mid \rho \rangle$

is well-formed.
Let $\langle e_1 \mid \rho\rangle \Downarrow_{ex}\langle \texttt{fun}\ x\texttt{->}bl \mid \rho_1\rangle$, by induction hypotheses $\langle \texttt{fun}\ x\texttt{->}bl \mid \rho_1\rangle$ is well-formed. Since $dom(\rho) \subseteq dom(\rho_1)$ we have that $Loc(e_2) \subseteq dom(\rho_1)$, so $\langle e_2 \mid \rho_1\rangle$ is well-formed.
Let $\langle e_2 \mid \rho_1\rangle \Downarrow_{ex}\langle v' \mid \rho_2\rangle$, by induction hypotheses $\langle v' \mid \rho_2\rangle$ is well-formed.
From $\langle \texttt{fun}\ x\texttt{->}bl \mid \rho_1\rangle$ well-formed, we have that $FV(bl) \subseteq \{x\}$ and $Loc(bl) \subseteq dom(\rho_1)$ so also $Loc(bl) \subseteq dom(\rho_2)$. From $\langle v' \mid \rho_2\rangle$ well-formed, we get $FV(v') = \emptyset$ and $Loc(v') \subseteq dom(\rho_2)$. Therefore, $FV(bl[x := l]) = \emptyset$, $\rho_2[l \mapsto v']$ is location closed and $Loc(bl[x := l]) \subseteq dom(\rho_2[l \mapsto v'])$. Therefore, $\langle bl[x := l] \mid \rho_2[l \mapsto v']\rangle$ is well-formed.
Let $\langle bl[x := l] \mid \rho_2[l \mapsto v']\rangle \Downarrow_{bl}\langle v \mid \rho'\rangle$, by induction hypotheses, $\langle v \mid \rho'\rangle$ is well-formed.

Rule (Code) From the fact that $\langle \texttt{exc}(e, \overline{Z} \mapsto \overline{l}, \overline{e}) \mid \rho\rangle$ is well-formed, ρ is location closed, $FV(e) = \emptyset$, $FV(\overline{e}) = \emptyset$, $Loc(e) \subseteq dom(\rho)$, $Loc(\overline{e}) \subseteq dom(\rho)$ and $\{\overline{l}\} \subseteq dom(\rho)$. Therefore, $\langle e \mid \rho\rangle$ is well-formed.
Let $\langle e \mid \rho\rangle \Downarrow_{ex}\langle \texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \rho_1\rangle$, by induction hypotheses $\langle \texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \rho_1\rangle$ is well-formed. From the fact that ρ_1 is location closed we have that $\langle \overline{e} \mid \rho_1\rangle$ is well-formed.
Let $\langle \overline{e} \mid \rho_1\rangle \Downarrow_{ex}\langle \overline{v} \mid \rho_2\rangle$, by induction hypotheses, $\langle \overline{v} \mid \rho_2\rangle$ is well-formed. Let $bl' = (bl[\overline{x} := \overline{l}'])[y_i := l_j \mid Y_i = Z_j\ \ 1 \leq i \leq n]$. From the fact that $FV(\texttt{code}(bl, \overline{y} \mapsto \overline{Y}, \overline{x})) = \emptyset$ and definition of Fig. 10 we have that $FV(bl) \subseteq \{\overline{x}, \overline{y}\}$ and so $FV(bl') = \emptyset$. From $Loc(bl) \subseteq dom(\rho_1)$ and $dom(\rho_1) \subseteq dom(\rho_2)$ we have that $Loc(bl') \subseteq dom(\rho_2[\overline{l}' \mapsto \overline{v}])$. Moreover, $\langle \overline{v} \mid \rho_2\rangle$ well-formed implies that $FV(\overline{v}) = \emptyset$, $Loc(\overline{v}) \subseteq dom(\rho_2)$ and therefore $\rho_2[\overline{l}' \mapsto \overline{v}]$ is location closed. Therefore, $\langle bl' \mid \rho_2[\overline{l}' \mapsto \overline{v}]\rangle$ is well-formed.
Let $\langle bl' \mid \rho_2[\overline{l}' \mapsto \overline{v}]\rangle \Downarrow_{bl}\langle v \mid \rho'\rangle$, by induction hypotheses $\langle v \mid \rho'\rangle$ is well-formed. $\square$

An `IL` *program* is a closed block, bl, such that $Loc(bl) = \emptyset$. The *initial configuration* for a program is $\langle \{bl\} \mid []\rangle$. An initial configuration is, therefore, well-formed. From Theorem 7 we derive that the execution of an `IL` program produces a well-formed configuration.

4. Translation of Core `F#` to `IL`

4.1. Challenges of the translation

In our translation we flatten the `let` constructs transforming them into definitions of the corresponding variables followed by the translation of the expression in their body. This may lead, in conjunction to the fact that in an `IL` block we may have forward definitions to a wrong translation of an `F#` expression. E.g., consider the `F#` expression on the top left side of Fig. 12. If its translation was the `IL` code on the top right, it would be incorrect, since in the `IL` code the occurrence of `y` in the body of `f` is bound to the definition of `y` that follows. Therefore the `F#` expression evaluates to 3 whereas its translation in `IL` evaluates to 5. Moreover, the flattening of consecutive `let` constructs, may lead to defining the same variable twice in a block, as the example on the bottom part of Fig. 12 shows. Again the `F#` expression evaluates to 3, whereas its translation in `IL` evaluates to 5 and has two declarations of the variable `y` in the same block. (In JavaScript it is possible, even though strongly discouraged, to have more than one `var` declaration of the same variable.)

```
let y = 3 in
   if ( y = 3) then (
      let f = (fun x -> y) in
         let y = 5 in
            (f y)  )
   else 4
```

```
{ def y = 3;
    if ( y = 3) then {
      def f = (fun x -> { y });
      def y = 5;
      (f y)  }
    else 4 }
```

```
let y = 3 in
   let f = (fun x -> y) in
      let y = 5 in
         (f y)  )
```

```
{ def y = 3;
  def f = (fun x -> { y });
  def y = 5;
  (f y) }
```

Figure 12: Wrong binding of `y` in the the body of `f` and duplicated declaration of `y`

In the translation we use renaming to resolve both these problems, leading to an `IL` program that has the same semantics of the original `F#` expression and in which variables have at most one declaration in each block.

4.2. Formal definition of the translation

We define three translations of `F#` expressions. The first to `IL` expressions, $[\![\cdot]\!]^{I,M}_{ex}$, the second to `IL` sequences, $[\![\cdot]\!]^{I,M}_{sq}$ and the third to `IL` blocks, $[\![\cdot]\!]^{I,M}_{bl}$. The translations are parametric in the sets of the immutable variables, I and mutable variables, M, of the context of the `F#` expression that is translated. The translations produce, in addition to an `IL` expression/sequence/block also variable declarations bound to `code` expressions. The metavariable δ denotes a declaration of a variable "`def` $x{=}e$" and $\overline{\delta}$ a sequence of declarations separated by ";" (semicolon).

Before giving the clauses of the formal translation, we introduce the definition of the wrapping needed to extrude a block from its definition environment and how the construct `exc` rebinds it in the runtime environment.

Definition 8. *Let* se *be an* `IL` *sequence and let* $I = \{\overline{x}\}$ *and* $M = \{\overline{y}\}$ *be sets of variables such that* $I \cap M = \emptyset$ *and* $FV(se) \subseteq I \cup M$*. Define* $seqToExp(se, I, M)$ *to be* $(\texttt{exc}(z, \overline{Y} \mapsto \overline{y}, \overline{x}), \delta)$ *where* δ *is* $\texttt{def}\ z{=}\texttt{code}(\{se\}, \overline{y} \mapsto \overline{Y}, \overline{x})$*,* z *is a fresh variable and* $\overline{Y}$ *are fresh names.*

In the following we give the translations of `F#` expressions into `IL` code. The main translation is the one from `F#` expressions into sequences of statements or expressions. However, to define it we need to define also the translation into `IL` blocks and expressions. The three definitions are mutually recursive.

Definition 9. *Let* e *be an* `F#` *expression such that* $\overline{x}{:}\overline{T}, \overline{y}{:}\overline{T}' \,!\, \mid \emptyset \vdash e : T$ *for some* T *and let* $I = \{\overline{x}\}$ *and* $M = \{\overline{y}\}$*. The functions* $[\![e]\!]^{I,M}_{sq}$*,* $[\![e]\!]^{I,M}_{ex}$*,* $[\![e]\!]^{I,M}_{bl}$ *are defined as follows.*

(sq) *Define* $[\![e]\!]^{I,M}_{sq}$ *from* `F#` *expressions into* `IL` *sequences of statements or expressions by:*

1. *if* e *is* n *or* `tr` *or* `fls` *or* x *or* l, *then* $[\![e]\!]^{I,M}_{sq} = (\, e, \emptyset\,)$
2. $[\![\texttt{fun}\ x{:}T\texttt{->}e]\!]^{I,M}_{sq} = (\,\texttt{fun}\ x\texttt{->}bl, \overline{\delta}\,)$ *where* $[\![e]\!]^{I\cup\{x\},\emptyset}_{bl} {=} (\,bl, \overline{\delta}\,)$
3. $[\![e_1{+}e_2]\!]^{I,M}_{sq} = (\,e'_1{+}e'_2, \overline{\delta}^1;\overline{\delta}^2\,)$ *(or* $[\![e_1\ e_2]\!]^{I,M}_{sq} = (\,e'_1\ e'_2, \overline{\delta}^1;\overline{\delta}^2\,)$*) where* $[\![e_i]\!]^{I,M}_{ex} = (\,e'_i, \overline{\delta}^i\,)$ *(*$1 \le i \le 2$*)*
4. $[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]^{I,M}_{sq} = (\,\texttt{if}\ e'\ \texttt{then}\ bl_1\ \texttt{else}\ bl_2, \overline{\delta};\overline{\delta}^1;\overline{\delta}^2\,)$ *where* $[\![e_1]\!]^{I,M}_{ex} = (\,e', \overline{\delta}\,)$ *and* $[\![e_i]\!]^{I,M}_{bl} = (\,bl_i, \overline{\delta}^i\,)$ *(*$2 \le i \le 3$*)*
5. $[\![e_1, e_2]\!]^{I,M}_{sq} = (\,se_1; se_2, \overline{\delta}^1;\overline{\delta}^2\,)$ *where* $[\![e_i]\!]^{I,M}_{sq} {=} (\,se_i, \overline{\delta}^i\,)$ *(*$1 \le i \le 2$*)*
6. (a) *if* $x \notin I{\cup}M$ *then* $[\![\texttt{let}\ x{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{sq} = (\,\texttt{def}\ x{=}e'_1; se, \overline{\delta}^1;\overline{\delta}^2\,)$ *where* $[\![e_1]\!]^{I,M}_{ex} = (\,e'_1, \overline{\delta}^1\,)$ *and* $[\![e_2]\!]^{I\cup\{x\},M}_{sq} {=} (\,se, \overline{\delta}^2\,)$
   (b) *if* $x \in I \cup M$, *then* $[\![\texttt{let}\ x{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{sq} =$ $[\![\texttt{let}\ z{:}T{=}e_1\ \texttt{in}\ (e_2[x := z])]\!]^{I,M}_{sq}$ *where* z *is a fresh variable*
7. (a) *if* $y \notin I{\cup}M$, *then* $[\![\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{sq} = (\,\texttt{def}\ y{=}e'_1; se, \overline{\delta}^1;\overline{\delta}^2\,)$ *where* $[\![e_1]\!]^{I,M}_{ex} = (\,e'_1, \overline{\delta}^1\,)$ *and* $[\![e_2]\!]^{I,M\cup\{y\}}_{sq} {=} (\,se, \overline{\delta}^2\,)$
   (b) *if* $y \in I \cup M$, *then* $[\![\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{sq} =$ $[\![\texttt{let mutable}\ z{:}T{=}e_1\ \texttt{in}\ (e_2[y := z])]\!]^{I,M}_{sq}$ *where* z *is a fresh variable.*
8. (a) *if* $\{\overline{w}\} \cap (I \cup M) = \emptyset$, *then* $[\![\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e]\!]^{I,M}_{sq} = (\,\texttt{def}\ \overline{w}{=}\overline{F}'; se, \overline{\delta}^1; \cdots ;\overline{\delta}^m;\overline{\delta}'\,)$ *where* $[\![F_j]\!]^{I\cup\{\overline{w}\},M}_{ex} = (\,F'_j, \overline{\delta}^j\,)$ *(* $1 \le j \le m$*) and* $[\![e]\!]^{I\cup\{\overline{w}\},M}_{sq} {=} (\,se, \overline{\delta}'\,)$,
   (b) *if* $\{\overline{w}\} \cap (I \cup M) \neq \emptyset$, *then* $[\![\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{v}\ \texttt{in}\ e]\!]^{I,M}_{sq} =$ $[\![\texttt{let rec}\ \overline{z}{:}\overline{T}{=}(\overline{v}[\overline{w} := \overline{z}])\ \texttt{in}\ (e[\overline{w} := \overline{z}])]\!]^{I,M}_{sq}$ *where* $\overline{z}$ *are fresh variables*
9. $[\![x\texttt{<-}e]\!]^{I,M}_{sq} = (\,x\texttt{<-}e', \overline{\delta}\,)$ *where* $[\![e]\!]^{I,M}_{ex} {=} (\,e', \overline{\delta}\,)$.

(ex) *Define* $[\![e]\!]^{I,M}_{ex}$ *from* `F#` *expressions into* `IL` *expressions as follows. Let* $[\![e]\!]^{I,M}_{sq}$ *be* $(\,se, \overline{\delta}\,)$,

- *if* $se = e'$ *for some expression* e', *then* $[\![e]\!]^{I,M}_{ex} = (\,se, \overline{\delta}\,)$
- *otherwise* $[\![e]\!]^{I,M}_{ex} = (\,e', \delta;\overline{\delta}\,)$ *where* $seqToExp(se, I, M) = (\,e', \delta\,)$.

(bl) *Define* $[\![e]\!]^{I,M}_{bl}$ *from* `F#` *expressions into* `IL` *blocks as follows. Let* $[\![e]\!]^{I,M}_{sq} = (\,se, \overline{\delta}\,)$. *Then* $[\![e]\!]^{I,M}_{bl} = (\,\{se\}, \overline{\delta}\,)$.

Translation of constants, locations and variables is the identity, see clause 1 of the previous definition. The translation of functions, clause 2, produces a function whose body is the translation of the body into a block of the original function. In the translation of the body of the function the variable x is added to the set of free immutable variables I. For sums and applications, clause 3, we first translate into expressions, e_1 and e_2, and then produce the corresponding construct and for the if construct, clause 4, the condition is translated into an expression whereas the branches are translated into blocks. The translation of a sequence, clause 5, is the sequence of the sequences of statements or expressions which are the translations of its subexpressions.

The translations of the `let` constructs, clauses 6, 7 and 8 produce a sequence in which first are the definitions of the variables bound to the translation into expressions of the associated expressions, followed by the translation into a sequence of the body of the `let` construct. For the translations there are two clauses depending on the fact that the variables defined are or are not already present in the context.
In case the variable defined is not already present in the context, to translate the `let` construct to an `IL` sequence, clause 6, we produce a definition of the variable x bound to the translation of e_1 into an `IL` expression followed by the translation of e_2 into a sequence. For the translation of e_2 the variable x is added to the immutable variables of the context.
If the variable is already present in the context (either as immutable or mutable) the `let` construct is α renamed substituting x with a fresh variable z. This insures that z is not in $I \cup M$ or occurs in the `let` expression.
The translation of `let mutable`, clause 7, differs only in the fact that in the translation of e_2, the variable y, being mutable, is added to M.
The translation of the `letrec` construct, clause 8, is similar to the one of the `let` construct, just considering the fact that instead of defining a variable we define a set of variables $\{\overline{w}\}$ (all immutable) and that the variables in $\{\overline{w}\}$ are also free in the function definitions $\overline{F}$.

Finally, the translation of assignment, clause 9, produces an assignment statement in which the variable is assigned the translation to an expression of the expression on the right-hand-side of the assignment expression.

The correctness of the translation of the `let` constructs, relays on the fact that the renaming of variables used does not change the operational semantics of expressions, as the following lemma asserts. In the lemma, if $\sigma = \overline{x}{:}\overline{T}{\mapsto}\overline{v}$ and z is such that $z \notin \{\overline{x}\} \cup FV(\overline{v})$. With $\sigma[x_i := z]$ we denote the stack σ in which x_i is substituted with z, i.e., $[x_1 \cdots x_{i-1}\, z\, x_{i+1} \cdots x_n{:}\overline{T}{\mapsto}\overline{v}]$.

Lemma 10. *Let $\sigma = \overline{x}{:}\overline{T}{\mapsto}\overline{v}$ and $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. Let $\overline{x}' = x_{r_1} \cdots x_{r_n}$ be such that $\{\overline{x}'\} \subseteq \{\overline{x}\}$ and $\overline{z}$ be fresh variables; $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ if and only if $\langle e[\overline{x}' := \overline{z}] \mid \sigma[\overline{x}' := \overline{z}] \mid \rho\rangle \Downarrow \langle v' \mid \rho_1'\rangle$ where $v = v'$, $dom(\rho_1) = dom(\rho_1')$ and for all $l \in dom(\rho_1)$, $\rho_1(l) = \rho_1'(l)$.*

Proof:
By an easy induction on the derivation of $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$, using the fact that equality between values is up-to α-conversion. $\Box$

For the translation of the `F#` construct into an `IL` expressions, we first translate the construct to a sequence of statements or expressions. If the result is an expression (clauses $(1)\div(9)$) then we return it, otherwise we return the `exc` construct generated by the function *seqToExp* and the definition of a new variable bound to a `code` expression, see Definition 8. Note that the sets of mutable and immutable variable of the context are needed to generate the correct matching for the expressions `exc` and `code`.

For the translation of the `F#` construct into an `IL` block, we first translate the construct to a sequence of statements or expressions and then add the curly brackets.

The following lemma asserts the properties of the variables resulting from the translation of an `F#` expression into an `IL` sequence or expression.

Lemma 11. *Let $\overline{x}\,\overline{y}$ be a sequence of distinct variables. Let e be such that $FV(e) \subseteq \{\overline{x}, \overline{y}\}$ and $Loc(e) = \emptyset$. Let $[\![e]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\,se, \overline{\delta}\,)$ where $\overline{z} = def(\overline{\delta})$ and $\overline{w} = def(se)$. Then*

1. *the variables in the sequence $\overline{x}\,\overline{y}\,\overline{w}\,\overline{z}$ are distinct*
2. *$FV(se) \subseteq \{\overline{x}, \overline{y}, \overline{z}\}$*
3. *for all* `def` *$z{=}e' \in \overline{\delta}$, we have that $FV(e') \subseteq \{\overline{z}\}$*
4. *for all in $z \in def(\overline{\delta})$ there is only one* `def` *$z{=}\_$ in $\overline{\delta}$ and*
5. *for all in $w \in def(se)$ there is only one* `def` *$w{=}\_$ in se.*

Properties $1 \div 4$ hold also for $[\![e]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}} = (\,se, \overline{\delta}\,)$. Note that in this case since se is an expression, $\{\overline{w}\} = \emptyset$.

Proof:
By induction on the definition of the translation relations $\approx_{sq}$ and $\approx_{ex}$ (Definition 16(sq) and (ex)). The renaming in the clauses for the let constructs is needed to enforce the property that the variables in $def(se)$ are disjoint from the one in $\{\overline{x}\,\overline{y}\}$. The variables in $\{\overline{z}\}$, introduced in the translation $[\![e]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}}$, are fresh so they are disjoint from any other variable. $\square$

4.3. Correctness of the Translation

The translation preserves the dynamic semantics of the `F#` expressions. That is consider an `F#` program e and its translation in then `IL` program bl. We want to show that, if the evaluation in `F#` of e in the empty store produces a primitive value v, then the evaluation of the block bl in the empty store in `IL` produces v. Moreover, if the evaluation in `F#` of e does not terminate, then also the the evaluation of the block bl in `IL` does not terminate.

Theorem 12 (Correctness). *Let e be an* `F#`*program and let $[\![e]\!]_{sq}^{\emptyset,\emptyset}{=}(\,se, \overline{\delta}\,)$. Then $\langle e \mid [\,] \mid [\,]\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some ρ_* and v either integer or boolean value if and only if $\langle\{\overline{\delta}; se\} \mid [\,]\rangle[\,] \Downarrow_{bl}\langle v \mid \rho'_*\rangle$ for some ρ'_*.*

The proof of the theorem relays on Lemmas 18 and 19 and will be given at the end of the current section.

To prove the result we define a *translation relations* between `F#` values and `IL` values and between well-formed configurations of `F#`and well-formed configurations of `IL`. Then we show that if the evaluation of an `F#` configuration converges, then the evaluation of the related `IL` configuration converges and the result is a related value, Lemma 18; moreover, if the evaluation of an `F#` configuration does not converge to a value, then the evaluation of the related `IL` configuration does not converge, Lemma 19. Therefore, since an integer/boolean value in `F#` is translated into the same integer/-boolean value in `IL` this proves the result.

Looking at the translation from `F#` to `IL` we can see that both immutable and mutable variables of `F#` are translated into `IL` (mutable) variables that are allocated in the store. Moreover, when we translate to an expression an `F#` expression whose translation produces a sequence we introduce new variables, see Definition 8, which will be bound to `code` constructs. So, the store, ρ', of a runtime `IL` configuration produced by the execution of the translation of an `F#` expression can be partitioned in two parts, ρ^M and ρ^I, where ρ^M correspond to store of the `F#` configuration, containing the values of the mutable variables and ρ^I records the values of the `F#` immutable variables and contains the values of the variables bound to `code` constructs. We assume that the names of the location in ρ^M are equal to the corresponding ones in the `F#` store. Looking at the evaluation rules of Fig. 11, we can see that, the locations in ρ^I will be assigned ?, by the rule (Block), when they are allocated in the store, before starting the execution of the sequence of statements or expressions of the block containing the definition of the immutable variables. Then they are assigned a value, when executing the `def` statement associated to their definition. After this, the value of the associated location does not change. In the following, when need to identify the two portions of the `IL` store we denote ρ' by $\rho^I + \rho^M$. The partitioning is always identified by the fact that the name of locations in ρ^M coincide with the one of the locations in the corresponding `F#` store.

We now define the representation on the `IL` memory of a sequence of `code` constructs.

Definition 13. *Let ρ' be an `IL` store and $\overline{\delta}$ be*

$$\texttt{def } z_1 = \texttt{code}(bl_1, \overline{y}^1 \mapsto \overline{Y}^1, \overline{x}^1); \cdots ; \texttt{def } z_m = \texttt{code}(bl_m, \overline{y}^m \mapsto \overline{Y}^m, \overline{x}^m)$$

$\overline{\delta}$ is represented by $\overline{l}$ in ρ' *if* $\{\overline{l}\} \subseteq dom(\rho')$ *and* $\rho'(l_i) = \texttt{code}(bl_i[\overline{z} := \overline{l}], \overline{y}^i \mapsto \overline{Y}^i, \overline{x}^i)$ *($1 \leq i \leq m$).*

To define the correspondence between an `F#` configuration and the `IL` configuration which arises from its translation we first define a *translation relation between `F#` values and `IL` values.* For function values, we need the `IL` store since the (immutable) variables of the stack are translated in `IL` variables, which are, at runtime, allocated in the store. Moreover, the `IL` store contains the `code` constructs generated by the translation. So we relate an `F#` value with an `IL` configuration containing a value.

We assume equality between primitive values in `F#` and `IL`. In order to define the equivalence between an `F#` function (or recursive function) F and its `IL` translation, say F' we have to establish equivalence between the `F#` values associated to variables in the definition stack of F, and the `IL` value contained in the location of the `IL` store corresponding. To break circularity, we define the relation by induction on the depth of the scope in which the value is defined, starting from primitive values and top level functions that have an empty definition stack.

Definition 14. *The* translation relation between `F#` values and `IL` configurations, $\cong$, *is defined by $v \cong \langle v' \mid \rho' \rangle$, if for some $k \in \mathbb{N}$, $v \cong_k \langle v' \mid \rho' \rangle$, where $\rho' = \rho^I + \rho^M$, and the relations $\cong_k$ are defined by induction on k as follows*

1. $\cong_0$, *is defined by*
   (a) $n \cong_0 \langle n \mid \rho' \rangle$, $\texttt{tr} \cong_0 \langle \texttt{tr} \mid \rho' \rangle$, $\texttt{fls} \cong_0 \langle \texttt{fls} \mid \rho' \rangle$, *for any ρ'*
   (b) $(F, \emptyset) \cong_0 \langle v'[\overline{z} := \overline{l}^z] \mid \rho' \rangle$, *if v', $\overline{z}$, and $\overline{l}^z$ are such that*

i. $[\![F]\!]_{sq}^{\emptyset,\{\overline{y}\}} = (\, v', \overline{\delta}\,)$ *for any* $\overline{y}$ *and*

ii. $\overline{\delta}$ *is represented by* $\overline{l}^z$ *in* ρ'

(c) $(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \emptyset) \cong_0 \langle v'_i[\overline{w}\,\overline{z}^i := \overline{l}^w\,\overline{l}^{z_j}] \mid \rho'\rangle$, *if* $1 \leq i \leq m$ *and there are* $\overline{l}^w$, $\{\overline{l}^w\} \subseteq dom(\rho^I)$, *such that for all* j, $1 \leq j \leq m$,

i. $[\![F_j]\!]_{sq}^{\{\overline{w}\},\{\overline{y}\}} = (\, v'_j, \overline{\delta}^j\,)$, *for any* $\overline{y}$, $\overline{z}^j = def(\overline{\delta}^i)$,

ii. $\rho'(l_j^w) = v'_j[\overline{w}\,\overline{z}^j := \overline{l}^w\,\overline{l}^{z_j}]$ *and*

iii. $\overline{\delta}^j$ *is represented by* $\overline{l}^{z_j}$ *in* ρ'.

2. $\cong_{k+1}$, *is defined by*

(a) $(F, \sigma) \cong_{k+1} \langle v'[\overline{x}\,\overline{z} := \overline{l}^x\,\overline{l}^z] \mid \rho'\rangle$, *if* v', $\overline{x}$, $\overline{z}$, $\overline{l}^x$ *and* $\overline{l}^z$ *are such that*

i. $\overline{x} = dom(\sigma)$, $\{\overline{l}^x\} \subseteq dom(\rho^I)$,

ii. $[\![F]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, v', \overline{\delta}\,)$ *for any* $\overline{y}$,

iii. $\overline{\delta}$ *is represented by* $\overline{l}^z$ *in* ρ' *and*

iv. *for all* i, $1 \leq i \leq n$, *let* $x_i{:}T_i{\mapsto}v_i \in \sigma$, *we have that* $v_i \cong_h \langle \rho'(l_i^x) \mid \rho'\rangle$, *for some* $h \leq k$

(b) $(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma) \cong_{k+1} \langle v'_i[\overline{x}\,\overline{w}\,\overline{z}^i := \overline{l}^x\,\overline{l}^w\,\overline{l}^{z_i}] \mid \rho'\rangle$, *if* $1 \leq i \leq m$ *and there are* $\overline{l}^w$ *and* $\overline{l}^x$, $\{\overline{l}^w, \overline{l}^x\} \subseteq dom(\rho^I)$, *such that for all* j, $1 \leq j \leq m$,

- i. $[\![F_j]\!]_{sq}^{\{\overline{x},\overline{w}\},\{\overline{y}\}} = (\, v'_j, \overline{\delta}^j\,)$, *for any* $\overline{y}$, $\overline{z}^j = def(\overline{\delta}^i)$,

  ii. $\rho'(l_j^w) = v'_j[\overline{x}\,\overline{w}\,\overline{z}^j := \overline{l}^x\,\overline{l}^w\,\overline{l}^{z_j}]$,

  iii. $\overline{\delta}^j$ *is represented by* $\overline{l}^{z_j}$ *in* ρ',
- *for all* p, $1 \leq p \leq n$, *let* $x_p{:}T_p{\mapsto}v_p \in \sigma$, *we have that* $v_p \cong_h \langle \rho'(l_p^x) \mid \rho'\rangle$, *for some* $h \leq k$.

All the locations involved in the previous definition, $\overline{l}^x$, $\overline{l}^z$ and $\overline{l}^w$ are in the domain of ρ^I. So the translation relation between F# and IL values depends only on ρ^I. Therefore, since, as we will prove evaluation of IL constructs does not modify ρ^I, the existing relations between F# and IL values are preserved by evaluation. If the value of a recursive definition is in the translation relation with an IL configuration, then the function value which is the result of the lookup function of Definition 1 is also in the translation relation with the IL configuration.

Lemma 15. *If for all* i, $1 \leq i \leq m$ *we have* $(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma) \cong \langle v'_i[\overline{x}\,\overline{w}\,\overline{z}^i := \overline{l}^x\,\overline{l}^w\,\overline{l}^{z_i}] \mid \rho'\rangle$, *then for all* j, $1 \leq j \leq m$ *we have*

$$(F_j, \sigma[w_i{:}T_i{\mapsto}(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)]_{1\leq i\leq m}) \cong \langle v'_j[\overline{x}\,\overline{w}\,\overline{z}^i := \overline{l}^x\,\overline{l}^w\,\overline{l}^{z_i}] \mid \rho'\rangle.$$

Proof:
Immediate from Definition 14.2 (b) and (c). □

We define two translation relations between well-typed F# configurations and well-formed IL configurations. The first and main relation, $\approx_{sq}$, between F# configurations and IL configurations containing a sequence of statements and expressions, and the second, $\approx_{ex}$, in which the IL configuration contains an expression.

Definition 16. *Let $\langle e \mid \sigma \mid \rho\rangle$ be an* `F#` *configuration such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ where $\Sigma = \overline{l}{:}\overline{T}$ and $\overline{x} = dom(\sigma)$.*
Let e° be such that $e = e^\circ[\overline{y} := \overline{l}^y]$, $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ for some T and $\overline{l}^y{:}\overline{T}^y \subseteq \Sigma$.
Let $[\![e^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ, \overline{\delta}\,)$ where $\overline{z} = def(\overline{\delta})$ and $\overline{w} = def(se^\circ)$.

1. $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{sq} \langle se \mid \rho'\rangle$ *if $\langle se \mid \rho'\rangle$ is well-formed*
   (a) *$dom(\rho) = dom(\rho^M) \subseteq dom(\rho')$ (so $\rho' = \rho^I + \rho^M$) and there are mutually disjoint sets of locations $\{\overline{l}^x\}$, $\{\overline{l}^z\}$, $\{\overline{l}^w\}$ disjoint from $\{\overline{l}^y\}$ such that*
   (b) *$se = se^\circ[\overline{x}\,\overline{y}\,\overline{w}\,\overline{z} := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,\overline{l}^z]$*
   (c) *$\{\overline{l}^x\} \subseteq dom(\rho^I)$, $\overline{\delta}$ is represented by $\overline{l}^z$ in ρ' and $\rho'(\overline{l}^w) = \overline{?}$*
   (d) *for all i, $1 \leq i \leq n$, let $x_i{:}T_i{\mapsto}v_i \in \sigma$, we have that $v_i \cong \langle \rho'(l_i^x) \mid \rho^I + \rho^M\rangle$,*
   (e) *for all $l \in dom(\rho)$, we have that $\rho(l) \cong \langle \rho'(l) \mid \rho^I + \rho^M\rangle$.*
2. $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{ex} \langle e' \mid \rho'_e\rangle$ *if there are se and ρ' such that,*
   $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{sq} \langle se \mid \rho'\rangle$*, and*
   (a) *if se is an expression, then $e' = se$ and $\rho'_e = \rho'$*
   (b) *otherwise $e' = \texttt{exc}(l', \overline{Y} \mapsto \overline{l}^y, \overline{l}^x)$ and ρ'_e is such that:*
   $\rho'_e(l') = \texttt{code}(\{se^\circ\}[\overline{z} := \overline{l}^z], \overline{y} \mapsto \overline{Y}, \overline{x})$ where $l' \notin dom(\rho')$ and for all $l \in dom(\rho) \cup \{\overline{l}^x, \overline{l}^z\}$ we have that $\rho'_e(l) = \rho'(l)$.

Note that, given an `F#` expression e and the sequences of variables $\overline{x}$ and $\overline{y}$ the `IL` sequence se is uniquely determined. Moreover, if e is not a `let` expression or a sequence of expressions or an assignment the two definitions, $\approx_{sq}$ and $\approx_{ex}$ coincide.

Lemma 11 insures that the relation $\approx_{se}$ is well defined, in particular, that in clause 1(b), the variables in the sequence $\overline{x}\,\overline{y}\,\overline{w}\,\overline{z}$ are all distinct and and the free variables of se° should be a subset of $\{\overline{x}, \overline{y}, \overline{z}\}$ so $FV(se) = \emptyset$.

The following lemma shows how the translation relation between an `F#` expression e and an `IL` sequence se induces relations between the subexpressions of e and se. of se.

Lemma 17. *Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{sq} \langle se \mid \rho'\rangle$ be such that $e = e^\circ[\overline{y} := \overline{l}^y]$ where $\overline{l}^y{:}\overline{T}^y \subset \Sigma$ and $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ for some T. Then*

1. *if $e{=}x$, then $x = x_i$ for some $x_i \in \{\overline{x}\}$ and $se = l_i^x$*
2. *if $e{=}n$ or $e{=}\texttt{tr}$ or $e{=}\texttt{fls}$, then $se{=}e$ and $e \cong \langle se \mid \rho'\rangle$*
3. *if $e{=}\texttt{fun}\ x{:}T_1\texttt{->}e_1$, then $se{=}\texttt{fun}\ x\texttt{->}bl$ for some bl, and $(e, \sigma) \cong \langle se \mid \rho'\rangle$*
4. *if $e{=}l$ for some location l, then $l = l_j^y$ for some $l_j^y \in \overline{l}^y$ and $se = l_j^y$*
5. *if $e{=}e_1{+}e_2$, then $T = \texttt{int}$,*
   (a) *$e^\circ = e_1^\circ{+}e_2^\circ$ for some e_i° such that $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$, $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_i^\circ : \texttt{int}$ and $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_i : \texttt{int}$ $(1 \leq i \leq 2)$*
   (b) *$[\![e_i^\circ]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}} = (\, se_i^\circ, \overline{\delta}^i\,)$ for some $\overline{\delta}^i$ with $def(\overline{\delta}^i) = \{\overline{z}^i\}$ and $\overline{l}^{z_i}$ such that $\overline{\delta}^i$ is represented by $\overline{l}^{z_i}$ in ρ' $(1 \leq i \leq 2)$*
   (c) *$se = e_1'{+}e_2'$ where $e_i' = se_i^\circ[\overline{x}\,\overline{y}\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_i}]$ $(1 \leq i \leq 2)$*
6. *if $e{=}\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ then*

(a) $e^\circ = \mathtt{if}\ e_1^\circ\ \mathtt{then}\ e_2^\circ\ \mathtt{else}\ e_3^\circ$ *for some* e_i° *such that* $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \le i \le 3)$
$env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_1^\circ : \mathtt{bool}$ $\quad env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : \mathtt{bool}$
$env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_j^\circ : T$ $\quad env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_j : T$ $(2 \le j \le 3)$

(b) $[\![e_i^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se_i^\circ, \overline{\delta}^i\,)$ *for some* $\overline{\delta}^i$ *with* $def(\overline{\delta}^i) = \{\overline{z}^i\}$ *and* $\overline{l}^{z_i}$ *such that* $\overline{\delta}^i$ *is represented by* $\overline{l}^{z_i}$ *in* ρ' $(2 \le i \le 3)$*,*

(c) $se = \mathtt{if}\ e_1'\ \mathtt{then}\ \{se_2\}\ \mathtt{else}\ \{se_3\}$*, where*
  i. $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e_1' \mid \rho'\rangle$,
  ii. $se_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{w}^i\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,\overline{l}^{z_i}]$ *where* $def(se_i^\circ) = \{\overline{w}\}$ *and* $\rho'(\overline{l}^{w_i}) = \overline{?}$ $(2 \le i \le 3)$

7. *if* $e = e_1\ e_2$*, then*

(a) $e^\circ = e_i^\circ\ e_i^\circ$ *for some* e_i° *such that* $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \le i \le 2)$
$env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_1^\circ : T' \to T$ *and* $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T' \to T$
$env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_2^\circ : T'$ *and* $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_2 : T'$ *for some* T'

(b) $[\![e_i^\circ]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}} = (\, se_i^\circ, \overline{\delta}^i\,)$ *for some* $\overline{\delta}^i$ *with* $def(\overline{\delta}^i) = \{\overline{z}^i\}$ *and* $\overline{l}^{z_i}$ *such that* $\overline{\delta}^i$ *is represented by* $\overline{l}^{z_i}$ *in* ρ' $(1 \le i \le 2)$

(c) $se = e_2'\ e_1'$ *where* $e_i' = se_i^\circ[\overline{x}\,\overline{y}\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_i}]$ $(1 \le i \le 2)$

8. *if* $e{=}e_1, e_2$ *then*

(a) $e^\circ = e_1^\circ, e_2^\circ$ *for some* e_i° *such that* $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$*,* $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_i^\circ : T_i$ *and* $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_i : T_i$ $(1 \le i \le 2)$ *for some* T_i *such that* $T_2 = T$

(b) $[\![e_i^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se_i^\circ, \overline{\delta}^i\,)$ *for some* $\overline{\delta}^i$ *with* $def(\overline{\delta}^i) = \{\overline{z}^i\}$ *and* $\overline{l}^{z_i}$ *such that* $\overline{\delta}^i$ *is represented by* $\overline{l}^{z_i}$ *in* ρ' $(1 \le i \le 2)$

(c) $se = se_1; se_2$*, where* $se_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{w}^i\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,\overline{l}^{z_i}]$ *where* $def(se_i^\circ) = \{\overline{w}^i\}$ *and* $\rho'(\overline{l}^{w_i}) = \overline{?}$ $(1 \le i \le 2)$

9. *if* $e{=}\mathtt{let}\ x{:}T'{=}e_1\ \mathtt{in}\ e_2$ *and* $x \notin \{\overline{x}, \overline{y}\}$*, then*

(a) $e^\circ{=}\mathtt{let}\ x{:}T'{=}e_1^\circ\ \mathtt{in}\ e_2^\circ$ *for some* e_i° *such that* $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \le i \le 2)$
$env(\sigma) \mid \emptyset \vdash e_1^\circ : T'$ *and* $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T'$
$env(\sigma), x{:}T, \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_2^\circ : T$ *and* $env(\sigma), x{:}T \mid \overline{l}^y{:}\overline{T}^y \vdash e_2 : T$

(b) $[\![e_1^\circ]\!]_{ex}^{I,M} = (\, e_1'', \overline{\delta}^1\,)$ *and* $[\![e_2^\circ]\!]_{sq}^{\{x\,\overline{x}\},\{\overline{y}\}} = (\, se_2^\circ, \overline{\delta}^2\,)$ *with* $def(\overline{\delta}^i) = \{\overline{z}^i\}$ *such that* $\overline{\delta}^i$ *is represented by* $\overline{l}^{z_i}$ *in* ρ' $(1 \le i \le 2)$*,*

(c) $se = \mathtt{def}\ l_x{=}e_1'; se_2$ *where*
  i. $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e_1' \mid \rho'\rangle$ *for* $e_1' = e_1''[\overline{x}\,\overline{y}\,\overline{z}^1 := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_1}]$*,*
  ii. $se_2 = se_2^\circ[x\,\overline{x}\,\overline{y}\,\overline{w}\,\overline{z}^2 := l_x\,\overline{l}^x\,\overline{l}^y\,\overline{l}^w\,\overline{l}^{z_2}]$ *where* $\rho'(l_x) = ?$*,* $\overline{w} = def(se_2^\circ)$ *and* $\overline{l}^w$ *are such that* $\rho'(\overline{l}^w) = \overline{?}$*.*

10. *if* $e{=}\mathtt{let\ mutable}\ y{:}T{=}e_1\ \mathtt{in}\ e_2$ *and* $y \notin \{\overline{x}, \overline{y}\}$*, then*

(a) $e^\circ{=}\mathtt{let\ mutable}\ y{:}T'{=}e_1^\circ\ \mathtt{in}\ e_2^\circ$ *for some* e_i° *such that* $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \le i \le 2)$ *and for* $l_y \notin \{\overline{l}^y\}$ *we have*
$env(\sigma) \mid \emptyset \vdash e_1^\circ : T'$ *and* $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T'$
$env(\sigma), y{:}T'!, \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_2^\circ : T$ *and* $env(\sigma) \mid l_y{:}T', \overline{l}^y{:}\overline{T}^y \vdash e_2[y := l_y] : T$

(b) $[\![e_1^\circ]\!]_{ex}^{I,M} = (\, e_1'', \overline{\delta}^1\,)$ *and* $[\![e_2^\circ]\!]_{sq}^{\{\overline{x}\},\{y\,\overline{y}\}} = (\, se_2^\circ, \overline{\delta}^2\,)$ *with* $def(\overline{\delta}^i) = \{\overline{z}^i\}$ *such that* $\overline{\delta}^i$ *is represented by* $\overline{l}^{z_i}$ *in* ρ' $(1 \le i \le 2)$*,*

   (c) $se = \texttt{def}\ l_y{=}e'_1; se_2$ *where*
      i. $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e'_1 \mid \rho'\rangle$ *for* $e'_1 = e''_1[\overline{x}\,\overline{y}\,\overline{z}^1 := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_1}]$,
      ii. $se_2 = se^\circ_2[\overline{x}\,y\,\overline{y}\,\overline{w}\,\overline{z}^2 := \overline{l}^x\,l_y\,\overline{l}^y\,\overline{l}^w\,\overline{l}^{z_2}]$ *where* $\rho'(l_y) =?$, $\overline{w} = \mathit{def}(se^\circ_2)$ *and* $\overline{l}^w$ *are such that* $\rho'(\overline{l}^w) = \overline{?}$.

11. *if* $e{=}\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e_1$ *and* $\{\overline{w}\} \cap \{\overline{x}, \overline{y}\} = \emptyset$*, then*
   (a) $e^\circ{=}\texttt{let rec}\ \overline{w}{=}\overline{F}\ \texttt{in}\ e^\circ_1$ *is such that* $e_1{=}e^\circ_1[\overline{y} := \overline{l}^y]$
   $\mathit{env}(\sigma), \overline{w}{:}\overline{T}, \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ_1 : T$ *and* $\mathit{env}(\sigma), \overline{w}{:}\overline{T} \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T$
   $\mathit{env}(\sigma), \overline{w}{:}\overline{T} \mid \emptyset \vdash F_j : T_j \quad (1 \le j \le m)$,
   (b) $\llbracket e^\circ_1 \rrbracket^{\{\overline{x},\overline{w}\},\{\overline{y}\}}_{sq} = (\, se^\circ_1, \overline{\delta}\,)$ *with* $\mathit{def}(\overline{\delta}) = \{\overline{z}\}$ *and* $\overline{\delta}$ *is represented by* $\overline{l}^z$ *in* ρ',
   $\llbracket F_j \rrbracket^{\{\overline{w}\,\overline{x}\},\{\overline{y}\}}_{sq} = (\, e'_j, \overline{\delta}^j\,)$ *with* $\mathit{def}(\overline{\delta}^j) = \{\overline{z}^j\}$ *such that* $\overline{\delta}^j$ *is represented by* $\overline{l}^{z_j}$ *in* ρ' $(1 \le j \le m)$
   (c) $se = \texttt{def}\ \overline{l}^w{=}\overline{e}''; se_1$ *where*
      i. $e''_j = e'_j[\overline{w}\,\overline{x}\,\overline{z}^j := \overline{l}^w\,\overline{l}^x\,\overline{l}^{z_j}]$ *and* $\mathit{def}(e'_j) = \emptyset$ *(since* e'_j *is an expression)* $(1 \le j \le m)$
      ii. $se_1 = se^\circ_1[\overline{w}\,\overline{x}\,\overline{w}'\,\overline{z} := \overline{l}^w\,\overline{l}^x\,\overline{l}^{w'}\,\overline{l}^z]$ *where* $\overline{w}' = \mathit{def}(se^\circ_1)$ *and* $\overline{l}^{w'}$ *are such that* $\rho'(\overline{l}^{w'}) = \overline{?}$.

12. *if* $e{=}l{<}{-}e_1$*, then*
   (a) $e^\circ = y{<}{-}e^\circ_1$ *such that* $e_1 = e^\circ_1[\overline{y} := \overline{l}^y]$, $l = l^y_j$ *for some* j $\mathit{env}(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ_1 : T^y_j$ *and* $\mathit{env}(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T^y_j$
   (b) $se = l{<}{-}e'_1$ *where* $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e'_1 \mid \rho'\rangle$

Proof:

1. If $e = x$, then $e^\circ = x$ and since $\mathit{env}(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ for some i, we have that $x_i \in \mathit{dom}(\sigma)$ and $e^\circ = x_i$. From Definition 9.1 then $se^\circ = x_i$ and $se = l^x_i$.
2. In these cases $e^\circ{=}e$. From Definition 9.1 $se^\circ{=}e$ and so also $se{=}e$. From Definition 14.1(a), we have $e \cong_0 \langle se \mid \rho'\rangle$, therefore $e \cong \langle se \mid \rho'\rangle$.
3. Let $F{=}\texttt{fun}\ x{:}T_1{-}{>}e_1$. From $\Sigma, \overline{x}, \overline{y} \models \langle F \mid \rho\rangle \approx \langle se \mid \rho'\rangle$, we get $e^\circ{=}\texttt{fun}\ x{:}T_1{-}{>}e^\circ_1$ is such that $\mathit{env}(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash \texttt{fun}\ x{:}T_1{-}{>}e^\circ_1 : T_1 \to T_2$, for some T_2 and $T = T_1 \to T_2$. From rule (TyAbs) of Fig. 3 we have that $\mathit{env}(\sigma) \mid \emptyset \vdash \texttt{fun}\ x{:}T_1{-}{>}e^\circ_1 : T$. Therefore, $e^\circ{=}F$. From Definition 9.2, $\llbracket F \rrbracket^{\{\overline{x}\},\{\overline{y}\}}_{sq} = (\,\texttt{fun}\ x{-}{>}bl', \overline{\delta}\,)$, where $\llbracket e_1 \rrbracket^{\{\overline{x},x\},\emptyset}_{bl} = (\, bl', \overline{\delta}\,)$ and from Definition 16.1(b) and (c), since $\mathit{def}(\texttt{fun}\ x{-}{>}bl') = \emptyset$ and $\{\overline{y}\} \cap FV(F) = \emptyset$, then $se{=}se^\circ[\overline{x}\,\overline{z} := \overline{l}^x\,\overline{l}^z]$, where $\mathit{def}(\overline{\delta}) = \overline{z}$, $\overline{\delta}$ is represented by $\overline{l}^z$ in ρ'.
   From Definition 16.1(d), for all i, $1 \le i \le n$, let $x_i{:}T_i{\mapsto}v_i \in \sigma$, we have that $v_i \cong \langle \rho'(l^x_i) \mid \rho^I + \rho^M\rangle$. Therefore, for all i, $1 \le i \le n$, we have that, for some k_i, $v_i \cong_{k_i} (\rho'(l^x_i), \rho^I + \rho^M)$. Let $k = \max\{k_i \mid 1 \le i \le n\}$, from Definition 14.2(a), $(F, \sigma) \cong_{k+1} \langle se \mid \rho'\rangle$ and therefore $(F, \sigma) \cong \langle se \mid \rho'\rangle$.
4. If $e = l$, from $\mathit{env}(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ we derive that e° may not contain locations. Therefore, for some j, $e = l^y_j$ and $e^\circ = y_j$. From Definition 9.1 then $se^\circ = y_j$ and so $se = l^y_j$.

All the other cases derive directly from Definition 16 (translation relations) and Definition 9(sq), (ex) and (bl) (translation to sequences, expressions and blocks). We show only the proof for the case of `let mutable` which is one of the most complex cases.

10. Let e be `let mutable` $y{:}T'{=}e_1$ `in` e_2. From $y \notin \{\overline{x}, \overline{y}\}$ and $e = e^\circ[\overline{y} := \overline{l}^y]$, we have that $e^\circ{=}$`let mutable` $y{:}T{=}e_1^\circ$ `in` e_2° where $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \leq i \leq 2)$. From $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ and rule (TyMut), we derive $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e_1^\circ : T'$ and $env(\sigma), \overline{y}{:}\overline{T}^y!, y{:}T'! \mid \emptyset \vdash e_2^\circ : T$. Therefore, $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1^\circ[\overline{y} := \overline{l}^y] : T'$ and if $l_y \notin \{\overline{l}^y\}$ then $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y, l_y{:}T' \vdash e_2^\circ[\overline{y}\, y := \overline{l}^y\, l_y] : T$. So clause (a) holds.
Let $[\![e^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ, \overline{\delta}\,)$ where $\overline{z} = def(\overline{\delta})$, $\overline{w} = def(se^\circ)$ and $\overline{z}$ is represented by $\overline{l}^z$ in ρ' for some $\overline{l}^z$. From Definition 9(sq).7, se° is `def` $y{=}e_1''; se_2^\circ$ and $\overline{\delta} = \overline{\delta}^1; \overline{\delta}^2$ where
$(*)$ $[\![e_1^\circ]\!]_{ex}^{I,M} = (\, e_1'', \overline{\delta}^1\,)$ and
$(\star)$ $[\![e_2^\circ]\!]_{sq}^{I,M\cup\{y\}} {=} (\, se_2^\circ, \overline{\delta}^2\,)$.
Since $def(se^\circ) = def(se_2^\circ) \cup \{y\}$, from Definition 16.1(b) we have that $se = se^\circ[\overline{x}\, y\, \overline{y}\, \overline{w}\, \overline{z} := \overline{l}^x\, l_y\, \overline{l}^y\, \overline{l}^w\, \overline{l}^z]$. Therefore $se =$ `def` $l_y{=}e_1'; se_2$ where
$(**)$ $e_1' = e_1''[\overline{x}\, \overline{y}\, \overline{z} := \overline{l}^x\, \overline{l}^y\, \overline{l}^z]$
$(\star\star)$ $se_2 = se_2^\circ[\overline{x}\, y\, \overline{y}\, \overline{w}\, \overline{z} := \overline{l}^x\, l_y\, \overline{l}^y\, \overline{l}^w\, \overline{l}^z]$.
Let $\overline{z}^i = def(\overline{\delta}^i)$ $(1 \leq i \leq 2)$. From $\{\overline{z}^1\} \cap \{\overline{z}^2\} = \emptyset$ we have that $\overline{z} = \overline{z}^1\overline{z}^2$, $\overline{l}^z = \overline{l}^{z_1}\overline{l}^{z_2}$ and $\overline{\delta}^i$ is represented by $\overline{l}^{z_i}$ in ρ' $(1 \leq i \leq 2)$. Moreover, from Definition 16.1(c) both $\rho'(l_y) = ?$ and $\rho'(\overline{l}^w) = \overline{?}$. So from $(\star)$ and $(\star\star)$ we derive clauses (b) and (c).ii of the result.
From clauses (a), (c), and (d) of Definition 16 for $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{sq} \langle se \mid \rho'\rangle$, $(*)$, $(**)$ and Definition 9(ex) we get $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e_1' \mid \rho'\rangle$ (clause (c).i), which concludes the proof. $\square$

Theorem 12 asserts the correctness result for the execution of an initial configuration evaluating, if it converges, to a primitive value. To prove this result, however, we have to deal with the intermediate configurations generated during the evaluation and with function values. To this extent we prove the following lemma, which asserts that configurations related by the translation relation evaluate to values and stores which are related by the translation relation.

Lemma 18. *Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_C \langle se \mid \rho'\rangle$ ($C = se$ or $C = ex$), where $\rho' = \rho^I + \rho^M$ ($dom(\rho^M) = dom(\rho)$). If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, then $\langle se \mid \rho'\rangle \Downarrow_C \langle v' \mid \rho'_*\rangle$, where $\rho'_* = \rho^I_* + \rho^M_*$ ($dom(\rho^M_*) = dom(\rho_*)$)*

(A) $v \cong \langle v' \mid \rho^I_* + \rho^M_*\rangle$,
(B) *for all* $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ *implies* $\rho'(l) = \rho'_*(l)$,
(C) *for all* $l \in dom(\rho_*)$, *we have that* $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho^I_* + \rho^M_*\rangle$ *and*
(D) $dom(\rho^I) \subseteq dom(\rho^I_*)$.

Proof:
We first prove the lemma for $\approx_{sq}$ by induction on the derivation of $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$. In doing the proof we will use also inductive hypotheses on $\approx_{ex}$. Then we prove that if the lemma holds for $\approx_{sq}$ then it also holds for $\approx_{ex}$. So the use of the inductive hypotheses on $\approx_{ex}$ in the proof for $\approx_{sq}$ are justified.

Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{sq} \langle se \mid \rho'\rangle$. From Definition 16.1 we have that there are sequences of locations $\overline{l}^x$ such that $\{\overline{l}^x\} \subseteq dom(\rho^I)$ and

(α) for all i, $1 \leq i \leq n$, let $x_i{:}T_i{\mapsto}v_i \in \sigma$, we have that $v_i \cong \langle \rho'(l_i^x) \mid \rho^I + \rho^M\rangle$,
(β) for all $l \in dom(\rho)$, we have that $\rho(l) \cong \langle \rho'(l) \mid \rho^I + \rho^M\rangle$.

Let $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$. By induction on the derivation of $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$. We only show the most interesting cases which are the base cases, the rule for application, `let mutable` and `let rec` and assignment. The others are similar.
Consider the last rule of Fig. 5 applied in the derivation.

Rule (VAR-F) In this case $e{=}x$ and, from Lemma 17.1, for some i, $x = x_i$, $x_i{\mapsto}v_i \in \sigma$ and $se = l_i^x$. Let $\langle x \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$, where $lkp(x, \sigma) = v$. Applying rule (LOC) of Fig. 11, we have that $\langle se \mid \rho'\rangle \Downarrow_{ex} \langle \rho'(l_i^x) \mid \rho'\rangle$ and from rule (EXPR) we get $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle \rho'(l_i^x) \mid \rho'\rangle$.
If $v = v_i$, i.e., the value is not a recursively defined function, then, from (α) we have that $v \cong (\rho'(l_i^x), \rho')$. Therefore (A) holds.
If $v_i = (\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$ for some k, then

$$v = (F_k, \sigma[x_j{:}T_j{\mapsto}(\texttt{let rec}\ \overline{x}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j, \sigma)]_{1\leq j\leq m}).$$

Since, from (α), $v_i \cong (\rho'(l_i^x), \rho')$, from Lemma 15, we also have that $v \cong (\rho'(l_i^x), \rho')$ and (A) holds.
Since ρ and ρ' are not modified clauses (B), (C) and (D) hold.

Rule (PR-VAL-F) In this case $e{=}n$ or $e{=}\texttt{tr}$ or $e{=}\texttt{fls}$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle e \mid \rho\rangle$, i.e., $v{=}e$. From Lemma 17.2, we have that $se{=}e$. Applying rule (VAL) of Fig. 11, we have that $\langle se \mid \rho'\rangle \Downarrow_{ex} \langle e \mid \rho'\rangle$ and from rule (EXPR) we get $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle e \mid \rho'\rangle$.
From clause 1 of Definition 14 also $e \cong \langle e \mid \rho'\rangle$. Therefore (A) holds.
Since ρ and ρ' are not modified clauses (B) (C) and (D) hold.

Rule (FN-VAL-F) In this case $e{=}\texttt{fun}\ x{:}T\texttt{->}e'$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle (e, \sigma) \mid \rho\rangle$, i.e., $v = (e, \sigma)$. From Lemma 17.3 we have that se is an `IL` value. As before, applying rule (VAL) and (EXPR) of Fig. 11, we get $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle se \mid \rho'\rangle$.
From Lemma 17.3 we also get that $(e, \sigma) \cong \langle se \mid \rho'\rangle$. Therefore (A) holds.
Since ρ and ρ' are not modified clauses (B) (C) and (D) hold.

Rule (LOC-F) In this case $e{=}l$ for some l; so $\langle l \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$ where $v = \rho(l)$. From Lemma 17.4, we have that $l = l_j^y$ for some $l_j^y \in \overline{l}^y$ and $se = l_j^y$, so $v = \rho(l_j^y)$. Applying rule (LOC) of Fig. 11, we have that $\langle se \mid \rho'\rangle \Downarrow_{ex} \langle \rho'(l_j^y) \mid \rho'\rangle$ and from rule (EXPR) we get $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle \rho'(l_j^y) \mid \rho'\rangle$.
From (β) we have that $v \cong (\rho'(l_j^y), \rho')$. Therefore (A) holds.
Since ρ and ρ' are not modified clauses (B) (C) and (D) hold.

Rule (App-F) In this case $e{=}e_1\ e_2$ for some e_1, e_2 and $\langle e_1\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ and $\langle e_b \mid \sigma'[x{:}T_1{\mapsto}v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$.
From Lemma 17.7(c) we have that $se = e_1'\ e_2'$ where $e_i' = se_i^\circ[\overline{x}\,\overline{y}\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_i}]$ and $[\![e_i^\circ]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}} = (\,se_i^\circ, \overline{\delta}^i\,)$ $(1 \le i \le 2)$.
Consider the configurations $\langle e_1 \mid \sigma \mid \rho\rangle$ and $\langle e_1' \mid \rho'\rangle$. From $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ and 17.7(a), we get $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. From the fact that $\langle se \mid \rho'\rangle$ is well-formed, and Definition 16.2, also $\langle e_1' \mid \rho'\rangle$ is well-formed. Moreover, from Lemma 17.7(b) and (c), (α) and (β) we derive that

$$\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e_1' \mid \rho'\rangle.$$

Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, we get

(1) $\langle e_1' \mid \rho'\rangle \Downarrow_{ex} \langle v'' \mid \rho_1'\rangle$

where $\rho_1' = \rho_1^I + \rho_1^M$ $(dom(\rho_1^M) = dom(\rho_1))$

(A_1) $(\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma') \cong \langle v'' \mid \rho_1^I + \rho_1^M\rangle$,

(B_1) for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho_1'(l)$,

(C_1) for all $l \in dom(\rho_1)$, we have that $\rho_1(l) \cong \langle \rho_1'(l) \mid \rho_1^I + \rho_1^M\rangle$ and

(D_1) $dom(\rho^I) \subseteq dom(\rho_1^I)$.

Now consider the configurations $\langle e_2 \mid \sigma \mid \rho_1\rangle$ and $\langle e_2' \mid \rho_1'\rangle$ where $\rho_1' = \rho_1^I + \rho_1^M$.
From Lemma 3 we have that $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$ and from Lemma 17.7(a), $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_2 : T_1$ and since $\overline{l}^y{:}\overline{T}^y \subseteq \Sigma_1$, we get $\Sigma_1 \models \langle e_2 \mid \sigma \mid \rho_1\rangle\diamond$. Therefore, $\models \sigma\diamond$ implies $\Sigma_1 \models \langle e_2 \mid \sigma \mid \rho_1\rangle\diamond$. From Theorem 7 we have that $\langle v_1 \mid \rho_1'\rangle$ is well-formed and, since $Loc(se) \subseteq dom(\rho') \subseteq dom(\rho_1')$, then also $Loc(e_2') \subseteq dom(\rho_1')$. So $\langle e_2' \mid \rho_1'\rangle$ is well-formed.
From Lemma 17.7(b), $\overline{\delta}^2$ is represented by $\overline{l}^{z_2}$ in ρ'. From (B_1), since $\{\overline{l}^{z_2}\} \subseteq dom(\rho^I)$, $\overline{\delta}^2$ is represented by $\overline{l}^{z_2}$ in ρ_1'. Therefore from (B_1) and (α), we have that:

(α') for all i, $1 \le i \le n$, let $x_i{\mapsto}v_i \in \sigma$, $v_i \cong (\rho_1'(l_i^x), \rho_1')$.

From Lemma 17.7(b) and (c), (α') and (C_1) we derive that

$$\Sigma_1, \overline{x}, \overline{y} \models \langle e_2 \mid \sigma \mid \rho_1\rangle \approx_{ex} \langle e_2' \mid \rho_1'\rangle.$$

Applying the inductive hypothesis to $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$, we get

(2) $\langle e_2' \mid \rho_1'\rangle \Downarrow_{ex} \langle v_x' \mid \rho_2'\rangle$

where $\rho_2' = \rho_2^I + \rho_2^M$ $(dom(\rho_2^M) = dom(\rho_2))$

(A_2) $v_x \cong \langle v_x' \mid \rho_2'\rangle$,

(B_2) for all $l \in dom(\rho_1^I)$, $\rho_1'(l) \neq ?$ implies $\rho_2'(l) = \rho_1'(l)$

(C_2) for all l, $l \in dom(\rho_2)$, we have that $\rho_2(l) \cong \langle \rho_2'(l) \mid \rho_2' \rangle$ and

(D_2) $dom(\rho_1^I) \subseteq dom(\rho_2^I)$.

From (A_1) and Definition 14 for some k, $(\texttt{fun}\, x{:}T_1\texttt{->}e_b, \sigma') \cong_k \langle v'[\overline{x}'\,\overline{z} := \overline{l}^{x'}\,\overline{l}^{z}] \mid \rho_1' \rangle$ where v', $\overline{x}'$, $\overline{z}$, $\overline{l}^{x'}$ and $\overline{l}^{z}$ are such that:

(P_1) $\overline{x}' = dom(\sigma')$, $\{\overline{l}^{x'}\} \subseteq dom(\rho_1^I)$,

(P_2) $[\![\texttt{fun}\, x{:}T_1\texttt{->}e_b]\!]_{sq}^{\{\overline{x}'\},\{\overline{y}\}} = (\, v', \overline{\delta}\,)$ for any $\overline{y}$ and so $v'' = \texttt{fun}\, x\texttt{->}\{se^\circ[\overline{x}'\,\overline{z} := \overline{l}^{x'}\,\overline{l}^{z}]\}$ where $[\![e_b]\!]_{bl}^{\{\overline{x}',x\},\emptyset} = (\, \{se^\circ\}, \overline{\delta}\,)$ and $\overline{\delta}$ is represented by $\overline{l}^{z}$ in ρ_1' and

(P_3) for all i, $1 \leq i \leq n$, let $x_i'{:}T_i' {\mapsto} v_i' \in \sigma'$, we have that $v_i' \cong_h \langle \rho_1'(l_i^{x'}) \mid \rho_1' \rangle$, for some $h \leq k$

Let $se' = se^\circ[x\,\overline{x}'\,\overline{w}\,\overline{z} := l_x\,\overline{l}^{x'}\,\overline{l}^{w}\,\overline{l}^{z}]$. Consider the configurations: $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle$ and $\langle se' \mid \rho_2'[l_x \mapsto v_x', \overline{l}^{w} \mapsto \overline{?}] \rangle$ where $\overline{w} = def(se^\circ)$.

From Lemma 17.7(a), $env(\sigma) \mid \overline{l}^{y}{:}\overline{T}^{y} \vdash e_1 : T' \to T$. Since $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle (\texttt{fun}\, x{:}T_1\texttt{->}e_b, \sigma') \mid \rho_1 \rangle$, from Lemma 3 we have $\models (\texttt{fun}\, x{:}T_1\texttt{->}e_b, \sigma'){:}T_1 \to T$ and from Definition 2(c) and rule <span style="font-variant:small-caps">(TyAbs)</span> (which takes into account also the location environment) preceding Definition 2 we derive that $env(\sigma')[x{:}T_1] \mid \emptyset \vdash e_b : T$ and $\models \sigma'\diamond$. Moreover, applying Lemma 3 to $\langle e_2 \mid \sigma \mid \rho_1 \rangle \Downarrow \langle v_x \mid \rho_2 \rangle$, we have that, $\models v_x{:}T_1$ and for some $\Sigma_2 \supseteq \Sigma_1$, $\Sigma_2 \models \rho_2$. From $\models \sigma'\diamond$ and $\models v_x{:}T_1$ we get $\models \sigma'[x{:}T_1 \mapsto v_x]\diamond$. So $\Sigma_2 \models \langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle\diamond$.

From $\langle e_1' \mid \rho' \rangle \Downarrow_{ex} \langle v'' \mid \rho_1' \rangle$ and Theorem 7, $\langle v'' \mid \rho_1' \rangle$ is well-formed. From (P_1) and (P_2), $\{\overline{x}', \overline{z}\} \subseteq Loc(\rho_1') \subseteq Loc(\rho_2')$, therefore $\{\overline{x}, \overline{z}, l_x, \overline{l}^{w}\} \subseteq Loc(\rho_2'[l_x \mapsto v_x', \overline{l}^{w} \mapsto \overline{?}])$, so $\langle se' \mid \rho_2'[l_x \mapsto v_x', \overline{l}^{w} \mapsto \overline{?}] \rangle$ is well-formed.

From (P_3) and (D_2), since $\{\overline{l}^{x'}\} \subseteq dom(\rho_1^I)$, we have

(P_3') for all i, $1 \leq i \leq n$, let $x_i'{:}T_i' {\mapsto} v_i' \in \sigma'$, we have that $v_i' \cong \langle \rho_2'(l_i^{x'}) \mid \rho_2' \rangle$

Therefore from (C_2), (P_2) and (P_3) we derive that:

$$\Sigma_2, \overline{x}'\,x, \emptyset \models \langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle \approx_{se} \langle se' \mid \rho_2'[l_x \mapsto v_x', \overline{l}^{w} \mapsto \overline{?}] \rangle.$$

Applying the induction hypothesis to $\langle e_b \mid \sigma'[x{:}T' {\mapsto} v_x] \mid \rho_2 \rangle \Downarrow \langle v \mid \rho_* \rangle$ we get $\langle se' \mid \rho_2'[l_x \mapsto v_x', \overline{l}^{w} \mapsto \overline{?}] \rangle \Downarrow_{sq} \langle v' \mid \rho_*' \rangle$, where $\rho_*' = \rho_*^I + \rho_*^M$ $(dom(\rho_*^M) = dom(\rho_*))$

(A_3) $v \cong \langle v' \mid \rho_*' \rangle$,

(B_3) for all $l \in dom(\rho_2^I)$, $\rho_2'(l) \neq ?$ implies $\rho_*'(l) = \rho_2'(l)$

(C_3) for all l, $l \in dom(\rho_*)$, we have that $\rho_*(l) \cong \langle \rho_*'(l) \mid \rho_*' \rangle$ and

(D_3) $dom(\rho_2^I) \subseteq dom(\rho_*^I)$.

Applying rule (Block) of Fig. 11 we have

$(\mathbf{3})$ $\langle \{ se^\circ[x\,\overline{x}'\,\overline{z} := l_x\,\overline{l}^{x'}\,\overline{l}^{z}] \} \mid \rho'_2[l_x \mapsto v'_x] \rangle \Downarrow_{bl} \langle v' \mid \rho'_3 \rangle$

From $(\mathbf{1})$, $(\mathbf{2})$ and $(\mathbf{3})$ and $v'' = \texttt{fun}\ x\text{->}\{se^\circ[\overline{x}'\,\overline{z} := \overline{l}^{x'}\,\overline{l}^{z}]\}$ applying rule (App) of Fig. 11 we have $\langle e'_1\ e'_2 \mid \rho' \rangle \Downarrow_{ex} \langle v' \mid \rho'_* \rangle$ and from rule (Exp)

$$(\star) \qquad \langle e'_1\ e'_2 \mid \rho' \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$$

From (B_1), (B_2), (B_3), (D_1), (D_2) and (D_3) we derive

(B') for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'_*(l) = \rho'(l)$

and from (D_1), (D_2), (D_3) and transitivity of $\subseteq$

(D') $dom(\rho^I) \subseteq dom(\rho^I_*)$.

So $(\star)$, (A_3), (B'), (C_3) and (D') prove the result.

Rule (Let-Mut-F) In this case $e{=}\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2$. Assume that $y \notin \{\overline{x}, \overline{y}\}$, $\langle \texttt{let mutable}\ y{:}T'{=}e_1\ \texttt{in}\ e_2 \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$ where $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v_y \mid \rho_1 \rangle$ and $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle \Downarrow \langle v \mid \rho_* \rangle$ for $l_y \notin dom(\rho_1)$.
From Lemma 17.10(c) we have that $se{=}\texttt{def}\ y{=}e'_1; se_2$. Moreover, from Lemma 17.10(a) $e^\circ{=}\texttt{let mutable}\ y{:}T{=}e^\circ_1\ \texttt{in}\ e^\circ_2$ where $e_i = e^\circ_i[\overline{y} := \overline{l}^{y}]$ $(1 \leq i \leq 2)$.

From Lemma 17.10(c).i

$$\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho \rangle \approx_{ex} \langle e'_1 \mid \rho' \rangle.$$

Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v_x \mid \rho_1 \rangle$, we get $\langle e'_1 \mid \rho' \rangle \Downarrow_{sq} \langle v'_x \mid \rho'_1 \rangle$, where $\rho'_1 = \rho^I_1 + \rho^M_1$ $(dom(\rho^M_1) = dom(\rho_1))$

(A_1) $v_1 \cong \langle v'_x \mid \rho'_1 \rangle$,

(B_1) for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_1(l)$,

(C_1) for all $l \in dom(\rho_1)$, we have that $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho'_1 \rangle$ and

(D_1) $dom(\rho^I) \subseteq dom(\rho^I_1)$.

Consider now the configurations $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle$ and $\langle se_2 \mid \rho'_1[l_y \mapsto v'_y] \rangle$, where $\rho'_1[l_y \mapsto v'_y] = \rho^I_1 + \rho^M_1$ $(dom(\rho^M_1) = dom(\rho_1[l_y \mapsto v_y]))$. Note that $l_y \in dom(\rho^M_1)$.

From $\Sigma \models \langle e_1 \mid \sigma \mid \rho \rangle \diamond$ and Lemma 3 we have that $\Sigma' \models \rho_1$ for some $\Sigma' \supseteq \Sigma$, $\models \sigma \diamond$ and, since from Lemma 17.10(a), $env(\sigma) \mid \overline{l}^{y}{:}\overline{T}^{y} \vdash e_1 : T'$, we also have $\models v_y{:}T'$. Let $\Sigma_1 = \Sigma', l_y{:}T$, we have that $\Sigma_1 \models \rho_1[l_y \mapsto v_y]$. From Lemma 17.10(a), we also have $env(\sigma) \mid \overline{l}^{y}{:}\overline{T}^{y}, l_y{:}T \vdash e_2 : T$. So, $\Sigma_1 \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle \diamond$.
From Lemma 17.10.(c).ii and (b), $se_2 = se^\circ_2[\overline{x}\,y\,\overline{y}\,\overline{w}\,\overline{z}^2 := \overline{l}^{x}\,l_y\,\overline{l}^{y}\,\overline{l}^{w}\,\overline{l}^{z_2}]$ where $[\![e^\circ_2]\!]^{\{x\,\overline{x}\},\{\overline{y}\}}_{sq} = (\,se^\circ_2, \overline{\delta}^2\,)$, $def(\overline{\delta}^i) = \{\overline{z}^i\}$, $\overline{w} = def(se^\circ_2)$, such that $\overline{\delta}^2$ is represented by $\overline{l}^{z_2}$ in ρ' and $\rho'(\overline{l}^{w}) = \overline{?}$.
From Theorem 7, $\langle v'_y \mid \rho'_1 \rangle$ is well-formed and from $Loc(se_2) \subseteq dom(\rho') \subseteq dom(\rho'_1)$ we get that $\langle se_2 \mid \rho'_1[l_y \mapsto v'_y] \rangle$ is well-formed.

From Lemma 17.10(c).i $\{\overline{l}^{w}\} \cap Loc(e'_1) = \emptyset$, so since $\rho'(\overline{l}^{w}) = \overline{?}$, also $\rho'_1[l_x \mapsto v'_x](\overline{l}^{w}) = \overline{?}$. Therefore, from Lemma 17.10.(b) and (c).ii we have that

$$\Sigma_1, \overline{x}, y\,\overline{y} \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \approx_{se} \langle se_2 \mid \rho'_1[l_y \mapsto v'_y]\rangle.$$

Applying the inductive hypothesis to $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$ we get $\langle se_2 \mid \rho'_1[l_y \mapsto v'_y]\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ where $\rho'_* = \rho^I_* + \rho^M_*$ $(dom(\rho^M_*) = dom(\rho_*))$

(A_2) $v \cong \langle v' \mid \rho'_*\rangle$

(B_2) for all $l \in dom(\rho^I_1)$, $\rho'_1[l_y \mapsto v'_y](l) \neq ?$ implies $\rho'_*(l) = \rho'_1[l_y \mapsto v'_y](l)$ (note that $l_y \notin dom(\rho^I_1)$)

(C_2) for all l, $l \in dom(\rho_*)$, we have that $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$ and

(D_2) $dom(\rho^I_1) \subseteq dom(\rho^I_*)$.

From $\langle e'_1 \mid \rho'\rangle \Downarrow_{ex} \langle v'_y \mid \rho'_1\rangle$ applying rule (Def) of Fig. 5, for $l_y \notin dom\rho'_1$ we derive: $\langle \texttt{def}\ l_y{=}e'_1 \mid \rho'\rangle \Downarrow_{st} \langle v'_y \mid \rho'_1[l_y \mapsto v'_y]\rangle$, and from $\langle se_2 \mid \rho'_1[l_y \mapsto v'_y]\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$ applying rule (Seq) of Fig. 5 we get

$$(\star) \quad \langle \texttt{def}\ l_y{=}e'_1; se_2 \mid \rho'\rangle \Downarrow_{st}\langle v' \mid \rho'_*\rangle$$

From (B_1), (B_2) and (D_1), we derive that

(B') for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'_*(l) = \rho'(l)$

and from (D_1), (D_2) and transitivity of $\subseteq$

(D') $dom(\rho^I) \subseteq dom(\rho^I_*)$

Therefore,$(\star)$, (A_2), (B'), (C_2) and (D') prove the result.

Rule (LetRec-F) In this case $e{=}\texttt{let rec}\ \overline{w}{=}\overline{F}\ \texttt{in}\ e_1$ and $e \Downarrow \langle v \mid \rho_1\rangle$ where

$$\langle e_1 \mid \sigma[w_j{:}T_j {\mapsto} (\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j, \sigma)]_{1\leq j\leq m} \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$$

From Lemma 17.11(c), $se = \texttt{def}\ \overline{l}^{w}{=}\overline{e}''; se_1$. Moreover, from Lemma 17.11(a) and (b) we have that $e^\circ{=}\texttt{let rec}\ \overline{w}{=}\overline{F}\ \texttt{in}\ e^\circ_1$ where $[\![e^\circ_1]\!]^{\{\overline{x},\overline{w}\},\{\overline{y}\}}_{sq} = (\,se^\circ_1, \overline{\delta}\,)$ with $def(\overline{\delta}) = \{\overline{z}\}$ and $\overline{\delta}$ is represented by $\overline{l}^{z}$ in ρ' and $[\![F_j]\!]^{\{\overline{w}\,\overline{x}\},\{\overline{y}\}}_{sq} = (\,e'_j, \overline{\delta}^{j}\,)$ with $def(\overline{\delta}^{j}) = \{\overline{z}^{j}\}$ such that $\overline{\delta}^{j}$ is represented by $\overline{l}^{z_j}$ in ρ' $(1 \leq j \leq m)$.
Consider the configurations $\langle e_1 \mid \sigma[w_k{:}T_k{\mapsto}v^R_k]_{1\leq k\leq m} \mid \rho\rangle$ where $v^R_k = (\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$ and $\langle se_1 \mid \rho'[\overline{l}^{w} \mapsto \overline{e}'']\rangle$ where $e''_j = e'_i[\overline{w}\,\overline{x}\,\overline{z}^{j} := \overline{l}^{w}\,\overline{l}^{x}\,\overline{l}^{z_j}]$ $(1 \leq j \leq m)$.
From Lemma 17.11(a), $env(\sigma), \overline{w}{:}\overline{T} \mid \overline{l}^{y}{:}\overline{T}^{y} \vdash e_1 : T$ and from $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$, we have $\Sigma \models \rho$ and $\models \sigma\diamond$. From Lemma 17.11(a), for all j, $1 \leq j \leq m$ we have $env(\sigma), \overline{w}{:}\overline{T} \mid \emptyset \vdash F_j : T_j$ and from rule (TyRec) of Fig. 3, for all k, $1 \leq k \leq m$, we derive $env(\sigma), \overline{w}{:}\overline{T} \mid \emptyset \vdash \texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k : T_k$. Therefore, $\models v^R_k{:}T_k$, which proves that

$$\Sigma \models \langle e_1 \mid \sigma[w_j{:}T_j{\mapsto}v^R_j]_{1\leq j\leq m} \mid \rho\rangle\diamond$$

From Lemma 17.11(c).ii $Loc(se_1) \subseteq dom(\rho'[\overline{l}^w \mapsto \overline{e}''])$, therefore since $\langle se \mid \rho' \rangle$ is well-formed, also $\langle se_1 \mid \rho'[\overline{l}^w \mapsto \overline{e}''] \rangle$ is well-formed.
From $[\![F_j]\!]_{sq}^{\{\overline{w},\overline{x}\},\{\overline{y}\}} = (\,e'_j, \overline{\delta}^j\,)$, $e''_j = e'_j[\overline{w}\,\overline{x}\,\overline{z}^i := \overline{l}^w\,\overline{l}^x\,\overline{l}^{z_j}]$ and (α), for all k, $1 \leq k \leq m$, $v_k^R \cong (e''_k, \rho'[\overline{l}^w \mapsto \overline{e}''])$. Therefore from 17.11(c).ii we derive

$$\Sigma, \overline{x}\,\overline{w}, \overline{y} \models \langle e_1 \mid \sigma[w_k{:}T_k {\mapsto} v_k^R]_{1\leq k\leq m} \mid \rho \rangle \approx_{se} \langle se_1 \mid \rho'[\overline{l}^w \mapsto \overline{e}''] \rangle.$$

Applying the inductive hypothesis to $\langle e_1 \mid \sigma[w_k{:}T_k {\mapsto} v_k^R]_{1\leq k\leq m} \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$, we have that

$$(\star) \qquad \langle se_1 \mid \rho'[\overline{l}^w \mapsto \overline{e}''] \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$$

where $\rho'_* = \rho_*^I + \rho_*^M$ $(dom(\rho_*^M) = dom(\rho_*))$

(A_1) $v \cong \langle v' \mid \rho'_* \rangle$,

(B_1) for all $l \in dom(\rho^I) \cup \{\overline{l}^w\}$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_*(l)$,

(C_1) for all $l \in dom(\rho_*)$, we have that $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_* \rangle$ and

(D_1) $dom(\rho^I \cup \{\overline{l}^w\}) \subseteq dom(\rho_*^I)$.

From (B_1) and (D_1) we also have that

(B') for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_*(l)$ and

(D') $dom(\rho^I) \subseteq dom(\rho_*^I)$.

Therefore, $(\star)$, (A_1), (B'),(C_1) and (D') prove the result.

Rule (Ass-F) In this case $e{=}l{<\!\!-}e_1$ and $\langle l{<\!\!-}e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$, where $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_1 \rangle$ and $\rho_* = \rho_1[l \mapsto v]$.
From Lemma 17.12(b) we have that $se = l{<\!\!-}e'_1$ where e'_1 is such that $\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho \rangle \approx_{ex} \langle e'_1 \mid \rho' \rangle$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_1 \rangle$, we get $\langle e'_1 \mid \rho' \rangle \Downarrow_{ex} \langle v' \mid \rho'_1 \rangle$, where $\rho'_1 = \rho_1^I + \rho_1^M$ $(dom(\rho_1^M) = dom(\rho_1))$

(A_1) $v \cong \langle v' \mid \rho_1^I + \rho_1^M \rangle$

(B_1) for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_1(l)$,

(C_1) for all $l \in dom(\rho_1)$, we have that $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho_1^I + \rho_1^M \rangle$ and

(D_1) $dom(\rho^I) \subseteq dom(\rho_1^I)$.

From Lemma 17.12(b), we get that $l \in dom(\rho_1^M)$. From rule (Ass) of Fig. 11 we derive $\langle l{<\!\!-}e_1 \mid \rho' \rangle \Downarrow_{st} \langle v' \mid \rho_* \rangle$ and from rule (St) also $\langle l{<\!\!-}e_1 \mid \rho' \rangle \Downarrow_{sq} \langle v' \mid \rho_* \rangle$. From (A_1), we derive that $(A_1) \div (D_1)$ hold also replacing ρ_1 with ρ_* and ρ'_1 with ρ'_*, which proves the result.

We now prove the statement of the Lemma where $C = ex$, assuming it for $C = se$. Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho \rangle \approx_{ex} \langle e' \mid \rho'_e \rangle$. From Definition 16.2 there are se and ρ' such that $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \rho \rangle \approx_{sq} \langle se \mid \rho' \rangle$.

- Assume first that clause (a) of Definition 16.2 holds. Then $e' = se$ and $\rho'_e = \rho'$. If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, then $\langle e' \mid \rho'_e\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ and $(A) \div (D)$ hold. Looking at the rules of Fig. 11 the only rule that could have been applied to derive $\langle e' \mid \rho'_e\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ is (Exp). Therefore, $\langle e' \mid \rho'_e\rangle \Downarrow_{ex} \langle v' \mid \rho'_*\rangle$, which proves the result.

- If, instead, clause (b) of Definition 16.2 holds, let $[\![e^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ, \overline{\delta}\,)$ where $e = e^\circ[\overline{y} := \overline{l}^y]$, $e' = \mathtt{exc}(l', \overline{Y} \mapsto \overline{l}^y, \overline{l}^x)$, where $l' \notin dom(\rho')$, ρ'_e is such that
  - $\rho'_e(l') = \mathtt{code}(\{se^\circ\}[\overline{z} := \overline{l}^z], \overline{y} \mapsto \overline{Y}, \overline{x})$ $(l' \notin dom(\rho'))$ and
  - for all $l \in dom(\rho) \cup \{\overline{l}^x, \overline{l}^z\}$ we have that $\rho'_e(l) = \rho'(l)$.

  From rule (Loc) of Fig. 5, we have

  $(\mathbf{1})$ $\langle l \mid \rho'_e\rangle \Downarrow_{ex} \langle \mathtt{code}(\{se^\circ\}[\overline{z} := \overline{l}^z], \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \rho'_e\rangle$ and

  $(\mathbf{2})$ $\langle \overline{l}^x \mid \rho'_e\rangle \Downarrow_{ex} \langle \overline{v}' \mid \rho'_e\rangle$ where $v'_i = \rho'(l^x_i)$ $(1 \leq i \leq n)$.

  Let $se' = se^\circ[\overline{x}\,\overline{y}\,\overline{w}\,\overline{z} := \overline{l}'\,\overline{l}^y\,\overline{l}^w\,\overline{l}^z]$ where $\{\overline{l}'\} \cap dom(\rho'_e) = \emptyset$. Define $\rho'_1 = \rho'_e[\overline{l}' \mapsto \rho'(\overline{l}^x)]$ and $\rho'_2 = \rho'_1[\overline{l}^w \mapsto \overline{?}]$. Since $\langle se \mid \rho'\rangle$ is well-formed, then $\langle se' \mid \rho'_2\rangle$ is well-formed. From definition of ρ'_2, for all $l \in dom(\rho) \cup \{\overline{l}^x, \overline{l}^z\}$, $\rho'_2(l) = \rho'(l)$. Therefore, from Definition 16.1(d), we get that: for all i, $1 \leq i \leq n$, let $x_i{\mapsto}v_i \in \sigma$, $v_i \cong (\rho'_2(l'_i), \rho'_2)$, and from Definition 16.1(e), for all $l \in dom(\rho)$, $\rho(l) \cong \langle \rho'_2(l) \mid \rho'_2\rangle$ that implies

  $$\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se' \mid \rho'_2\rangle.$$

  Let $\rho'_2 = \rho^I_2 + \rho^M_2$ $(dom(\rho^M_2) = dom(\rho))$. If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, then $\langle se' \mid \rho'_2\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$, where $\rho'_* = \rho^I_* + \rho^M_*$ $(dom(\rho^M_*) = dom(\rho_*))$

  (A_1) $v \cong \langle v' \mid \rho'_*\rangle$

  (B_1) for all $l \in dom(\rho^I_2)$, $\rho'_2(l) \neq ?$ implies $\rho'_2(l) = \rho'_*(l)$,

  (C_1) for all $l \in dom(\rho_*)$, we have that $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$ and

  (D_1) $dom(\rho^I_2) \subseteq dom(\rho^I_*)$.

  From $\langle se' \mid \rho'_2\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$, applying rule (Block) of Fig. 5, we have

  $(\mathbf{3})$ $\langle \{se'\} \mid \rho'_1\rangle \Downarrow_{bl} \langle v' \mid \rho'_*\rangle$

  Therefore, from $(\mathbf{1})$, $(\mathbf{2})$ and $(\mathbf{3})$, applying rule (Code) of Fig. 5, we derive that $\langle e' \mid \rho'_e\rangle \Downarrow_{ex} \langle v' \mid \rho'_*\rangle$.
  From (B_1), (D_1) and $\rho^I \subset \rho^I_2$, we have that

  (B') for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_*(l)$,

  (D') $dom(\rho^I) \subseteq dom(\rho^I_*)$.

  Therefore $\langle e' \mid \rho'_e\rangle \Downarrow_{ex} \langle v' \mid \rho'_*\rangle$, (A_1), (B'), (C_1), (D') prove the result. □

The following lemma asserts that if the evaluation of an `F#` expression does not converge to a value, then also the evaluation of a translation related `IL` sequence of statement or expressions does not converge to a value.

Lemma 19. *Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_C \langle se \mid \rho'\rangle$ ($C = se$ or $C = ex$), where $\rho' = \rho^I + \rho^M$ ($dom(\rho^M) = dom(\rho)$). If $\langle se \mid \rho'\rangle \Downarrow_C \langle v' \mid \rho'_*\rangle$, then $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, for some v and ρ_*.*

Proof:
By case analysis on the shape of e, and then induction on the derivation of the judgment $\langle se \mid \rho'\rangle \Downarrow_C \langle v' \mid \rho'_*\rangle$.
We first prove the lemma for $C = se$.
If $e = x$, from $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se \mid \rho'\rangle$ and Lemma 17.1, $x = x_i$ and $se = l_i^x$, so $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle \rho'(l_i^x) \mid \rho'\rangle$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle lkp(x, \sigma) \mid \rho\rangle$.
If $e{=}n$, or $e{=}\texttt{tr}$, or $e{=}\texttt{fls}$, or $e{=}\texttt{fun}\ x{:}T'\texttt{->}e_1$, then we have that $se = e$, $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle se \mid \rho'\rangle$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$ for some v.
If $e = l$, from $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se \mid \rho'\rangle$ and Lemma 17.4, $l = l_j^y$ and $se = l_j^y$ and again $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle \rho'(l) \mid \rho'\rangle$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \rho(l) \mid \rho\rangle$.

For the structured expressions, we only show the proof for application and let recursive and mutable. The others are similar.

$\underline{e{=}e_1\ e_2}$ for some e_1, e_2. Assume that $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se \mid \rho'\rangle$. From Lemma 17.7(c) we have that $e_1\ e_2 = (e_1^\circ[\overline{y} := \overline{l}^y])\ (e_2^\circ[\overline{y} := \overline{l}^y])$ for some e_i° and $se = e'_1\ e'_2$ where $e'_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{z}^i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{z_i}]$ and $[\![e_i^\circ]\!]_{ex}^{\{\overline{x}\},\{\overline{y}\}} = (\,se_i^\circ, \overline{\delta}_i\,)$ $(1 \leq i \leq 2)$. Moreover, from Lemma 17.7(a), $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T' \to T$.
Consider the configurations $\langle e_1 \mid \sigma \mid \rho\rangle$ and $\langle e'_1 \mid \rho'\rangle$. From $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ and Lemma 17.7(a), we get $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. From the fact that $\langle se \mid \rho'\rangle$ is well-formed, and Definition 16.2, also $\langle e'_1 \mid \rho'\rangle$ is well-formed. Moreover, from $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se \mid \rho'\rangle$, we have

- for all i, $1 \leq i \leq n$, let $x_i{:}T_i{\mapsto}v_i \in \sigma$, we have that $v_i \cong \langle \rho'(l_i^x) \mid \rho^I{+}\rho^M\rangle$ and
- for all $l \in dom(\rho)$, we have that $\rho(l) \cong \langle \rho'(l) \mid \rho^I + \rho^M\rangle$.

Therefore
$$\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho\rangle \approx_{ex} \langle e'_1 \mid \rho'\rangle.$$

From $\langle e'_1\ e'_2 \mid \rho'\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ and rule (Exp) of Fig. 11 we have that $\langle e'_1\ e'_2 \mid \rho'\rangle \Downarrow_{ex} \langle v' \mid \rho'_*\rangle$. From rule (App) of Fig. 11, we have that $\langle e'_1 \mid \rho'\rangle \Downarrow_{ex} \langle \texttt{fun}\ x\texttt{->}\{se'\} \mid \rho'_1\rangle$ for some e' and ρ'_1. Applying the inductive hypotheses we derive $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$ for some v_1 and ρ_1. From Lemma 3, we get $\models v_1{:}T' \to T$, $v_1 = (\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma')$ for some e_b and σ' and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$.
From Lemma 18, let $\rho'_1 = \rho_1^I + \rho_1^M$ where $dom(\rho_1^M) = dom(\rho_1)$, we have that

(A_1) $(\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma') \cong \langle \texttt{fun}\ x\texttt{->}\{se'\} \mid \rho_1^I + \rho_1^M\rangle$,

(B_1) for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_1(l)$,

(C_1) for all $l \in dom(\rho_1)$, we have that $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho^I_1 + \rho^M_1 \rangle$ and

(D_1) $dom(\rho^I) \subseteq dom(\rho^I_1)$.

Considering the configurations $\langle e_2 \mid \sigma \mid \rho_1 \rangle$ and $\langle e'_2 \mid \rho'_1 \rangle$ where $\rho'_1 = \rho^I_1 + \rho^M_1$, we can prove, as for the case of Rule (App-F) of Lemma 18, that

$$\Sigma_1, \overline{x}, \overline{y} \models \langle e_2 \mid \sigma \mid \rho_1 \rangle \approx_{ex} \langle e'_2 \mid \rho'_1 \rangle.$$

From $\langle e'_1\ e'_2 \mid \rho' \rangle \Downarrow_{ex} \langle v' \mid \rho'' \rangle$ and rule (App) of Fig. 11, we have that $\langle e'_2 \mid \rho'_1 \rangle \Downarrow_{ex} \langle v'_x \mid \rho'_2 \rangle$. Applying the inductive hypotheses we derive $\langle e_2 \mid \sigma \mid \rho_1 \rangle \Downarrow \langle v_x \mid \rho_2 \rangle$ for some v_x and ρ_2. From Lemma 18, let $\rho'_2 = \rho^I_2 + \rho^M_2$ where $dom(\rho^M_2) = dom(\rho_2)$, we have that

(A_2) $v_x \cong \langle v'_x \mid \rho'_2 \rangle$,

(B_2) for all $l \in dom(\rho^I_1)$, $\rho'_1(l) \neq ?$ implies $\rho'_2(l) = \rho'_1(l)$

(C_2) for all l, $l \in dom(\rho_2)$, we have that $\rho_2(l) \cong \langle \rho'_2(l) \mid \rho'_2 \rangle$ and

(D_2) $dom(\rho^I_1) \subseteq dom(\rho^I_2)$.

From (A_1) and Definition 14 for some k, $(\texttt{fun}\ x{:}T'\text{->}e_b, \sigma') \cong_k \langle v''[\overline{x}'\,\overline{z} := \overline{l}^{x'}\,\overline{l}^{z}] \mid \rho'_1 \rangle$ where v'', $\overline{x}'$, $\overline{z}$, $\overline{l}^{x'}$ and $\overline{l}^{z}$ are such that:

(P_1) $\overline{x}' = dom(\sigma')$, $\{\overline{l}^{x'}\} \subseteq dom(\rho^I_1)$,

(P_2) $[\![\texttt{fun}\ x{:}T'\text{->}e_b]\!]^{\{\overline{x}'\},\{\overline{y}\}}_{sq} = (\, v'', \overline{\delta}\,)$ for any $\overline{y}$ and so $\texttt{fun}\ x\text{->}\{se'\} = \texttt{fun}\ x\text{->}\{se^{\circ}[\overline{x}'\,\overline{z} := \overline{l}^{x'}\,\overline{l}^{z}]\}$ where $[\![e_b]\!]^{\{\overline{x}',x\},\emptyset}_{bl} = (\,\{se^{\circ}\}, \overline{\delta}\,)$ and $\overline{\delta}$ is represented by $\overline{l}^{z}$ in ρ'_1 and

(P_3) for all i, $1 \leq i \leq n$, let $x'_i{:}T'_i{\mapsto}v'_i \in \sigma'$, we have that $v'_i \cong_h \langle \rho'_1(l^{x'}_i) \mid \rho'_1 \rangle$, for some $h \leq k$

Consider the configurations: $\langle e_b \mid \sigma'[x{:}T' \mapsto v_x] \mid \rho_2 \rangle$ and $\langle se'[x\,\overline{w} := l_x\,\overline{l}^{w}] \mid \rho'_2[l_x \mapsto v'_x, \overline{l}^{w} \mapsto \overline{?}] \rangle$ where $\overline{w} = def(se^{\circ})$. Again, as for the case of Rule (App-F) of Lemma 18, we can prove that

$$\Sigma_2, \overline{x}'\,x, \emptyset \models \langle e_b \mid \sigma'[x{:}T'{\mapsto}v_x] \mid \rho_2 \rangle {\approx_{se}} \langle se'[x\,\overline{w} := l_x\,\overline{l}^{w}] \mid \rho'_2[l_x{\mapsto}v'_x, \overline{l}^{w}{\mapsto}\overline{?}] \rangle.$$

Again from $\langle e'_1\ e'_2 \mid \rho' \rangle \Downarrow_{ex} \langle v' \mid \rho'' \rangle$ and rule (App) of Fig. 11, we have that $\langle \{se'[x := l_x]\} \mid \rho'_2[l_x \mapsto v'_x] \rangle \Downarrow_{bl} \langle v' \mid \rho'_* \rangle$, and so, from rule (Block) of Fig. 11, $\langle se'[x\,\overline{w} := l_x\,\overline{l}^{w}] \mid \rho'_2[l_x \mapsto v'_x, \overline{l}^{w} \mapsto \overline{?}] \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$. Applying the inductive hypotheses, we get that $\langle e_b \mid \sigma'[x{:}T_1{\mapsto}v_x] \mid \rho_2 \rangle \Downarrow \langle v \mid \rho_* \rangle$ for some v and ρ_*, which concludes the proof.

<u>$e{=}\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2$</u> for some e_1 and e_2. Assume that $y \notin \{\overline{x}, \overline{y}\}$ and $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho \rangle \approx_{se} \langle se \mid \rho'' \rangle$. From Lemma 17.10(c) we have that $se{=}\texttt{def}\ l_y{=}e'_1; se_2$. where

$$\Sigma, \overline{x}, \overline{y} \models \langle e_1 \mid \sigma \mid \rho \rangle \approx_{ex} \langle e'_1 \mid \rho' \rangle.$$

From $\langle \texttt{def}\ l_y{=}e'_1; se_2 \mid \rho'\rangle \Downarrow_{sq} \langle v' \mid \rho'*\rangle$ and rule (Seq) of Fig. 11, we have $\langle \texttt{def}\ l_y{=}e'_1 \mid \rho'\rangle \Downarrow_{sq}\langle v'_y \mid \rho'_2\rangle$, and, $\langle se_2 \mid \rho'_2\rangle \Downarrow_{sq}\langle se_2 \mid \rho'_*\rangle$. From $\langle \texttt{def}\ l_y{=}e'_1 \mid \rho'\rangle \Downarrow_{sq}\langle v'_y \mid \rho'_2\rangle$ and rule (St), we get $\langle \texttt{def}\ l_y{=}e'_1 \mid \rho'\rangle \Downarrow_{st}\langle v'_y \mid \rho'_2\rangle$ and from rule (Def), $\langle e'_1 \mid \rho'\rangle \Downarrow_{ex}\langle v'_y \mid \rho'_1\rangle$ where $\rho'_2 = \rho'_1[l_y \mapsto v'_y]$. Applying the inductive hypotheses to $\langle e'_1 \mid \rho'\rangle \Downarrow_{ex}\langle v'_y \mid \rho'_1\rangle$ we derive $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$. From Lemma 18, let $\rho'_1 = \rho^I_1 + \rho^M_1$ where $dom(\rho^M_1) = dom(\rho_1)$, we have that

(A_1) $v_y \cong \langle v'_y \mid \rho^I_1 + \rho^M_1\rangle$,

(B_1) for all $l \in dom(\rho^I)$, $\rho'(l) \neq ?$ implies $\rho'(l) = \rho'_1(l)$,

(C_1) for all $l \in dom(\rho_1)$, we have that $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho^I_1 + \rho^M_1\rangle$ and

(D_1) $dom(\rho^I) \subseteq dom(\rho^I_1)$.

Consider now the configurations $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle$ and $\langle se_2 \mid \rho'_2\rangle$, where $\rho'_2 = \rho^I_2 + \rho^M_2$ ($dom(\rho^M_2) = dom(\rho_1[l_y \mapsto v_y])$). As for the case of Rule (Let-Mut-F) of Lemma 18, we can prove that

$$\Sigma_1, \overline{x}, y\,\overline{y} \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \approx_{se} \langle se_2 \mid \rho'_2\rangle.$$

Applying the inductive hypotheses to $\langle se_2 \mid \rho'_2\rangle \Downarrow_{sq}\langle se_2 \mid \rho'_*\rangle$ we derive that $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$. Therefore, from rule (LetMut-F) of Fig. 5, $\langle \texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_*, which proves the result.

$\underline{e{=}\texttt{let rec}\ \overline{w}{=}\overline{F}\ \texttt{in}\ e_1}$ for some e_1, $\overline{w}$ and $\overline{F}$. Assume that $\{\overline{w}\} \cap \{\overline{x}, \overline{y}\} = \emptyset$ and $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho\rangle \approx_{se} \langle se \mid \rho''\rangle$. From Lemma 17.11(c) we have that $se = \texttt{def}\ \overline{l}^{w}{=}\overline{e}''; se_1$ where

1. $[\![F_i]\!]^{\{\overline{w}\,\overline{x}\},\{\overline{y}\}}_{sq} = (\,e'_i, \overline{\delta}_i\,)$ and $e''_i = e'_i[\overline{w}\,\overline{x}\,\overline{z}^i := \overline{l}^{w}\,\overline{l}^{x}\,\overline{l}^{z_i}]$
2. $se_1 = se^{\circ}_1[\overline{w}\,\overline{x}\,\overline{w}'\,\overline{z} := \overline{l}^{w}\,\overline{l}^{x}\,\overline{l}^{w'}\,\overline{l}^{z}]$ where $\overline{w}' = def(se^{\circ}_1)$, and $[\![e^{\circ}_1]\!]^{\{\overline{x},\overline{w}\},\{\overline{y}\}}_{sq} = (\,se^{\circ}_1, \overline{\delta}\,)$.

Let $v^R_k = (\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$ $(1 \leq k \leq m)$. Consider the configurations $\langle e_1 \mid \sigma[w_k{:}T_k{\mapsto}v^R_k]_{1\leq k\leq m} \mid \rho\rangle$ and $\langle se_1 \mid \rho'[\overline{l}^{w} \mapsto \overline{e}'']\rangle$. As for the case of Rule (LetRec-F) of Lemma 19, we can prove that

$$\Sigma, \overline{x}\,\overline{w}, \overline{y} \models \langle e_1 \mid \sigma[w_k{:}T_k{\mapsto}v^R_k]_{1\leq k\leq m} \mid \rho\rangle \approx_{se} \langle se_1 \mid \rho'[\overline{l}^{w} \mapsto \overline{e}'']\rangle.$$

Since the expressions $\overline{e}''$ are function definitions, and their evaluation does not modify the store, from $\langle \texttt{def}\ \overline{l}^{w}{=}\overline{e}''; se_1 \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$, rules (Seq), (St), (Def) and (Val) of Fig. 11, we have $\langle \texttt{def}\ \overline{l}^{w}{=}\overline{e}'' \mid \rho'\rangle \Downarrow_{sq}\langle e''_m \mid \rho'[\overline{l}^{w} \mapsto \overline{e}'']\rangle$ and $\langle se_1 \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$. Applying the inductive hypotheses to $\langle se_2 \mid \rho'[\overline{l}^{w} \mapsto \overline{e}'']\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$ we get $\langle e_1 \mid \sigma[w_k{:}T_k{\mapsto}v^R_k]_{1\leq k\leq m} \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_*. Therefore, from rule (LetRec-F) of Fig. 5, $\langle \texttt{let rec}\ \overline{w}{=}\overline{F}\ \texttt{in}\ e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ which proves the result.

We now prove the result for $C = ex$.
For the base cases and application, since the expressions are translated into IL expressions $\approx_{se}$ and $\approx_{ex}$ coincide. So the result holds.
Let e be a let expression, assignment or a sequence expression. Let $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho \rangle \approx_{ex} \langle e' \mid \rho'_e \rangle$. From Definition 16.2 there are se and ρ' such that $\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho \rangle \approx_{sq} \langle se \mid \rho' \rangle$. Moreover, se is not an expression. Therefore, let $[\![e^\circ]\!]_{sq}^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ, \overline{\delta}\,)$ where $e = e^\circ[\overline{y} := \overline{l}^y]$, $e' = \texttt{exc}(l', \overline{Y} \mapsto \overline{l}^y, \overline{l}^x)$ for $l' \notin dom(\rho')$, $\rho'_e(l') = \texttt{code}(\{se^\circ\}[\overline{z} := \overline{l}^z], \overline{y} \mapsto \overline{Y}, \overline{x})$ and for all $l \in dom(\rho) \cup \{\overline{l}^x, \overline{l}^z\}$ we have that $\rho'_e(l) = \rho'(l)$.
From $\langle e' \mid \rho'_e \rangle \Downarrow_{ex} \langle v' \mid \rho'_* \rangle$ and rule (CODE) of Fig. 11 we get $\langle l' \mid \rho'_e \rangle \Downarrow_{ex} \langle \texttt{code}(\{se^\circ[\overline{z} := \overline{l}^z]\}, \overline{y} \mapsto \overline{Y}, \overline{x}) \mid \rho'_e \rangle$, $\langle \overline{l}^x \mid \rho'_e \rangle \Downarrow_{ex} \langle \overline{v}') \mid \rho'_e \rangle$ where $v'_i = \rho'(l^x_i)$ $(1 \leq i \leq n)$.
Let $se' = se^\circ[\overline{x}\,\overline{y}\,\overline{w}\,\overline{z} := \overline{l}'\,\overline{l}^y\,\overline{l}^w\,\overline{l}^z]$ where $\{\overline{l}'\} \cap dom(\rho'_e) = \emptyset$. Define $\rho''_1 = \rho'_e[\overline{l}' \mapsto \overline{v}']$ and $\rho''_1 = \rho'_e[\overline{l}^w \mapsto \overline{?}]$. As for the corresponding case of the proof of Lemma 18, we can prove that

$$\Sigma, \overline{x}, \overline{y} \models \langle e \mid \sigma \mid \rho \rangle \approx_{se} \langle se' \mid \rho'' \rangle.$$

From $\langle se' \mid \rho'' \rangle \Downarrow_{bl} \langle v' \mid \rho'_* \rangle$ and rule (BLOCK) of Fig. 11 we get $\langle \{se'\} \mid \rho"_1 \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$. Therefore, applying the inductive hypotheses we derive that $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$ for some v and ρ_* which proves the result. $\square$

Proof of Theorem 12 (correctness):
Let e be an F#program, then for some T we have that $\emptyset \mid \emptyset \vdash e : T$ and so $\emptyset \models \langle e \mid \emptyset \mid \emptyset \rangle \diamond$.
Let $e'_i = \texttt{code}(bl_i, \overline{y}^i \mapsto \overline{Y}^i, \overline{x}^i)$, $\overline{\delta} = \texttt{def}\ \overline{z}{=}\overline{e}'$, $\rho' = [\overline{l}^z \mapsto \overline{e}'[\overline{z} \mapsto \overline{l}^z], \overline{l}^w \mapsto \overline{?}]$ and $se' = se[\overline{z}\,\overline{w} := \overline{l}^z\,\overline{l}^w]$. From Lemma 11.2 and 3, $FV(se') = \emptyset$, $Loc(se') \subseteq dom(\rho')$ and for all $l \in dom(\rho')$ we have that $FV(\rho'(l)) = \emptyset$ and $Loc(\rho'(l)) \subseteq dom(\rho')$. Therefore, the IL configuration $\langle se \mid \rho' \rangle$ is well-formed and $\emptyset, \emptyset, \emptyset \models \langle e \mid \emptyset \mid \emptyset \rangle \approx_{sq} \langle se' \mid \rho' \rangle$.
Assume that for some ρ and v, $\langle e \mid [] \mid [] \rangle \Downarrow \langle v \mid \rho \rangle$. From Lemma 18, $\langle se' \mid \rho' \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$ where $v \cong \langle v' \mid \rho'_* \rangle$. Let $\rho'' = [\overline{l}^z \mapsto \overline{?}, \overline{l}^w \mapsto \overline{?}]$, since for all $\texttt{def}\ z{=}e' \in \overline{\delta}$ we have that e' is a value (so also $e'[\overline{z} := \overline{l}^z]$ is a value), applying (repeatedly) rules (SEQ) and (DEF) of Fig. 11, we derive that $\langle \texttt{def}\ \overline{l}^z{=}(\overline{e}'[\overline{z} := \overline{l}^z]) \mid \rho'' \rangle \Downarrow_{sq} \langle v'' \mid \rho' \rangle$ for some v''. Applying rule (SEQ), we get $\langle (\overline{\delta}; se)[\overline{z}\,\overline{w} := \overline{l}^z\,\overline{l}^w] \mid \rho'' \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$ and from rule (BLOCK) we derive that $\langle \overline{\delta}; se \mid [] \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$. Since $v \cong \langle v' \mid \rho'_* \rangle$ and v is an integer or boolean value, from Definition 14 we have that $v = v'$.
On the other side, assume that $\langle \{\overline{\delta}; se\} \mid [] \rangle [] \Downarrow_{bl} \langle v \mid \rho'_* \rangle$ for some v and ρ'_*. Applying rule (BLOCK) of Fig. 11 we have that $\langle (\overline{\delta}; se)[\overline{z}\,\overline{w} := \overline{l}^z\,\overline{l}^w] \mid \rho'' \rangle \Downarrow_{sq} \langle v \mid \rho'_* \rangle$. Therefore, from repeated application of rules (SEQ) and (DEF) of Fig. 11 we have that, for some v'', $\langle \overline{\delta}[\overline{z}\,\overline{w} := \overline{l}^z\,\overline{l}^w] \mid \rho'' \rangle \Downarrow_{sq} \langle v'' \mid \rho'' \rangle$ and $\langle se[\overline{z}\,\overline{w} := \overline{l}^z\,\overline{l}^w] \mid \rho'' \rangle \Downarrow_{sq} \langle v \mid \rho'_* \rangle$. From $\emptyset, \emptyset, \emptyset \models \langle e \mid \emptyset \mid \emptyset \rangle \approx_{sq} \langle se' \mid \rho' \rangle$ and Lemma 19, we derive that $\langle e \mid [] \mid [] \rangle \Downarrow \langle v' \mid \rho_* \rangle$ for some ρ_* and v'. From Lemma 18, $v' \cong \langle v \mid \rho'_* \rangle$ and since v' is an integer or boolean value we have that $v = v'$. $\square$

5. Implementation

The compiler was implemented in `F#`. The implementation relies on two features, namely `F#` *code quotations* and *reflection*, with which one can reason about the source code and through transformation processes generate target language code. Follows a brief description of the compiler implementation. The aim of this section is to provide a general idea of the compiler's architecture, data structures and the implemented compiler phases needed to transform `F#` code into target language code.

5.1. Data structures

The IL grammar is described by discriminated unions defining three main syntactic categories of our intermediate language, namely *blocks*, *statements* and *expressions*. This is a traditional approach adopted by most functional implementations, e.g., see [3]. This data structure allows us to represent input programs as traditional abstract syntax trees that can be naturally traversed and manipulated in any functional language.

5.2. Architecture and translation

From an abstract point of view, the compiler's architecture can be thought of as in Fig. 13.

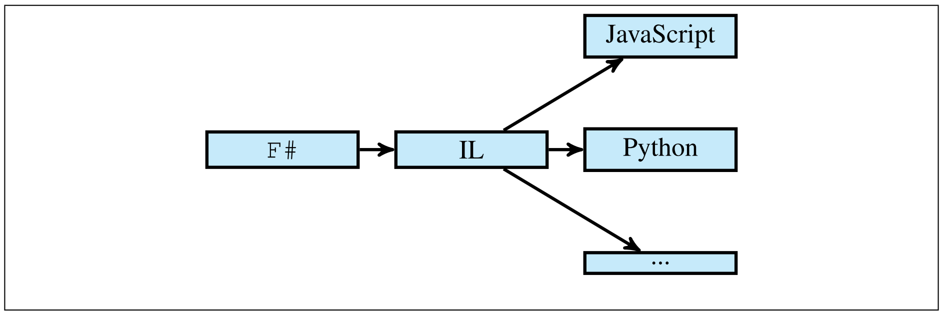

Figure 13: Compiler architecture

The `F#` source code is translated to IL. The compiler then delegates the translation of IL code to a target-specific driver. Currently, we implemented drivers for JavaScript and Python. A driver is an implementation of an `F#` interface defining methods that take in input an `IL` tree and return the target language code. The translation is done through a series of recursive method calls dispatched according to the construct type being translated.

The `IL` tree is generated from an AST representing the `F#` source code. This latter is obtained with the *Quotations* library that allows a programmer to mark `F#` code that should be ignored by the compiler and whose AST should be returned. This way, lexical and semantic analysis are done for us by the *Quotation* library. Fig. 14 shows an example of *Quotations* in action and the value of the `ast` constant is shown in Fig. 15.

```
open Microsoft.FSharp.Quotations

let ast = <@@ let add x y = x + y in add 2 3 @@>
```
Figure 14: Example `F#` quotations

```
val ast : Quotations.Expr =
  Let (add,
    Lambda (x,
      Lambda (y,
        Call (None,
           Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
           [x, y]))),
     Application (Application (add, Value (2)), Value (3)))
```
Figure 15: Result of example `F#` quotations

That is, instead of evaluating the code between `<@@ @@>`, the compiler returns an abstract syntax tree of the enclosed expression. The drawback of this method is that you may have to enclose many expressions making the code hard to read. For this reason, the Microsoft .NET platform has `ReflectedDefinitionAttribute`, a special attribute, that can be applied to a whole function, method or, in `F#` 3.x, to a module. The compiler will thus ignore the compilation of all constructs marked with this attribute and will instead return their abstract syntax tree representation.

To obtain the AST of all the functions or methods marked `ReflectedDefinition` we proceed as follows:

1. we compile at runtime the module containing the functions or methods, obtaining this way a compiled assembly
2. from the compiled assembly, by using the .NET reflection, we take all the defined types and other definitions
3. we filter out definitions without the `ReflectedDefinition` attribute
4. for each function/method with a `ReflectedDefinition` attribute we take the desired AST

Once the `F#` tree is constructed, we traverse it recursively relying on pattern matching and gradually build the corresponding `IL` tree. The *Quotation* library provides us with a rich set of active patterns for working with `F#` abstract syntax trees.

5.3. Extensions

We developed several extensions and are currently experimenting with new ones.

JavaScript DOM and project template. We developed a DOM manipulation library with a simple DSL for generating web pages. The DSL allows us to build web pages in a type-safe manner. Also, we implemented a small `F#` library that makes developing JavaScript applications easier. Thanks to this library, when the user launches a JavaScript application, the generated JavaScript code is put in a `.js` file that is loaded into a `.html` file, that in turn is served to the user through a browser by a small HTTP server that is automatically launched by the project.

Native and target code mix-ins. We are studying and have already partially implemented constructs for mixing in a type-safe manner native `F#` and target language code. This technique would let a developer reuse existing JavaScript code in a type-safe manner.

New language drivers. We emphasize that driver extensions are really easy to implement. Also, target language drivers can freely use target language libraries. For example, one could generate target language code that uses *jQuery* instead of plain JavaScript. Such a driver could inherit all the code from the original JavaScript driver and then just override the methods that should generate *jQuery* code.

Debugging. We are studying a debugging system that, when an error occurs in a target language, would correctly indicate the origin of errors in the origin language code.

Client-Server code. Also, following the example of [16], it would be useful to allow a programmer to separate client- and server-side code in a type-safe manner. This extension is in a very early stage.

Source languages. Many other extensions are possible. The `IL` in itself is not strongly linked to the source language, so one could implement a compiler which translates from a different source language.

The project can be downloaded at:

`https://www.assembla.com/spaces/bluestorm`

6. Comparisons with other work

Similar projects exist and are based on similar translation techniques, although, as far as we know, we are the first to introduce an intermediate language in order to translate to different target languages. Pit, see [5], and FunScript, see [4], are open source `F#` to JavaScript compilers. They support only translation to JavaScript. FunScript ha support for integration with JavaScript code. Websharper, see [11], is a professional web and mobile development framework, also available under an open source license. It is a very rich framework offering extensions for ExtJs, jQuery, Google Maps, WebGL and many more. Again it supports only JavaScript. `F#` Web Tools is an open source tool whose main objective is not the translation to JavaScript, instead, it is trying to solve the difficulties of web programming: “the heterogeneous nature of execution, the discontinuity between client and server parts of execution and the lack of type-checked execution on the client side”, see [16]. It does so by using meta-programming and monadic syntax. One of it features is translation to JavaScript. Finally, a translation between Ocaml byte code and JavaScript is provided by Ocsigen, and described in [18].

On the theoretical side, a framework integrating statically and dynamically typed (functional) languages is presented in [13]. Support for dynamic languages is provided with ad hoc constructs in Scala, see [14]. A construct similar to `code`, is studied in recent work by one of the authors, see [2], where it is shown how to use it to realize dynamic binding and meta-programming, an issue we are planning to address. The only work to our knowledge that proves the correctness of a translation between a statically typed functional language, with imperative features to a scripting language (namely JavaScript) is [6].

In [6] the proof of correctness is done by embedding the JavaScript translation in the functional language and showing that the semantics is preserved. Our proof instead is direct; we define a translation relation between `F#` and `IL` values and configuration and prove the correctness of the translation. That is, the evaluations of an `F#` configuration converge to a value if and only if the evaluation of the `IL` configuration which is in the relation translation with it converges to a related value. The soundness of the `F#` type system w.r.t. the operational semantics is essential in order to prove the correctness of the translation. Therefore, in the appendices we give also the proofs of type preservation and progress for well typed `F#` expressions w.r.t. the big-step semantics introduced. The use of a big-step semantics, for both languages, facilitates the already quite complex proof of equivalence, however, it introduced the need to characterize also non terminating computations, and prove that a well-typed `F#` expression either converges to a value or diverges.

7. Conclusions and future work

In this paper we proved that the translation of a significant fragment of `F#` to an intermediate language close to scripting languages such as Python and JavaScript is correct, in the sense that it preserves the dynamic semantics of the language. A richer version of the intermediate language, `IL`, and a preliminary version of the translation were presented at ICSOFT 2013, see [7] and [9]. We have a prototype implementation of the compiler that can be found at the project site [8]. The compiler is implemented in `F#` and is based on two metaprogramming features offered by the .net platform: *quotations* and *reflection*. Our future work will be on the practical side to use the intermediate language to integrate `F#` code and JavaScript or Python native code. (Some of the features of `IL`, such as dynamic type checking, which are not present in the current paper, as they were not relevant for the proof of correctness, were originally introduced for this purpose.) The current implementation also supports features such as namespacing, classes, pattern matching, discriminated unions, etc., some of which have poor or no support at all in JavaScript or Python. On the theoretical side, we are planning to do the proofs of correctness of the translations from `IL` to Python and JavaScript. For this, we need to formalize Python and JavaScript. (We anticipate that these proofs will be easier than the one from `F#` to `IL`.) Moreover, we want to formalize the integration of native code, and more in general meta-programming on the line of recent work by the authors, see [2] and [1].

AppendixA.

In this appendix we prove the Subject Reduction lemma.

Lemma 20. *If* $\Gamma[y{:}T'!] \mid \Sigma \vdash e : T$ *and* $l \notin dom(\Sigma)$*, and* $\models v{:}T'$ *then* $\Gamma \mid \Sigma[l{:}T'] \vdash e[l := v] : T$.

Proof:
By structural induction on e. $\square$

Lemma 21 (Inversion). 1. *If* $\Gamma \mid \Sigma \vdash n : T$*, then* $T = \texttt{int}$ *and if* $\Gamma \mid \Sigma \vdash \texttt{tr}, \texttt{fls} : T$ *then* $T = \texttt{bool}$.

2. *If* $\Gamma \mid \Sigma \vdash x : T$*, then* $x{:}T \dagger \in \Gamma$.
3. *If* $\Gamma \mid \Sigma \vdash l : T$*, then* $\Sigma(l) = T$.
4. *If* $\Gamma \mid \Sigma \vdash e_1{+}e_2 : T$ *then* $T = \texttt{int}$*, and* $\Gamma \mid \Sigma \vdash e_i : \texttt{int}$ $(1 \leq i \leq 2)$.
5. *If* $\Gamma \mid \Sigma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T$ *then* $\Gamma \mid \Sigma \vdash e_1 : \texttt{bool}$*, and* $\Gamma \mid \Sigma \vdash e_i : T$ $(2 \leq i \leq 3)$.
6. *If* $\Gamma \mid \Sigma \vdash \texttt{fun}\ x{:}T_1\texttt{->}e : T$ *then for some* T_2*,* Γ_1 *and* Γ_2
   (a) $T = T_1 \to T_2$*,* $\Gamma = \Gamma_1[\Gamma_2]$*,*
   (b) $\Gamma_2[x{:}T_1] \mid \emptyset \vdash e : T_2$ *and*
   (c) $\forall y, T \quad y{:}T! \notin \Gamma_2$.
7. *If* $\Gamma \mid \Sigma \vdash e_1\ e_2 : T$ *then* $\Gamma \mid \Sigma \vdash e_1 : T' \to T$ *for some* T' *and* $\Gamma \vdash e_2 : T'$.
8. *If* $\Gamma \mid \Sigma \vdash e_1, e_2 : T$ *then* $\Gamma \mid \Sigma \vdash e_1 : T'$ *for some* T' *and* $\Gamma \mid \Sigma \vdash e_2 : T$.
9. *If* $\Gamma \mid \Sigma \vdash \texttt{let}\ x{:}T_1{=}e_1\ \texttt{in}\ e_2 : T$ *then* $\Gamma \mid \Sigma \vdash e_1 : T_1$ *and* $\Gamma[x{:}T_1] \mid \Sigma \vdash e_2 : T$.
10. *If* $\Gamma \mid \Sigma \vdash \texttt{let mutable}\ y{:}T_1{=}e_1\ \texttt{in}\ e_2 : T$ *then* $\Gamma \mid \Sigma \vdash e_1 : T_1$ *and* $\Gamma[y{:}T_1!] \mid \Sigma \vdash e_2 : T$.
11. *If* $\Gamma \mid \Sigma \vdash \texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e : T$ *then* $\Gamma[\overline{w}{:}\overline{T}] \mid \Sigma \vdash F_j : T_j$ $(1 \leq j \leq n)$ *and* $\Gamma[\overline{w}{:}\overline{T}] \mid \Sigma \vdash e : T$.
12. *If* $\Gamma \mid \Sigma \vdash l\texttt{<-}e : T$ *then* $\Gamma \mid \Sigma \vdash e : T$ *and* $\Sigma(l) = T$.

Proof:
By induction on typing derivations. For each case, we have that the last rule applied in the derivation of $\Gamma \vdash e : T$ is the typing rule corresponding to the syntactic construct e. The result follows by analysis of the structural rules. $\square$

Proof of Lemma 3 (Type Preservation):
Let $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. From Definition 2,

1. $\mathit{env}(\sigma) \mid \Sigma \vdash e : T$ for some T
2. $\models \sigma\diamond$ and
3. $\Sigma \models \rho$.

Let $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$. By induction on the derivation of $\Downarrow$. We only show the most interesting cases which are the base cases, the rule for application, let recursive and mutable and assignment. The others are similar. Consider the last rule applied in the derivation.

Rule (Var-F) In this case $e{=}x$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathit{lkp}(x,\sigma) \mid \rho\rangle$. Let $x{:}T{\mapsto}v' \in \sigma$. If $\mathit{lkp}(x,\sigma) = v'$, then, from 2, $\models v'{:}T$, otherwise $v' = (\texttt{let rec}\ \overline{x}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$ and $v = (F_k, \sigma[x_i{:}T_i{\mapsto}(\texttt{let rec}\ \overline{x}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)]_{1\leq i\leq n})$. From $\models v'{:}T$ we have that $T = T_k$ and $\mathit{env}(\sigma)[\overline{x}{:}\overline{T}] \mid \emptyset \vdash F_k : T_k$. Therefore, from Definition 2.1(c) we also have $\models v{:}T$. Note that, if $T = T_1 \to T_2$ then $v = (\texttt{fun}\ x{:}T_1\texttt{->}e', \sigma')$ for some e' and σ'. Moreover, from 3, we have $\Sigma \models \rho$.

Rule (Pr-Val-F) In this case $e{=}n$ or $e{=}\texttt{tr}$ or $e{=}\texttt{fls}$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle e \mid \rho\rangle$, i.e., $v{=}e$. From 1 and Lemma 21.1, we have that for $e = n$ then $T = \texttt{int}$ and for $e{=}\texttt{tr}$ or $e{=}\texttt{fls}$ then $T = \texttt{bool}$. Therefore, from Definition 2.1(a) and (b) we have $\models v{:}T$. Note that, $T = T_1 \to T_2$ and $v = (\texttt{fun}\ x{:}T_1\texttt{->}e', \sigma')$.Moreover, from 3, we have $\Sigma \models \rho$.

Rule (Fn-Val-F) In this case $e{=}\texttt{fun}\ x{:}T_1\texttt{->}e'$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle (e,\sigma) \mid \rho\rangle$, i.e., $v = (e,\sigma)$. From 1 and Lemma 21.6, $T = T_1 \to T_2$, $\mathit{env}(\sigma) = \Gamma_1[\Gamma_2]$, $\Gamma_2[x{:}T] \mid \emptyset \vdash e' : T_2$ and $\forall y, T\ \ y{:}T! \not\in \Gamma_2$. Since in $\mathit{env}(\sigma)$ there are not $y{:}T!$, also $\mathit{env}(\sigma)[x{:}T] \mid \emptyset \vdash e' : T_2$. Therefore, from Definition 2.1(c), we get $\models (e,\sigma){:}T$. Moreover, from 3, we have $\Sigma \models \rho$.

Rule (Loc-F) In this case $e{=}l$ for some l; so $\langle l \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$ where $v = \rho(l)$. From 1 and Lemma 21.3, $T = \Sigma(T)$. From 3 and Definition 2.3 we have $\models \rho(l){:}T$, $\Sigma \models \rho$ and, in case T is a function type, v has the required shape.

Rule (App-F) In this case $e{=}e_1\ e_2$ for some e_1 and e_2. From $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ we have that $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ and $\langle e_b \mid \sigma'[x{:}T_1{\mapsto}v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some T_1. From 1 and Lemma 21.7, we have that $\Gamma \mid \Sigma \vdash e_1 : T' \to T$, $\Gamma \mid \Sigma \vdash e_2 : T'$ for some T'. From 2 and 3, $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma') \mid \rho_1\rangle$ we have that $\models (\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma'){:}T' \to T$ and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$.
From $\Gamma \mid \Sigma \vdash e_2 : T'$, 2 and $\Sigma_1 \models \rho_1$, we derive $\Sigma_1 \models \langle e_2 \mid \sigma \mid \rho_1\rangle\diamond$. Applying the inductive hypothesis to $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ we have that $\models v_x{:}T'$ and $\Sigma_2 \models \rho_2$ for some $\Sigma_2 \supseteq \Sigma_1$.
From $\models (\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma'){:}T' \to T$ and Definition 2.1(c), $\mathit{env}(\sigma') \mid \Sigma \vdash \texttt{fun}\ x{:}T_1\texttt{->}e_b : T' \to T$. From Lemma 21.6, $T' = T_1$ and $\mathit{env}(\sigma')[x{:}T_1] \mid \emptyset \vdash e_b : T$. Moreover, we have that $\models \sigma'\diamond$. From $\models v_x{:}T_1$, we have that $\models \sigma'[x{:}T_1 \mapsto v_x]\diamond$ and so $\Sigma_2 \models \langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle\diamond$. Applying the inductive hypothesis to $\langle e_b \mid \sigma'[x{:}T_1{\mapsto}v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$ we have that $\models v{:}T$ and $\Sigma' \models \rho_*$ for some $\Sigma' \supseteq \Sigma_2$ which proves the result. In case T is a function type, from the inductive hypotheses we also have that v has the required shape.

Rule (Let-Mut-F) In this case $e{=}\texttt{let mutable}\ y{:}T'{=}e_1\ \texttt{in}\ e_2$, and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$, $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$ where $l_y \not\in \rho_1$.
From 1 and Lemma 21.10, we have that $T' = T_1$ and $\Gamma \mid \Sigma \vdash e_1 : T_1$, so from 2 and 3, $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$ we have that $\models v_y{:}T_1$ and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$.
From Lemma 21.10 and $\Sigma_1 \supseteq \Sigma$, $\Gamma[y{:}T_1!] \mid \Sigma_1 \vdash e_2 : T$. From Lemma 20, since $l_y \not\in \mathit{dom}(\rho_1)$ we derive $\Gamma \mid \Sigma_1[l_y{:}T_1] \vdash e_2[y := l_y] : T$. Moreover, $\Sigma_1[l_y{:}T_1] \models \rho_1[l_y \mapsto v_y]$. Therefore, from 2, $\Sigma_1[l_y{:}T_1] \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle\diamond$. Applying the inductive hypothesis to $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$ we have that $\models v{:}T$ and $\Sigma' \models \rho_*$ for some $\Sigma' \supseteq \Sigma_1[l_y{:}T']$ which proves the result. In case T is a function type, from the inductive hypotheses we also have that v has the required shape.

Rule (LetRec-F) In this case e=`let rec` $\overline{w}$=$\overline{F}$ `in` e_1 and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ where $\langle e_1 \mid \sigma' \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ where $\sigma' = \sigma[w_j{:}T_j \mapsto (\texttt{let rec}\,\overline{w}{:}\overline{T}{=}\overline{F}\,\texttt{in}\,F_j,\sigma)]_{1\leq j\leq m}$.
From Lemma 21.11 and 2 we get $\models \sigma'\diamond$, and $\Gamma[\overline{w}{:}\overline{T}] \mid \Sigma \vdash e : T$. Therefore, from 3, $\Sigma \models \langle e_1 \mid \sigma' \mid \rho\rangle\diamond$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma' \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ we have that $\models v{:}T$ and $\Sigma' \models \rho_*$ for some $\Sigma' \supseteq \Sigma$. In case T is a function type, from the inductive hypotheses we also have that v has the required shape.

Rule (Ass-F) In this case $e{=}l{<}{-}e_1$ and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ and $\rho_* = \rho_1[l \mapsto v]$.
From 1 and Lemma 21.12, we have that $\Gamma \mid \Sigma \vdash e_1 : T$, therefore from 2 and 3 $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$, we have that $\models v{:}T$ and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$.
From Lemma 21.12, $\Sigma(l) = T$, so also $\Sigma_1(l) = T$, and from $\models v{:}T$, $\Sigma_1 \models \rho_1[l \mapsto v]$.
In case T is a function type, from the inductive hypotheses we also have that v has the required shape. □

AppendixB.

In this appendix we prove the Progress lemma. Since we give a coinductive interpretation to the rules of Fig. 6, the proof of the lemma uses coinduction. In particular, we prove that the assumption that "either $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* or $\langle e \mid \sigma \mid \rho\rangle\Uparrow$" is compatible with the rules defining the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ and also that the "or" is an "exclusive or".

Proof of Lemma 4 (Progress):
From $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ we have

1. $env(\sigma) \mid \Sigma \vdash e : T$ for some T
2. $\models \sigma\diamond$ and
3. $\Sigma \models \rho$.

The proof is by coinduction and case analysis over e.

For the cases $e{=}n$, e=`tr`, e=`fls`, e=`fun` $x{:}T_1$`->`e_1, $e = x$ and $e = l$ we have that $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$ for some v. (For $e = x$ and $e = l$ we have to use 1 to prove that the corresponding rule is applicable.)
For the structured expressions, we only show the proof for application and let mutable. The others are similar and simpler.

$\underline{e{=}e_1\ e_2}$ for some e_1, e_2. By excluded middle, either $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ (for some v_1 and ρ_1) or not.
In the latter case, the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (App-⇑) of Fig. 6 and the coinductive hypothesis $\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow$ using the first disjunct of the premises. Moreover, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* is false.
From 1 and rule (TyApp), we have $env(\sigma) \mid \Sigma \vdash e_1 : T' \to T$ and $env(\sigma) \mid \Sigma \vdash$

$e_2 : T'$ for some T'.
If $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$, from Lemma 3, we derive that $v_1 = (\texttt{fun}\, x{:}T'\texttt{->}e_b, \sigma')$ for some e_b and σ', $\models (\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma'){:}T' \to T$ and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$. From $env(\sigma) \mid \Sigma \vdash e_2 : T'$ we also have that $\Sigma_1 \models \langle e_2 \mid \sigma \mid \rho_1\rangle\diamond$.
By excluded middle, either $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ (for some v_x and ρ_2) or not. In the latter case, the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (App-⇑) of Fig. 6, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$ and the coinductive hypothesis $\langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow$ using the second disjunct of the premises. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* is false.
If $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ from Lemma 3, we derive that $\models v_x{:}T'$ and $\Sigma_2 \models \rho_2$ for some $\Sigma_2 \supseteq \Sigma_1$. From $\models (\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma'){:}T' \to T$ and rule (TyAbs) of Fig. 3 we have $env(\sigma'), x{:}T' \mid \emptyset \vdash e_b : T$ and $\models \sigma'\diamond$. Therefore, from $\models v_x{:}T'$ follows that $\models \sigma'[x{:}T' \mapsto v_x]\diamond$ and so $\Sigma_2 \models \langle e_b \mid \sigma'[x{:}T' \mapsto v_x] \mid \rho_2\rangle\diamond$.
By excluded middle again, either $\langle e_b \mid \sigma'[x{:}T' \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$ (for some v and ρ_*) or not. In the latter case, the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (App-⇑) of Fig. 6, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle(\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ and the coinductive hypothesis $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle\Uparrow$ using the third disjunct of the premises. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* is false.
Instead, if $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$, then from $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle(\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ and rule (App-F) of Fig. 5 we derive $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$. Moreover, it is not the case that $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. This concludes the proof.

$\underline{e{=}\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2}$ for some e_1 and e_2. By excluded middle, either $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$ (for some v_y and ρ_1) or not.
In the latter case, the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (LetMut-⇑) of Fig. 6 and the coinductive hypothesis $\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow$, using the first disjunct of the premises. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* is false.
From 1 and rule (TyMut) of Fig. 3, $env(\sigma) \mid \Sigma \vdash e_1 : T'$ for some T' and $env(\sigma), y{:}T'! \mid \Sigma \vdash e_2 : T$. If $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$, from Lemma 3 we have that $\models v_y{:}T'$ and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$. If $l_y \notin dom(\rho_1)$, from Lemma 20, we derive that $env(\sigma) \mid \Sigma, l_y{:}T' \vdash e_2[y := l_y] : T$. Moreover, from $\Sigma_1 \models \rho_1$ and $\models v_y{:}T'$ it follows that $\Sigma_1[l_y{:}T'] \models \rho_1[l_y \mapsto v_y]$. Therefore, we get $\rho_1[l_y \mapsto v_y] \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle\diamond$.
By excluded middle, we must have that either $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$ (for some v and ρ_*) or not. In the latter case, the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (LetMut-⇑) of Fig. 6, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ and the coinductive hypothesis $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle\Uparrow$ using the second disjunct of the premises. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ for some v and ρ_* is false.
If, instead, $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$, then from $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$ and rule (LetMut-F) of Fig. 5 $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$. Moreover, it is not the case that $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. This concludes the proof. □

Bibliography

[1] D. Ancona, P. Giannini, E. Zucca, Incremental rebinding, in ICTCS 2013, `http://www.di.unito.it/~giannini/papers/ICTCS13.pdf`, 2013.

[2] D. Ancona, P. Giannini, E. Zucca, Reconciling positional and nominal binding, in: ITRS 2012, EPTCS, 2013.

[3] A. W. Appel, Modern Compiler Implementation in ML, Cambridge University Press, 1998.

[4] Z. Bray, Funscript, `http://tomasp.net/files/funscript/tutorial.html` (February 2013).

[5] M. S. Fahad, Pit - F Sharp to JS compiler, `http://pitfw.org/` (May 2012).

[6] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, B. Livshits, Fully abstract compilation to javascript, in: POPL, ACM, 2013.

[7] P. Giannini, A. Shaqiri, An intermediate language for compilation to scripting languages, in: J. Cordeiro, D. A. Marca, M. van Sinderen (eds.), ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies, Reykjavík, Iceland, SciTePress, 2013.

[8] P. Giannini, A. Shaqiri, Blue Storm Compiler, `https://www.assembla.com/spaces/bluestorm` (2014).

[9] P. Giannini, A. Shaqiri, Compiling functional to scripting languages, Communications in Computer and Information Science.

[10] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM TOPLAS 23 (3) (2001) 396–450.

[11] Intellifactory, Websharper 2010 platform, `http://websharper.com/` (May 2012).

[12] X. Leroy, H. Grall, Coinductive big-step operational semantics, Inf. Comput. 207 (2) (2009) 284–304.

[13] J. Matthews, R. B. Findler, Operational semantics for multi-language programs, ACM Trans. Program. Lang. Syst. 31 (3).

[14] A. Moors, T. Rompf, P. Haller, M. Odersky, Scala-virtualized, in: O. Kiselyov, S. Thompson (eds.), Proceedings of PEPM 2012, Philadelphia, Pennsylvania, USA, ACM, 2012.

[15] A. Nanevski, From dynamic binding to state via modal possibility, in: PPDP'03, ACM, 2003.

[16] T. Petříček, D. Syme, AFAX: Rich client/server web applications in `F#`, `http://www.scribd.com/doc/54421045/Web-Apps-in-F-Sharp` (May 2012).

[17] J. F. Ranson, H. J. Hamilton, P. W. L. Fong, A semantics of python in isabelle/hol, Tech. Rep. CS-2008-04, CS Department, University of Regina,Saskatchewan (2008).

[18] J. Vouillon, V. Balat, From bytecode to javascript: the js of ocaml compiler, `http://www.pps.univ-paris-diderot.fr/~balat/publi.php` (2011).