Scientific Annals of Computer Science vol. 27 (1), 2017, pp. 19–76
doi: 10.7561/SACS.2017.1.19

A Provably Correct Compilation of Functional Languages into Scripting Languages

Paola GIANNINI[1], Albert SHAQIRI[2]

Abstract

In this paper we consider the problem of translating core F#, a typed functional language including mutable variables and exception handling, 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). IL is a block-structured imperative language in which 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.

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

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

[1] Università del Piemonte Orientale, DiSIT, Via Teresa Michel 11, 15121 Alessandria, Italy, E-mail: paola.giannini@uniupo.it
This original research has the financial support of the Università del Piemonte Orientale.
[2] Dipartimento di Informatica, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy, E-mail: albert.shaqiri@unimi.it

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.

To deal with these problems, in previous work, [12] and [14], 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, when translating to languages that do not have complete support for functions as first class objects (such as some Python dialects) or for translating to JavaScript (optionally using 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. 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. This is used to translate mutually recursive function definitions. (IL is partly inspired by IntegerPython, see [21].) 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 [12], 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 big-step semantics 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 discuss the challenges of the translation, formalize the translation giving the translation of F# expressions to IL constructs, and we prove that an F# program converges to a primitive value if and only if its IL translation converges to the same value. In Section 5, we sketch the implementation of the compiler and some possible extensions. Section 6 reviews the related work, and Section 7 draws some conclusions and discusses future work. The first two appendices contain the proof of the lemmas of type preservation and progress for F# expressions, stated in Section 2. These are the results from which the soundness of the type system for F# is proved. The third appendix contains the proof of the main lemma needed to prove the correctness of the translation.

This paper is an extended and completely revised version of [12] and [14], whose aim was to introduce IL and show the challenges of the translation. In this paper, we give a different definition of the operational semantics of both languages, add exception handling to the source and target languages, and formalize and prove the correctness of the translation from the source language F# to the target language IL. Following the suggestion of a previous referee, we also define a translation which is simpler that the one of [12] and [14].

2 Core F#

The syntax for core F# language is presented in Fig. 1. We include constructs, such as `let`, `let mutable`, `let rec`, `raise` and `try-with` that are used in the practice of programming and that raise challenges in the translation to dynamic languages. We do not introduce imperative features through reference types, but through mutable variables, as this is closer to the imperative style of programming. Reference types, with assignment and a dereferencing operator, could be easily handled. We present a simply typed version of F#. However, our translation does not depend on the presence of types, since it uses F# type inference.

In the grammar for expressions, in Fig. 1, the square brackets " [...]" delimit an optional part of the syntax, we use x, y, z for variable names,

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

Figure 1: Syntax of core F#

and the overbar sequence notation is used according to [15]. 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 as usual. 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.

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 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 also enforces 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 used in functions, however our compiler handles such variables.

The construct `raise` e raises an exception with the value of the expression e. Exceptions are caught with the `try` e_1 `with` x -> e_2 construct. In case the evaluation of e_1 raises an exception, in the evaluation of e_2 the variable x is bound to the value raised.

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#. We also assume that expressions associated with exceptions be of type T_{E}.

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 † denote either ! or the empty string, and let $dom(\Gamma) = \{x \mid x{:}T\,\dagger \in \Gamma\}$. As usual we assume that Γ is mapping on variables. 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. 2. In the rules of Fig. 2, 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 about its free variables and formal parameter. From the definition of $\Gamma[\Gamma']$ we have that, in the environment in which the function is defined 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. This is enforced in the second row of the premise, by checking that the environment Γ', in which the body of the function is typed, does not contain any declaration $y{:}T''!$ for some y and T''.
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 (TyMut) 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 in $\overline{w}$ are bound to the types in $\overline{T}$, 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 $(1 \le k \le m)$.

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. Values and stacks are defined as follows.

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

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

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

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

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1, e_2 : T}\ (\textsc{TySeq}) \qquad \frac{\Gamma \vdash e : T \quad x{:}T! \in \Gamma}{\Gamma \vdash x \texttt{<-} e : T}\ (\textsc{TyAssign})$$

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma[x{:}T'] \vdash e_2 : T}{\Gamma \vdash \texttt{let}\ x{:}T' \texttt{=} e_1\ \texttt{in}\ e_2 : T}\ (\textsc{TyLet}) \qquad \frac{\Gamma[\overline{w}{:}\overline{T}] \vdash \overline{F} : \overline{T} \quad \Gamma[\overline{w}{:}\overline{T}] \vdash e : T}{\Gamma \vdash \texttt{let rec}\ \overline{w}{:}\overline{T} \texttt{=} \overline{F}\ \texttt{in}\ e : T}\ (\textsc{TyRec})$$

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma[y{:}T'!] \vdash e_2 : T}{\Gamma \vdash \texttt{let mutable}\ y{:}T \texttt{=} e_1\ \texttt{in}\ e_2 : T}\ (\textsc{TyMut})$$

$$\frac{\Gamma \vdash e : T_{\mathsf{E}}}{\Gamma \vdash \texttt{raise}\ e : T}\ (\textsc{TyThrow}) \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma[x{:}T_{\mathsf{E}}] \vdash e_2 : T}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ x\ \texttt{->}\ e_2 : T}\ (\textsc{TyCatch})$$

Figure 2: Typing rules for core F#

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.

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 with the variable in the stack, if any. It handles recursive function values by returning their body and a stack in which the names of the mutually recursive function are bound to their recursive definition.

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

- $lkp(x, \sigma[x{:}T{\mapsto}v]) = (F_k, \sigma[w_i{:}T_i{\mapsto}(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma)]_{1\le i\le m})$, *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, dubbed $env(\sigma)$*, is defined by:

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

Core F# 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* are defined by adding the clauses:

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

to the grammar of expression of Fig. 1. In Fig. 3 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

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

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

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

$$\frac{\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \texttt{tr} \mid \rho_1\rangle \quad \langle e_1 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle}{\langle \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_2\rangle}\ (\textsc{If-True})$$

$$\frac{\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \texttt{fls} \mid \rho_1\rangle \quad \langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle}{\langle \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_2\rangle}\ (\textsc{If-Fls})$$

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

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

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

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

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

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

Figure 3: Big-step operational semantics for core F#

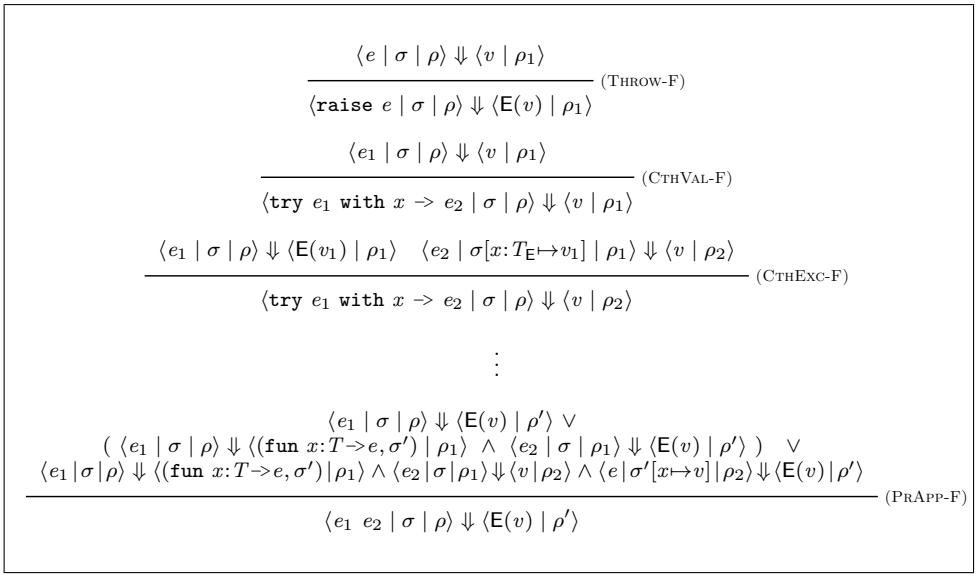

Figure 4: Big-step operational semantics of exception constructs

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 the function with 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. With $\tilde{n}$ we denote the integer value corresponding to n, and with $+^{\texttt{int}}$ the sum operation between integers. 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 a location. Between these occurrences there are the variables on the left-hand-side of assignments. Indeed, since in well-typed expressions variables on the left-hand-side of assignments were always introduced by `let mutable`, when an assignment is evaluated, rule (Ass-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, $(\texttt{let rec } \overline{w}{:}\overline{T}\texttt{=}\overline{F} \texttt{ in } F_i, \sigma)$, 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. In Fig. 4 we define the rules dealing with exception generation, and handling. Exceptions are generated by `raise` e, rule (Throw-F), and caught by the `try` e_1 `with` x `->` e_2 construct. If the evaluation of e_1 produces a value, then the construct evaluates to such a value, instead if the evaluation of e_1 raises an exception with value v, then the expression e_2 is evaluated, after binding x to v, rule (CthExc-F). The propagation rules are obvious. We show only the one for application.

We say that a *configuration* $\langle e \mid \sigma \mid \rho \rangle$ *converges*, dubbed $\langle e \mid \sigma \mid \rho \rangle\Downarrow$, if either $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho' \rangle$ or $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle \mathsf{E}(v) \mid \rho' \rangle$ for some v, and ρ'.

The typing rules in Fig. 2 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. 2, 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 subterms of function terms.

Definition 2 *1. An* F# *value* v has type T, *dubbed* $\models v{:}T$,

   (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, $1 \le j \le m$, $env(\sigma)[\overline{x}{:}\overline{T}] \mid \emptyset \vdash F_j : T_j$.

2. A stack σ is well-typed, *dubbed* $\models \sigma\diamond$, *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, *dubbed* $\Sigma \models \rho$, *if* $dom(\rho) = dom(\Sigma)$, *and for all* $l \in \rho$ *we have that* $\models \rho(l){:}\Sigma(l)$.

4. The F# *configuration* $\langle e \mid \sigma \mid \rho \rangle$ is well-typed, with respect to Σ, *dubbed* $\Sigma \models \langle e \mid \sigma \mid \rho \rangle\diamond$, *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 1 (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 Appendix A. □

Also evaluation of a well-typed configuration resulting in exceptions preserves well-typed stores, and the type of the value associated to exceptions.

Lemma 2 (Type Preservation for Exceptions) *Let* $\langle e \mid \sigma \mid \rho\rangle$ *be such that* $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. *If* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$, *then* $\models v{:}T_{\mathsf{E}}$, *and* $\Sigma' \models \rho'$ *for some* $\Sigma' \supseteq \Sigma$.

Proof The proof of the lemma is given in Appendix A. □

A 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, [12] and [14], we defined a small-step semantics which ensures that well-typed programs do not "get stuck". Here, following [19], we give a coinductive characterization of the *divergence judgment* $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. The divergence judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ is defined in Fig. 5, where the rules have to be interpreted coinductively, i.e., they define infinite derivation trees. The rules are obvious.

The following lemma asserts that our operational semantics is consistent, in the sense that given a configuration $\langle e \mid \sigma \mid \rho\rangle$ if $\langle e \mid \sigma \mid \rho\rangle\Downarrow$, i.e., either the rules of Fig. 3 or Fig. 4 are applicable, then it is not the case that $\langle e \mid \sigma \mid \rho\rangle\Uparrow$, i.e., the rules of Fig. 5 are not applicable. Moreover, if $\langle e \mid \sigma \mid \rho\rangle$ converges to a value then it cannot generate an exception.

Lemma 3 (Consistency) *Given a configuration* $\langle e \mid \sigma \mid \rho\rangle$, *if* $\langle e \mid \sigma \mid \rho\rangle\Downarrow$, *then* $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ *is not derivable. Moreover, if* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$ *for some* v *and* ρ', *then there is no* $\mathsf{E}(v')$ *and* ρ'' *such that* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v') \mid \rho''\rangle$.

Proof The proof of the lemma is given in Appendix B. □

The progress property says that well-typed configurations do not "get stuck". The lemma, that follows, asserts such a property by showing that, for a well-typed configuration $\langle e \mid \sigma \mid \rho\rangle$, either $\langle e \mid \sigma \mid \rho\rangle\Downarrow$ or $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. That is, if no rule of Fig. 3 or Fig. 4 is applicable to $\langle e \mid \sigma \mid \rho\rangle$, then the configuration diverges according to the rules of Fig. 5, i.e., $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ is provable.

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* $\langle e \mid \sigma \mid \rho\rangle\Downarrow$ *or* $\langle e \mid \sigma \mid \rho\rangle\Uparrow$.

Proof The proof of the lemma is given in Appendix B. □

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

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

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

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

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

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

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

$$\frac{\langle e \mid \sigma \mid \rho\rangle\Uparrow}{\langle l{\leftarrow}e \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Ass-}\Uparrow) \qquad \frac{\langle e \mid \sigma \mid \rho\rangle\Uparrow}{\langle \texttt{raise}\ e \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{Throw-}\Uparrow)$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle\Uparrow}{\langle \texttt{try}\ e_1\ \texttt{with}\ x\ {\to}\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{CthVal-}\Uparrow)$$

$$\frac{\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v_1) \mid \rho_1\rangle \quad \langle e_2 \mid \sigma[x{:}T_{\mathsf{E}}{\mapsto}v_1] \mid \rho_1\rangle\Uparrow}{\langle \texttt{try}\ e_1\ \texttt{with}\ x\ {\to}\ e_2 \mid \sigma \mid \rho\rangle\Uparrow}\ (\textsc{CthYes-}\Uparrow)$$

Figure 5: Coinductive characterization of divergence

Soundness of the type system for the big step semantics is expressed by the following theorem.

Theorem 1 (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* $\models v{:}T$ *or*
- $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$ *where* $\models v{:}T_{\mathsf{E}}$ *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 $\langle e \mid \sigma \mid \rho\rangle\Downarrow$ or $\langle e \mid \sigma \mid \rho\rangle\Uparrow$. Assume that $\langle e \mid \sigma \mid \rho\rangle\Downarrow$. If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$, then, from Lemma 1, we have that $\models v{:}T$. If $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$, then, from Lemma 2, we have that $\models v{:}T_{\mathsf{E}}$. □

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 1 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

In designing `IL` our goals were: on one hand to have a language close enough to the structure of the target languages (JavaScript and Python), and on the other to allow us to give a translation simple enough to be formally proved to preserve the semantics of the original language.

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 [4] for a similar choice). The block structure is inspired by IntegerPython (see [21]). 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. This feature, present in Python, is useful, for the translation of the `letrec` construct. As in Python and JavaScript closures are expressions, but sequences of expressions are not expressions, as this is a feature that could not be present in the target language. The constructs for handling exceptions are modeled to be similar to the ones of Python and JavaScript.

3.1 Syntax and Semantics of IL

In the syntax of IL, given in Fig. 6, there are three (main) syntactic categories: *blocks*, *statements*, and *expressions*.

bl	$::=$	$\{se\}$	block
se	$::=$	$st \mid e \mid se; se$	sequence
st	$::=$	x<-$e \mid$ `def` x:=$e \mid$ `try` bl `catch` $(\, x \,)$ bl	statement
e	$::=$	$x \mid n \mid$ `tr` $\mid$ `fls` $\mid$ `fun` x->$bl \mid e$+$e \mid$ `if` e `then` bl `else` bl $\mid e\ e \mid$ `raise` e	expression
v	$::=$	$n \mid$ `tr` $\mid$ `fls` $\mid$ `fun` x->bl	value

Figure 6: Syntax of IL

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 or the constructs of exception generation and catching. 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.,

```
{ def f=fun y->{x}; def x=5; (f 2) }
```

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

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

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

the block would return 7, since `x` is bound in the enclosing block. This is also the behavior in JavaScript and Python. Values are integers, booleans, and functions (as for `F#`).

In Fig. 7, we define the *free variables of expressions, sequences, and blocks* ($FV(e)$, $FV(se)$, and $FV(bl)$). Note that, the assignment statement does not define the assigned variable, which is free in the statement.

- $FV(e)$, is defined by
  - $FV(x) = \{x\}$
  - $FV(n) = FV(\texttt{tr}) = FV(\texttt{fls}) = \emptyset$
  - $FV(\texttt{fun}\ x\texttt{->}bl) = FV(bl) - \{x\}$
  - $FV(e_1\texttt{+}e_2) = FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$
  - $FV(\texttt{if}\ e\ \texttt{then}\ bl_1\ \texttt{else}\ bl_2) = FV(bl_1) \cup FV(bl_2) \cup FV(e)$
  - $FV(\texttt{raise}\ e) = FV(e)$
- $FV(se)$, is defined by
  - $FV(x\texttt{<-}e) = FV(e) \cup \{x\}$
  - $FV(\texttt{def}\ x\texttt{:=}e) = FV(e)$
  - $FV(\texttt{try}\ bl_1\ \texttt{catch}(\,x\,)\ bl_2) = FV(se_1) \cup (FV(se_2) - \{x\})$
  - $FV(se_1; se_2) = FV(se_1) \cup FV(se_2)$
- $def(se)$ is defined by
  - $def(x\texttt{<-}e) = def(e) = \emptyset$
  - $def(\texttt{def}\ x\texttt{:=}e) = \{x\}$
  - $def(se_1; se_2) = def(se_1) \cup def(se_2)$
- $FV(\{se\}) = FV(se) - def(se)$

Figure 7: Free variables of expressions and blocks

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, "$\langle C \mid \rho\rangle$ ", in which 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 clauses for expressions and statements that follows:

$$\begin{array}{lcl} st & ::= & \cdots \mid l{<}{-}e \mid \texttt{def}\ l{:=}e \\ e & ::= & \cdots \mid l \end{array}$$

The rules of the operational semantics are defined in Fig. 8.

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

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

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

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

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

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

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

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

Figure 8: Big-step operational semantics for IL

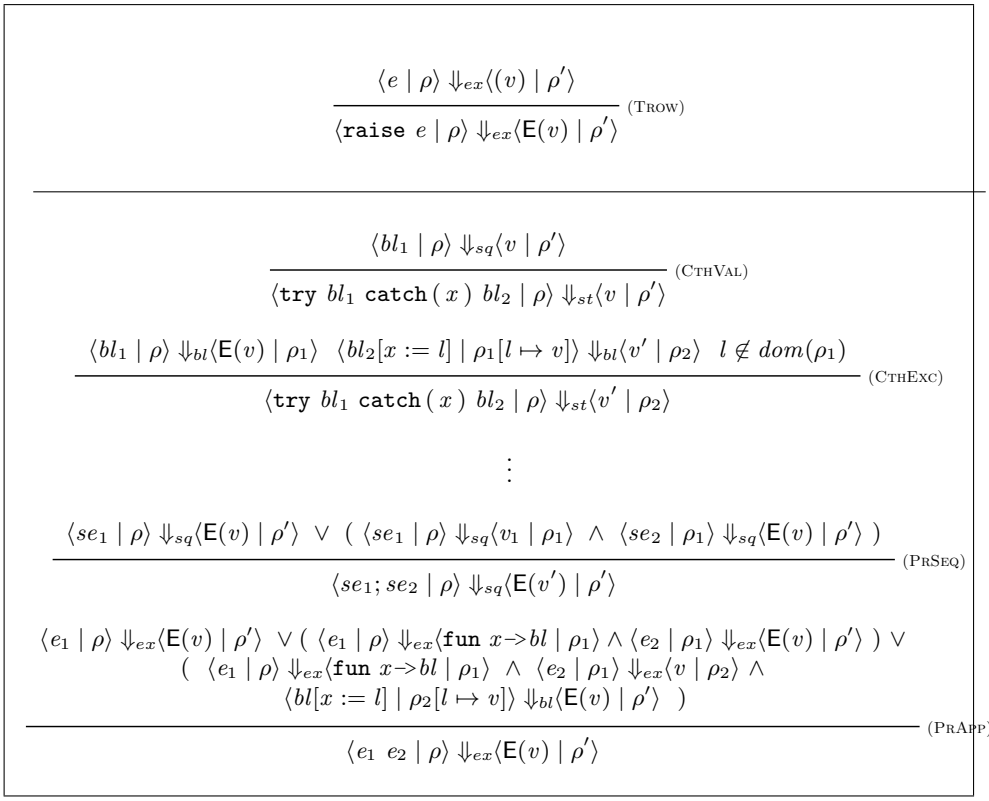

Figure 9: Exception generation, handling an propagation in IL

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. 7. We want to model forward declarations in blocks and the fact that the evaluation of an access to variables before it is assigned a value is not permitted. To this extent, the initial value of the location 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 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, which must result in a function, then the actual parameter of the function is evaluated. A location is allocated in 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 Fig. 9 we define the rules dealing with exception generation, and handling, which are similar to the ones for `F#`. We only show the propagation rule for sequencing and application.

Let C be an `IL` syntactic construct or $\mathsf{E}(v)$ for some v, $Loc(C)$, is the set of locations occurring in C. We define well-formed configurations.

Definition 3 *1. An* `IL` *store* ρ is location closed *if* $l \in dom(\rho)$ *implies 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. 8 preserves well-formed configurations, as the following theorem states.

Theorem 2 *Let* $\langle C \mid \rho \rangle$ *be well-formed, if* $\langle C \mid \rho \rangle \Downarrow_{\_} \langle C' \mid \rho' \rangle$, *then* $\langle C' \mid \rho' \rangle$ *is well-formed.*

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

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

Rule (App) From $\langle e_1\ e_2 \mid \rho\rangle$ well-formed we get that 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, $Loc(bl[x := l]) \subseteq dom(\rho_2[l \mapsto v'])$, and $\langle bl[x := l] \mid \rho_2[l \mapsto v']\rangle$ 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. □

□

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 well-formed. From Theorem 2 we derive that the execution of an IL program produces a well-formed configuration.

4 Translation

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. For example,

```
let mutable x = 3 in
  let z = x in
     x <- x+1, z
```

is translated in the following sequence

```
def x := 3; def z := x; x <- x+1; z
```

Note that, we do not distinguish immutable from mutable variables and the `let` construct is flattened in sequence starting with the definition of the `let`-bound variable, followed by the translation of its body.

In F# a sequence of expressions is itself an expression, while in Python it is a statement. We translate a sequence of expressions with a sequence

of statements. However, if the original expression was used in a context in which a statement is not allowed, we have to provide a way to refer to the value of the expression. Moreover, as we saw before a `let` construct is translated in a sequence of statements and expressions. For example, in the translation of

```
let mutable x = 3 in
  let z = x <- x+1, x in
     z
```

we cannot assign to the variable `z` a sequence of expressions, since

```
def z := ( x <- x+1; x )
```

is not a correct definition. To overcome this problem, in the translation we define a (new) variable which contains the value of the expression. In the previous case,

```
x <- x+1; x
```

is translated to

```
x <- x+1; def z1 := x;
```

indicating that `z1` holds the value of the expression. The translation of the whole expression is

```
def x := 3; x <- x+1; def z1 := x; def z := z1; z
```

To make the translation uniform we always define a new variable, holding the value of the expression, also for the cases in which the translated expression is already an expression. For example, even though it would not be necessary, also the translation of expressions `3`, `x`, `x+1`, generates a new variable. So the formal translation would be

```
def x1 := 3; def x := x1;
def x2 := x; def x3 := 1; def x4 := x2 + x3; x <- x4;
def x5 := x; def z = x5;
def z1 = z; z1
```

where all the variable followed by a digit are generated by the translation. The first line translate `x = 3`, the second `x <- x+1`, the third `z = (x <- x+1, x)` and the last one the body of the inner `let`, i.e., the expression `z`.

We could avoid the generation of the unneeded variables by giving two translations, one that produces an expression and another that produces a sequence, as in our previous work [14]. However, the uniformity of the

translation simplifies the formal proof of correctness, which is the aim of this paper.

4.1 Formal Definition of the Translation

In the following we give the translations of an F# expression, e, into IL. The translation is parametric in the sets of variables I and M, which are the immutable (I) and the mutable (M) variables defined in the context of e. The translation produces a sequence of IL statements and/or expressions, se, and a variable, z, which holds the value of the translation of the expression e. Tracing mutable and immutable variables is needed for the proof of correctness, but it is avoided in the implementation.

Definition 4 *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 function $[\![e]\!]^{I,M}$ is defined as follows:*

1. *if e is n or* `tr` *or* `fls` *or x or l, then $[\![e]\!]^{I,M} = (\,\texttt{def}\ z{:=}e, z\,)$ where z is a fresh variable;*
2. $[\![\texttt{fun}\ x{:}T \texttt{->} e']\!]^{I,M} = (\,\texttt{def}\ z{:=}\texttt{fun}\ x\texttt{->}\{se\}, z\,)$ *where z is a fresh variable, and $[\![e']\!]^{I\cup\{x\},M}{=}(\,se, z'\,)$;*
3. $[\![e_1{+}e_2]\!]^{I,M} = (\,se_1; se_2; \texttt{def}\ z{:=}(z_1{+}z_2), z\,)$ *where z is a fresh variable, and $[\![e_i]\!]^{I,M} = (\,se_i, z_i\,)$ $(1 \le i \le 2)$;*
4. $[\![e_1\ e_2]\!]^{I,M} = (\,se_1; se_2; \texttt{def}\ z{:=}(z_1\ z_2), z\,)$ *where z is a fresh variable, and $[\![e_i]\!]^{I,M} = (\,se_i, z_i\,)$ $(1 \le i \le 2)$;*
5. $[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]^{I,M} =$
   $(\,se_1; \texttt{def}\ z{:=}\_; \texttt{if}\ z_1\ \texttt{then}\ \{se_2; z\texttt{<-}z_2\}\ \texttt{else}\ \{se_3; z\texttt{<-}z_3\}, z\,)$ *where z is a fresh variable, and $[\![e_i]\!]^{I,M} = (\,se_i, z_i\,)$ $(1 \le i \le 3)$;*
6. $[\![e_1, e_2]\!]^{I,M} = (\,se_1; se_2, z_2\,)$ *where z is a fresh variable, and $[\![e_i]\!]^{I,M}{=}(\,se_i, z_i\,)$ $(1 \le i \le 2)$;*
7. (a) *if $x \notin I \cup M$, then*
   $[\![\texttt{let}\ x{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M} = (\,se_1; \texttt{def}\ x{:=}z_1; se_2, z_2\,)$
   where $[\![e_1]\!]^{I,M}{=}(\,se_1, z_1\,)$, and $[\![e_2]\!]^{I\cup\{x\},M}{=}(\,se_2, z_2\,)$;
   (b) *if $x \in I \cup M$, then*
   $[\![\texttt{let}\ x{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M} = [\![\texttt{let}\ w{:}T{=}e_1\ \texttt{in}\ (e_2[x := w])]\!]^{I,M}$ *where w is a fresh variable;*

8. (a) *if* $y \notin I \cup M$, *then*
$[\![\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M} = (\ se_1; \texttt{def}\ x{:=}z_1; se_2, z_2\ )$ *where*
$[\![e_1]\!]^{I,M}{=}(\ se_1, z_1\ )$, *and* $[\![e_2]\!]^{I,M\cup\{y\}}{=}(\ se_2, z_2\ )$;

   (b) *if* $y \in I \cup M$, *then*
$[\![\texttt{let mutable}\ y{:}T{=}e_1\ \texttt{in}\ e_2]\!]^{I,M} =$
$[\![\texttt{let mutable}\ w{:}T{=}e_1\ \texttt{in}\ (e_2[y := w])]\!]^{I,M}$ *where* w *is a fresh variable;*

9. (a) *if* $\{\overline{w}\} \cap (I \cup M) = \emptyset$, *then*
$[\![\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ e]\!]^{I,M} = (\ se_1; \cdots; se_m; \texttt{def}\ \overline{w}{:=}\overline{z}; se, z_0\ )$
where $[\![F_j]\!]^{I\cup\{\overline{w}\},M} = (\ se_j, z_j\ )$ $(1 \le j \le m)$, *and* $[\![e]\!]^{I\cup\{\overline{w}\},M}{=}(\ se, z_0\ )$;

   (b) *if* $\{\overline{w}\} \cap (I \cup M) \neq \emptyset$, *then*
$[\![\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{v}\ \texttt{in}\ e]\!]^{I,M} =$
$[\![\texttt{let rec}\ \overline{z}{:}\overline{T}{=}(\overline{v}[\overline{w} := \overline{w}'])\ \texttt{in}\ (e[\overline{w} := \overline{w}'])]\!]^{I,M}$ *where* $\overline{w}'$ *are fresh variables;*

10. $[\![y{<}{-}e']\!]^{I,M} = (\ se; y{<}{-}z, z\ )$ *where* $[\![e']\!]^{I,M}{=}(\ se, z\ )$;

11. $[\![\texttt{raise}\ e']\!]^{I,M} = (\ se; \texttt{raise}\ z, z\ )$ *where* $[\![e']\!]^{I,M}{=}(\ se, z\ )$;

12. $[\![\texttt{try}\ e_1\ \texttt{with}\ x\ {-}{>}\ e_2]\!]^{I,M} =$
$(\ \texttt{def}\ z{:=}\_; \texttt{try}\ \{se_1; z{<}{-}z_1\}\ \texttt{catch}(\ x\ )\ \{se_2; z{<}{-}z_2\}, z\ )$ *where* z *is a fresh variable, and* $[\![e_i]\!]^{I,M} = (\ se_i, z_i\ )$ $(1 \le i \le 2)$.

Translation of constants, locations and variables is the identity, see clause 1 of the previous definition. The variable returned is initialized to the expression. The translation of functions, clause 2, produces a function whose body is the translation of the body into an IL sequence of the original function. In the translation of the body of the function, e', the variable x is added to the set of free immutable variables of the context of e'. The variable returned is initialized to the translation of the function. For sums and applications, clauses 3 and 4, we first translate e_1 and e_2, getting the sequence of IL statements se_1 and se_2 and the variables z_1 and z_2, that can be used to refer to the value of the expressions e_1 and e_2. In the sequence produced we first have the evaluation of se_1, then se_2, and finally the definition of a new variable z, initialized to the sum (respectively application) of z_1 and z_2. So, the translation reflects the left-to-right evaluation of the constructs. For the if construct, clause 5, the translations of the condition and then/else branches produce the sequences of IL statements se_i and the variables z_i $(1 \le i \le 3)$. The translation is then defined by the evaluation of

the condition se_1, followed by the definition of a new variable z, to which it is assigned the result of the evaluation of the then/else branches. Since the branches are blocks, to be visible the variable has to be declared outside the blocks. We use _ to indicate any default value. The translation of the then/else branches is given by the sequence of the `IL` statements followed by the assignment of their resulting value to the variable z. The translation of a sequence, clause 6, is the sequence of the sequences of statements or expressions which are the translations of its subexpressions. The result is available through the variable z_2, which is the variable holding the result of the evaluation of the second expression.

The translations of the `let` constructs, clauses 7, 8, and 9, produce a sequence in which first the definitions of the variables are 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 whether the variables defined are or are not already present in the context.
In case the variable x is not defined in the context. The translations of the `let` and `let mutable` constructs, clauses 7 and 8 produce a sequence starting with the translation of the expression which is assigned to the local variable x. Then, there is the definition of the local variable x to which it is assigned the result of the evaluation of the expression e_1 which is referred to by the variable z_1, followed by the translation of the body of the construct e_2. For the translation of e_2 the variable x is added to the immutable or mutable variables of the context depending on the fact that x is introduced with the `let` or `let mutable` construct. The variable that holds the result of the construct is the same as the one holding the result of the translation of e_2.
If the variable is already present in the context (either as immutable or mutable) the `let` or `let mutable` construct is α renamed substituting x with a fresh variable w. This ensures that w is not in $I \cup M$ and does not occur in the `let` expression.
The translation of the `letrec` construct, clause 9, 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 $\overline{F}$.

The translation of assignment, clause 10, produces an assignment statement in which the variable is assigned the result of the translation of the expression on the right-hand-side of the assignment expression.

The translation of raising an exception, clause 11, is the sequence of IL statements translation of the expression associated to the exception, followed by the corresponding IL construct, with associated the variable that refers to the value of the expression. The same variable holds the result of the raise construct.

The translation of the try-with construct, clause 12, produces the corresponding IL construct. The translations of the body of the try-with and the body of the catcher produce the sequences of IL statements se_i and the variables z_i ($1 \leq i \leq 2$). However, since the result of the evaluation could be either one of them and the variables z_i are local, to refer to the value of the try-with expression we need to declare a new variable z, outside the blocks (similar to the translation of the if construct). Note that, if the expression e_1 raises an exception the assignment $z \leftarrow z_1$ will not be executed.

The correctness of the translation of the `let` constructs relies on the fact that the renaming of variables used does not change the operational semantics of expressions. In the following lemma 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 5 *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.

4.2 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 block bl. We want to show that the evaluation in F# of e has the same result of the evaluation of bl in IL. That is if e evaluates to a primitive value, v, bl evaluates to the same v, and if the evaluation of e raises an exception, then also bl raises an exception. Moreover, if the evaluation of e diverges, then bl cannot either converge to a value or raise an exception.

Given an F# program e, $e \Downarrow v$ and $e \Downarrow \mathsf{E}(v)$ are abbreviations for $\langle e \mid [] \mid [] \rangle \Downarrow \langle v \mid \rho_* \rangle$ and $\langle e \mid [] \mid [] \rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_* \rangle$ for some ρ_*. Similarly for IL blocks bl.

Theorem 3 (Correctness) *Let e be an* F# *program, and let* $[\![e]\!]^{\emptyset,\emptyset}=(\, se, z\,)$. *Then*

- $e \Downarrow v$ *for some* v *either integer or boolean value if and only if* $\{\, se; z\,\} \Downarrow_{bl} v$, *and*
- $e \Downarrow \mathsf{E}(v)$ *for some* v *if and only if* $\{\, se; z\,\} \Downarrow_{bl} \mathsf{E}(v')$ *for some* v'.

The proof of the theorem relies on Lemmas 7 and 8 and will be given at the end of the current section.

To prove the result we define a *translation relation* 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 7; 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 8. 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, we introduce new variables to hold the result of expressions, see Definition 4. 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 the new variables introduced. We assume that the names of the locations in ρ^M are equal to the corresponding ones in the F# store. Looking at the evaluation rules of Fig. 8, 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 (except for the case of the variable introduced for the translation of the if expression, where at the end of the chosen branch it will be assigned).

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 and the variables generated by the translation are translated in IL variables, which are, at runtime, allocated in the store. 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 value ($\texttt{fun}\ x{:}T\texttt{->}e, \sigma$) and its IL translation, we have to establish equivalence between the F# values associated to variables in the definition stack σ, and the IL value contained in the location of the IL store corresponding. Therefore, we give a coinductive definition.

Definition 5 *The* translation relation between F# values and well-formed IL configurations*, dubbed* $\cong$*, is defined by*

1. $n \cong \langle n \mid \rho \rangle$, $\texttt{tr} \cong \langle \texttt{tr} \mid \rho \rangle$, *and* $\texttt{fls} \cong \langle \texttt{fls} \mid \rho \rangle$ *for all* ρ.
2. $(\texttt{fun}\ x{:}T\texttt{->}e, \sigma) \cong \langle (\texttt{fun}\ x\texttt{->}\{se[\overline{x} := \overline{l}]\}) \mid \rho \rangle$ *for some* $\{\overline{l}\} \subseteq dom(\rho)$, *where* $dom(\sigma) = \{\overline{x}\}$, *and*
   (a) $[\![e]\!]^{dom(\sigma)\cup\{x\},\{\overline{y}\}} = (\, se, z \,)$ *for some* $\overline{y}$, *and* z, *and*
   (b) *for all* i, $1 \le i \le n$, *let* $x_i{:}T_i \mapsto v_i \in \sigma$, *we have that* $v_i \cong \langle \rho(l_i) \mid \rho \rangle$.
3. $(\texttt{let rec}\ \overline{w}{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_i, \sigma) \cong \langle v_i'[\overline{w}\,\overline{x} := \overline{l}^{\,w}\,\overline{l}] \mid \rho \rangle$ *for some* $\{\overline{l}^{\,w}, \overline{l}\} \subseteq dom(\rho)$ *where* $dom(\sigma) = \{\overline{x}\}$, *and*
   - *for all* j, $1 \le j \le m$,
     (a) $v_j' = \texttt{fun}\ x\texttt{->}\{se_j\}$ *where* $F_j = \texttt{fun}\ x{:}T_j'\texttt{->}e_j$, *and* $[\![e_j]\!]^{\{\overline{w},x\}\cup dom(\sigma),\{\overline{y}\}} = (\, se_j, z_j \,)$ *for some* $\overline{y}$, *and* z_j, *and*
     (b) $\rho(l_j^w) = v_j'[\overline{w}\,\overline{x} := \overline{l}^{\,w}\,\overline{l}]$, *and*
   - *for all* p, $1 \le p \le n$, *let* $x_p{:}T_p \mapsto v_p \in \sigma$, *we have that* $v_p \cong \langle \rho(l_p) \mid \rho \rangle$.

Primitive F# values are related to IL configurations whose first component is the same primitive value. For F# functions, ($\texttt{fun}\ x{:}T\texttt{->}e, \sigma$), we require that the first component of the IL configuration be the translation of $\texttt{fun}\ x{:}T\texttt{->}e$, in which the free variables of the body (a subset of $dom(\sigma)$) are replaced with IL locations which are associated in the store to values that are translation equivalent to the associated value in σ. For recursive functions we also must have that, in the IL store, the values associated

to the locations corresponding to the names, $\overline{w}$, of the recursively defined functions in $\overline{F}$, are the translation of the functions in $\overline{F}$.

The following lemma asserts that, 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 6 *If for all i, $1 \leq i \leq m$, we have* $(\texttt{let rec } \overline{w}{:}\overline{T}{=}\overline{F} \texttt{ in } F_i, \sigma) \cong \langle v_i'[\overline{x}\,\overline{w} := \overline{l}^x\,\overline{l}^w] \mid \rho' \rangle$*, then for all j, $1 \leq j \leq m$, we have*

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

Proof Directly from Definition 5. □

We, now, define a *translation relation* between F# configuration, $\langle e \mid \sigma \mid \rho \rangle$, and IL configuration, $\langle se \mid \rho' \rangle$, that holds if se is the translation of e, and the values of the mutable variables of e in the store ρ are in the translation relation of Definition 5 with the values of the corresponding variables in ρ'.

Definition 6 *The F# configuration $\langle e \mid \sigma \mid \rho \rangle$ and the IL configuration $\langle se \mid \rho' \rangle$ are* translation related*, written $\langle e \mid \sigma \mid \rho \rangle \approx \langle se \mid \rho' \rangle$, w.r.t. the location environment $\Sigma = \overline{l}{:}\overline{T}$, the sequences of disjoint variables $\overline{x}$, $\overline{y}$, $\overline{w}$, z, the sequences of disjoint locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^w$, l_z, and the F# expression e° if:*

1. $\Sigma \models \langle e \mid \sigma \mid \rho \rangle\diamond$, $\langle se \mid \rho' \rangle$ *is well-formed,* $\{\overline{x}\} = dom(\sigma)$, $\{\overline{w}\} = def(se) - \{z\}$, *and* $\overline{l}^y{:}\overline{T}^y \subseteq \Sigma$,
2. $e = e^\circ[\overline{y} := \overline{l}^y]$ *where* $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ *for some* T,
3. $se = se^\circ[\overline{x}\,\overline{y}\,\overline{w}\,z := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,l_z]$ *where* $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\,se^\circ, z\,)$,
4. $\rho' = \rho^I + \rho^M$ *for some* ρ^I *and* ρ^M *such that* $dom(\rho) = dom(\rho^M)$, *and* $\{\overline{l}^x, \overline{l}^w, l_z\} \subseteq dom(\rho^I)$,
5. *for all* i, $1 \leq i \leq n$, *let* $x_i{:}T_i{\mapsto}v_i \in \sigma$, *we have that* $v_i \cong \langle \rho'(l_i^x) \mid \rho' \rangle$, *and*
6. *for all* $l \in dom(\rho)$ *we have that* $\rho(l) \cong \langle \rho'(l) \mid \rho' \rangle$.

Theorem 3 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 7 *Let* $\langle e \mid \sigma \mid \rho\rangle \approx \langle se \mid \rho'\rangle$ *w.r.t.* $\Sigma = \overline{l}{:}\overline{T}$*, the variables* $\overline{x}$*,* $\overline{y}$*,* $\overline{w}$*,* z*, the locations* $\overline{l}^x$*,* $\overline{l}^y$*,* $\overline{l}^w$*,* l_z*, and the* F# *expression* e°*.*

- *If* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$*, then* $\langle se \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$ *where*
  - (*A*) $v \cong \langle v' \mid \rho'_*\rangle$*, and* $v' = \rho'_*(l_z)$*,*
  - (*B*) *for all* $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ *we have that* $\rho'(l) = \rho'_*(l)$*,*
  - (*C*) *for all* $l \in dom(\rho_*)$ *we have that* $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$*.*
- *If* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_*\rangle$*, then* $\langle se \mid \rho'\rangle \Downarrow_{sq}\langle \mathsf{E}(v') \mid \rho'_*\rangle$*, and properties (A), (B), and (C) hold.*

Proof The proof is given in Appendix C. □

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 statements or expressions does not converge to a value.

Lemma 8 *Let* $\langle e \mid \sigma \mid \rho\rangle \approx \langle se \mid \rho'\rangle$ *w.r.t.* $\Sigma = \overline{l}{:}\overline{T}$*, the variables* $\overline{x}$*,* $\overline{y}$*,* $\overline{w}$*,* z*, the locations* $\overline{l}^x$*,* $\overline{l}^y$*,* $\overline{l}^w$*,* l_z*, and the* F# *expression* e°*. If* $\langle se \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$ *or* $\langle se \mid \rho'\rangle \Downarrow_{sq}\langle \mathsf{E}(v') \mid \rho'_*\rangle$*, then* $\langle e \mid \sigma \mid \rho\rangle \longrightarrow BS\langle v \mid \rho_*\rangle$ *or* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_*\rangle$ *for some* v *and* ρ_**.*

Proof By case analysis on the shape of e and then induction on the derivation of $\langle se \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$.
The base cases of $e = x$, or e=n, or e=`tr`, or e=`fls`, or e=`fun` $x{:}T'$->e_1, or $e = l$ are obvious from Definition 6.2 and 3.

For the structured expressions we only show the proof for application. The others are simpler.

Let $\underline{e{=}e_1\ e_2}$ for some e_1 and e_2. Since $e = e^\circ[\overline{y} := \overline{l}^y]$, then $e^\circ{=}e_1^\circ\ e_2^\circ$ where $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$. From Definition 4.4 we get that $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se^\circ, z\ )$ where $se^\circ = se_1^\circ; se_2^\circ;$ `def` $z{:=}z_1\ z_2$, $[\![e_i^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se_i^\circ, z_i\ )$ $(1 \le i \le 2)$, and z is a fresh variable (so cannot occur in se_i°). From Definition 6.3 we have that se is

$$se_1; se_2; \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2})$$

where $se_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{w}^i\, z_i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{w_i}\, l_{z_i}]$, and $\{\overline{l}^x, \overline{l}^y\,\overline{l}^w, l_z\} \subseteq dom(\rho')$. Moreover $def(se^\circ) = def(se_1^\circ) \cup def(se_2^\circ) \cup \{z\}$, $def(se_1^\circ) \cap def(se_2^\circ) = \emptyset$, and $\{\overline{w}^i\} = def(se_i^\circ) - \{z_i\}$.
Assume that $\underline{\langle se \mid \rho'\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle}$. Then

- $\langle se_1 \mid \rho'\rangle \Downarrow_{sq} \langle v'_1 \mid \rho'_1\rangle$,
- $\langle se_2 \mid \rho'_1\rangle \Downarrow_{sq} \langle v'_2 \mid \rho'_2\rangle$, and
- $\langle \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho'_2\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$.

From the properties of se_1 and e_1 we derive that $\langle e_1 \mid \sigma \mid \rho\rangle \approx \langle se_1 \mid \rho'\rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}^1$, z_1, the locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^{w_1}$, l_{z_1}, and the F# expression e_1°. Applying the inductive hypothesis to $\langle se_1 \mid \rho'\rangle \Downarrow_{sq} \langle v'_1 \mid \rho'_1\rangle$ we have that $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$ for some v_1 and ρ_1. From Lemma 7 we derive

(A_1) $v_1 \cong \langle v'_1 \mid \rho'_1\rangle$, $v'_1 = \rho'_1(l_{z_1})$,

(A_2) $l \notin \{\overline{l}^y, \overline{l}^{w_1}, l_{z_1}\}$ implies $\rho'(l) = \rho'_1(l)$, and

(A_3) $l \in dom(\rho_1)$ implies $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho'_1\rangle$.

Consider the configurations $\langle e_2 \mid \sigma \mid \rho_1\rangle$ and $\langle se_2 \mid \rho'_1\rangle$. From ($A_2$), ($A_3$) and the properties of se_2 and e_2 we derive that

$$\langle e_2 \mid \sigma \mid \rho_1\rangle \approx \langle se_2 \mid \rho'_1\rangle. \tag{1}$$

Applying the inductive hypothesis to $\langle se_2 \mid \rho'_1\rangle \Downarrow_{sq} \langle v'_2 \mid \rho'_2\rangle$ we have that $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_2 \mid \rho_2\rangle$ for some v_2 and ρ_2. From Lemma 7 we derive

(A'_1) $v_2 \cong \langle v'_2 \mid \rho'_2\rangle$, $v'_2 = \rho'_2(l_{z_2})$,

(A'_2) $l \notin \{\overline{l}^y, \overline{l}^{w_2}, l_{z_2}\}$ implies $\rho'_1(l) = \rho'_2(l)$, and

(A'_3) $l \in dom(\rho_2)$ implies $\rho_2(l) \cong \langle \rho'_2(l) \mid \rho'_2\rangle$.

From $\langle \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho'_2\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ and the evaluation rules of Fig. 8 we have that:

$$\langle \{se'\}[x := l_x] \mid \rho'_2[l_x \mapsto v'_2]\rangle \Downarrow_{bl} \langle v' \mid \rho_*\rangle$$

where $l_x \notin dom(\rho'_2)$, and $v'_1 = \texttt{fun}\ x{\to}\{se'\}$. Therefore, from the evaluation rule for blocks we get

$$\langle se'[x\,\overline{w}' := l_x\,\overline{l}^{w'}] \mid \rho'_2[l_x \mapsto v'_2\,\overline{l}^{w'} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v' \mid \rho_*\rangle$$

where $\{\overline{w}'\} = def(se')$, and $\{\overline{l}^{w'}\} \cap dom(\rho_2') = \emptyset$. From (A_1) and Definition 5, for some e', T', and σ' we have that

(P_1) $v_1 = (\texttt{fun}\ x{:}T'{\to}e', \sigma')$,

(P_2) $v_1' = \texttt{fun}\ x{\to}\{se''[\overline{x}' := \overline{l}^{x'}]\}$ where $\{\overline{x}'\} = dom(\sigma')$, and $\{\overline{l}^{x'}\} \subseteq dom(\rho_1')$,

(P_3) $[\![e']\!]^{dom(\sigma')\cup\{x\},\{\overline{y}'\}} = (\ se'', z'\ )$ for some $\overline{y}'$, and z', and

(P_4) for all i, $1 \le i \le n$, if $x_i'{:}T_i' {\mapsto} v_i'' \in \sigma'$, then $v_i'' \cong \langle \rho_1'(l_i^{x'}) \mid \rho_1' \rangle$.

Consider the configurations $\langle e' \mid \sigma'[x{:}T' \mapsto v_2] \mid \rho_2 \rangle$ and $\langle se'[x\ \overline{w}' := l_x\ \overline{l}^{w'}] \mid \rho_2'[l_x \mapsto v_2', \overline{l}^{w'} \mapsto \overline{?}] \rangle$. From (P_3) we derive that conditions 2 and 3 of Definition 6 are verified. From (P_4) and (A_2') we have

(P_4') for all i, $1 \le i \le n$, if $x_i'{:}T_i' {\mapsto} v_i'' \in \sigma'$, then $v_i'' \cong \langle \rho_2'(l_i^{x'}) \mid \rho_2' \rangle$.

So condition 5 of Definition 6 holds. Moreover, from (A_3') condition 6 holds. Therefore

$$\langle e' \mid \sigma'[x{:}T' \mapsto v_2] \mid \rho_2 \rangle \approx \langle se'[x\ \overline{w}' := l_x\ \overline{l}^{w'}] \mid \rho_2'[l_x \mapsto v_2', \overline{l}^{w'} \mapsto \overline{?}] \rangle. \quad (2)$$

By induction hypothesis on $\langle se'[x\ \overline{w}' := l_x\ \overline{l}^{w'}] \mid \rho_2'[l_x \mapsto v_2'\ \overline{l}^{w'} \mapsto \overline{?}] \rangle \Downarrow_{sq} \langle v' \mid \rho_* \rangle$ we have that $\langle e' \mid \sigma'[x{:}T' \mapsto v_2] \mid \rho_2 \rangle \Downarrow \langle v \mid \rho_* \rangle$, for some v and ρ_*. Therefore, from rule (App-F) of Fig. 3 we have that $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$, which proves the result.
Assume that $\underline{\langle se \mid \rho' \rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho_*' \rangle}$. Then, by the propagation rule (PrSeq) of Fig. 9, we have

(C_1) $\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho_*' \rangle$, or

(C_2) $\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle v_1' \mid \rho_1' \rangle$ and $\langle se_2 \mid \rho_1' \rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho_*' \rangle$, or

(C_3) $\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle v_1' \mid \rho_1' \rangle$ and $\langle se_2 \mid \rho_1' \rangle \Downarrow_{sq} \langle v_2' \mid \rho_2' \rangle$ and $\langle \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho_2' \rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho_*' \rangle$.

In all three cases the result derives by induction hypothesis and application of rule (PrApp-F) of Fig. 4. To apply the inductive hypotheses in cases (C_2) and (C_3) we need the equivalences (1) and (2), respectively. Such equivalences are proved as for the case of convergence to a value. □

Proof of Theorem 3 (correctness):
Let e be an F# program, then for some T we have that $\emptyset \mid \emptyset \vdash e : T$ and so $\emptyset \models \langle e \mid \emptyset \mid \emptyset \rangle \diamond$. From $[\![e]\!]^{\emptyset,\emptyset}=(\, se, z\,)$ we have that $FV(se) = \emptyset$, and se does not contain any location. Consider the configuration $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle$ where $se' = se[\overline{w}\, z := \overline{l}^{\,w}\, l_z]$. The configuration is well-formed and it is translation related to $\langle e \mid [\,] \mid [\,]\rangle$ w.r.t. the empty location environment, the empty sets for variables $\overline{x}$ and $\overline{y}$, the sets of variables $\overline{w}$ and z, and the expression e.

Let $\underline{\langle e \mid [\,] \mid [\,]\rangle \Downarrow \langle v \mid \rho_*\rangle}$ for some ρ_*, and let v be either integer or boolean value. Then, by Lemma 7, we derive $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ where $v \cong \langle v' \mid \rho'_*\rangle$ and $v' = \rho'_*(l_z)$. Since v is of primitive type we have that $v' = v$. From $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ and the evaluation rule (Seq) and (Loc) of Fig. 8 we get $\langle se'; l_z \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v \mid \rho'_*\rangle$. Finally, from evaluation rule (Block) we have that $\langle \{\, se; z\,\} \mid [\,]\rangle \Downarrow_{bl} \langle v \mid \rho'_*\rangle$.
On the other side, let $\langle \{\, se; z\,\} \mid [\,]\rangle \Downarrow_{bl} \langle v' \mid \rho'_*\rangle$. From the evaluation rules of Fig. 8 we have that $\langle se'; l_z \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$. Therefore $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$. By Lemma 8 we derive that $\langle e \mid [\,] \mid \Downarrow\rangle \langle v \mid \rho_*\rangle$, and Lemma 7 implies $v' = v$.

Let $\underline{\langle e \mid [\,] \mid [\,]\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_*\rangle}$ for some ρ_* and v. Then, by Lemma 7, we derive $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. From rule (PrSeq) of Fig. 9 and a similar propagation rule of exceptions for blocks we have $\langle \{\, se; z\,\} \mid [\,]\rangle [\,] \Downarrow_{bl} \langle \mathsf{E}(v') \mid \rho'_*\rangle$.
On the other side, let $\langle \{\, se; z\,\} \mid [\,]\rangle \Downarrow_{bl} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. From the evaluation rules of Fig. 9 we have that $\langle se'; l_z \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. From rule (PrSeq) of Fig. 9 we have that either $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$, or $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle v'' \mid \rho'_*\rangle$ and $\langle l_z \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. However, it cannot be the case that $\langle l_z \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. So $\langle se' \mid [l_z \mapsto ?\, \overline{l}^{\,w} \mapsto \overline{?}]\rangle \Downarrow_{sq} \langle \mathsf{E}(v') \mid \rho'_*\rangle$. By Lemma 8, we have that $\langle e \mid [\,] \mid [\,]\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_*\rangle$ for some v and ρ_*. This proves the result. □

5 Implementation

The compiler was implemented in F#. The implementation relies on the .NET reflective features and the F# *code quotations*. These allow one to reason about the source code and through transformation processes generate target language code. Following is 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 defined in terms of discriminated unions that describe the main IL syntactic categories such as *statements* and *expressions*. This is a traditional approach adopted by most functional implementations, e.g., see [4]. 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

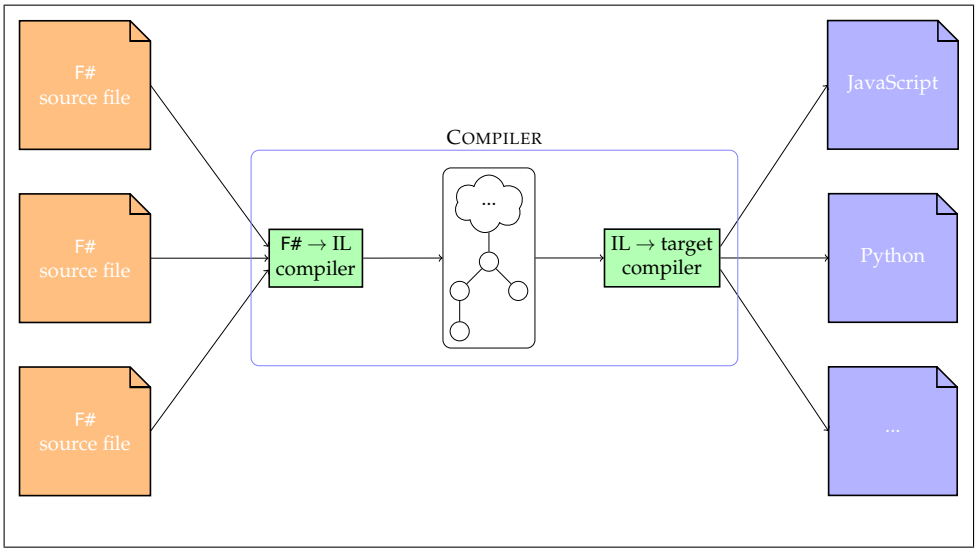

Figure 10: Compiler architecture

The compiler's architecture is shown in Fig. 10. The translation process is divided into two phases. First, the F# → IL component translates the F# source code into its IL representation. Then, the IL → target language compiler translates the IL to the chosen target language. The translation is internally delegated to the target-specific driver. The compiler currently supports translation to JavaScript, however, a previous proof-of-concept implementation supported also Python. A driver is an implementation of an F# interface defining methods that take an IL tree as input and return the target language code. During the translation process, the IL tree is visited

and at each tree node a specific translation method, defined in a driver, is executed.

The IL tree is generated from the 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. 11 shows an example of Quotations in action. The constant ast is bound to the tree representation of the quoted expression. In other words, instead of evaluating the code between <@@ @@>, the compiler returns an abstract syntax tree of the enclosed expression. The core F# library provides the ReflectedDefinitionAttribute (RefDefAtt for short) which can be used to annotate functions, methods or, in F# 3.x, modules, to obtain the relative AST representation.

```
> open Microsoft.FSharp.Quotations

> let ast = <@@ let add x y = x + y in add 2 3 @@>

val ast : Expr<int> =
 Let (add, Lambda (x, Lambda (y, Call(None, op_Addition, [x, y]))),
 Application (Application (add, Value (2)), Value (3)))
```

Figure 11: Example of F# quotations

To obtain the AST of all functions or methods annotated with RefDefAtt we proceed as follows:

1. at runtime, we compile the desired modules to obtain a compiled assembly;
2. by using the .NET reflection, we extract from the compiled assembly all types and definitions;
3. we filter out definitions that are not annotated with RefDefAtt;
4. for each function/method annotated with RefDefAtt, we take the relative AST.

Once we obtain the F# ASTs, we use pattern matching to recursively traverse the trees and gradually build the corresponding IL tree. The Quotation library provides us with a rich set of active patterns for working with F# ASTs.

5.3 Extensions

We have developed several extensions and are currently experimenting with several 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 [20], 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://app.assembla.com/spaces/ironcat

6 Comparisons with Other Work

The compilation process of high-order functional languages is often preceded by continuation-passing style (CPS) transformations to generate an intermediate representation of the source program, [3]. The intermediate form, originally expressed in the source functional language, was designed to allow code optimization. More specialized intermediate languages, such as the A-normal forms, [9], first-order intermediate languages, [10], and CPS languages, [17], were introduced to reduce the size of the intermediate representation of the code, and perform better optimizations. Our intermediate language is designed to be close to the target of our compilation, which is scripting languages such as JavaScript and Python. Our translation is reminiscent of the compilation with continuation (CPS), in that it produces not only the `IL` code but also a variable which holds the value of the translated expression.

Compiler correctness has been studied extensively, see [7]. In [5], a logical relation is defined between a simply-typed functional language with recursion and low-level code of a SECD machine. The logical relation connects the denotational semantics of the source language with a small-step operational semantics of the SECD machine. We also use a logical relation, but we define it directly between the big-step semantics of the two languages. In [18] a mechanically verified implementation of ML in low-level code is presented. An intermediate functional language is introduced, to simplify the translation. The proof of correctness uses, as in our case a big-step operational semantics. The only work to our knowledge that proves the correctness of a translation between a statically typed functional language with imperative features and a scripting language (in this case JavaScript) is [11]. In [11] 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 evaluation 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. The use of 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.

Projects similar to ours 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 [8], and FunScript, see [6], are open source F# to JavaScript compilers. They support only translation to JavaScript. FunScript has support for integration with JavaScript code. Websharper, see [16], 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, but rather 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 [20]. 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 [22].

In our previous work, [12, 14], we defined a translation of the same source language to an intermediate language containing a construct for wrapping and executing code outside its definition environment. The current translation, and the proof of correctness, are simpler.

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 [12] and [14]. We have a prototype implementation of the compiler that can be found at the project site [13]. 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 namespaces, 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 in general meta-programming along the lines of work by the authors, see [1] and [2].

Acknowledgements

We thank the referees for their helpful comments, the paper improved greatly due to their suggestions.

References

[1] D. Ancona, P. Giannini, and E. Zucca. Reconciling positional and nominal binding. In *Proceedings of the Sixth Workshop on Intersection Types and Related Systems (ITRS'12), Dubrovnik, Croatia, 29th June 2012*, volume 121 of *EPTCS*, pages 81–93, 2012. doi:10.4204/EPTCS.121.6.

[2] D. Ancona, P. Giannini, and E. Zucca. Incremental Rebinding. In *Proceedings of the 14th Italian Conference on Theoretical Computer Science (ICTCS'13)*. http://www.di.unito.it/~giannini/papers/ICTCS13.pdf, 2013.

[3] A.W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

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

[5] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In Graham Hutton and A.P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 97–108. ACM, 2009. doi:10.1145/1596550.1596567.

[6] Z. Bray, T. Petricek, R. Pickering, and J. Freiwirth. Funscript. http://funscript.info/, February 2013.

[7] M.A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, November 2003. doi:10.1145/966221.966235.

[8] I. Elliot. Pit - F# to JavaScript compiler. http://www.i-programmer.info/news/167-javascript/3678-pit-f-to-javascript-compiler.html, 2012.

[9] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993. doi:10.1145/155090.155113.

[10] M. Fluet and S. Weeks. Contification using dominators. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5*, pages 2–13. ACM, 2001. doi:10.1145/507635.507639.

[11] C. Fournet, N. Swamy, Juan Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384. ACM, 2013. doi:10.1145/2480359.2429114.

[12] P. Giannini and A. Shaqiri. An intermediate language for compilation to scripting languages. In José Cordeiro, David A. Marca, and Marten van Sinderen, editors, *ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies, Reykjavík, Iceland*, pages 92–103. SciTePress, 2013. doi:10.5220/0004588600920103.

[13] P. Giannini and A. Shaqiri. Blue Storm Compiler. https://www.assembla.com/spaces/bluestorm, 2014.

[14] P. Giannini and A. Shaqiri. Compiling functional to scripting languages. *Communications in Computer and Information Science*, 457:114–130, 2014. doi:10.1007/978-3-662-44920-2_8.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

[16] Intellifactory. Websharper 2010 platform. http://websharper.com/, May 2012.

[17] A. Kennedy. Compiling with continuations, continued. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190. ACM, 2007. doi:10.1145/1291151.1291179.

[18] R. Kumar, M.O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

[19] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.

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

[21] J.F. Ranson, H.J. Hamilton, and P.W. L. Fong. A semantics of Python in Isabelle/HOL. Technical Report CS-2008-04, CS Department, University of Regina,Saskatchewan, 2008. URL: http://www.cs.uregina.ca/Research/Techreports/2008-04.pdf.

[22] J. Vouillon and V. Balat. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014. doi:10.1002/spe.2187.

A Proof of Subject Reduction for F#

In this appendix we prove the Subject Reduction lemmas.

Lemma 9 *If* $\Gamma[y{:}T'!] \mid \Sigma \vdash e : T$, $l \notin dom(\Sigma)$, *and* $\models v{:}T'$, *then* $\Gamma \mid \Sigma[l{:}T'] \vdash e[l := v] : T$.

Proof By structural induction on e. □

Lemma 10 (Inversion) *1. If* $\Gamma \mid \Sigma \vdash n : T$, *then* $T = \texttt{int}$. *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* $\Gamma \mid \Sigma \vdash e_i : \texttt{int}$ $(1 \le i \le 2)$.

5. If $\Gamma \mid \Sigma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T$, *then* $\Gamma \mid \Sigma \vdash e : \texttt{bool}$, *and* $\Gamma \mid \Sigma \vdash e_i : T$ $(1 \le i \le 2)$.

6. If $\Gamma \mid \Sigma \vdash \texttt{fun}\ x{:}T_1 \texttt{->} e : T$, *then for some* T_2, Γ_1, *and* Γ_2 *we have that*

(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* $\Gamma[\overline{w}{:}\overline{T}] \mid \Sigma \vdash F_j : T_j$ $(1 \le j \le n)$, *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* $\Sigma(l) = T$.

13. If $\Gamma \mid \Sigma \vdash \texttt{raise}\ e : T$, *then* $\Gamma \mid \Sigma \vdash e : T_{\mathsf{E}}$.

14. If $\Gamma \mid \Sigma \vdash \texttt{try}\ e_1\ \texttt{with}\ x\ \texttt{->}\ e_2 : T$, *then* $\Gamma \mid \Sigma \vdash e_1 : T$, *and* $\Gamma[x{:}T_{\mathsf{E}}] \mid \Sigma \vdash e_2 : T$.

Proof By case analysis in the last rule used in the type derivation of $\Gamma \mid \Sigma \vdash e : T$. $\square$

Proof of Lemma 1 (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, clause 4, we have that

1. $env(\sigma) \mid \Sigma \vdash e : T$ 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, assignment, and the rule for try-catch. The others are similar. Consider the last rule applied in the derivation.

Rule (Var-F) In this case $e{=}x$, and $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle lkp(x,\sigma) \mid \rho \rangle$. Let $x{:}T \mapsto v' \in \sigma$, and let $f_j = \texttt{let rec } \overline{x}{:}\overline{T}{=}\overline{F} \texttt{ in } F_j$ $(1 \leq j \leq n)$. If $lkp(x,\sigma) = v'$, then, from 2, we have that $\models v'{:}T$. Otherwise $v' = (f_k, \sigma)$, and $v = (F_k, \sigma[x_i{:}T_i \mapsto (f_i,\sigma)]_{1 \leq i \leq n})$. From $\models v'{:}T$ we have that $T = T_k$, and $env(\sigma)[\overline{x}{:}\overline{T}] \mid \emptyset \vdash F_k : T_k$. Therefore, from Definition 2, clause 1(c), we derive $\models v{:}T$. 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 get $\Sigma \models \rho$.

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 10, clause 1, we have that if $e = n$, then $T = \texttt{int}$. Moreover, if $e{=}\texttt{tr}$ or $e{=}\texttt{fls}$, then $T = \texttt{bool}$. Therefore, from Definition 2, clauses 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 get $\Sigma \models \rho$.

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 10, clause 6, we derive $T = T_1 \to T_2$, $env(\sigma) = \Gamma_1[\Gamma_2]$, $\Gamma_2[x{:}T] \mid \emptyset \vdash e' : T_2$, and $y{:}T! \notin \Gamma_2$ for all y and T. Since in $env(\sigma)$ there are not $y{:}T!$, we also have $env(\sigma)[x{:}T] \mid \emptyset \vdash e' : T_2$. Therefore, from Definition 2, clause 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 10, clause 3, we have that $T = \Sigma(l)$. From 3, and Definition 2, clause 3, we get $\models \rho(l){:}T$, $\Sigma \models \rho$. If T is a function type, then 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 10, clause 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, we derive $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. 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, clause 1(c), we have that $env(\sigma') \mid \Sigma \vdash \texttt{fun}\ x{:}T_1{\texttt{->}}e_b : T' \to T$. Lemma 10, clause 6, implies $T' = T_1$, and $env(\sigma')[x{:}T_1] \mid \emptyset \vdash e_b : T$. Moreover, we have that $\models \sigma'\diamond$. From $\models v_x{:}T_1$ we have that $\models \sigma'[x{:}T_1 \mapsto v_x]\diamond$, 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 derive $\models v{:}T$, and $\Sigma' \models \rho_*$ for some $\Sigma' \supseteq \Sigma_2$. This proves the result. If T is a function type, then the inductive hypotheses implies that v has the required shape.

Rule (LetMut-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 $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$, $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle \Downarrow \langle v \mid \rho_*\rangle$, and $l_y \notin dom(\rho_1)$.

From 1, and Lemma 10, clause 10, we have that $T' = T_1$ and $\Gamma \mid \Sigma \vdash e_1 : T_1$. So, from 2 and 3 we get $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_y \mid \rho_1\rangle$ we derive $\models v_y{:}T_1$, and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$.

From Lemma 10, clause 10, and $\Sigma_1 \supseteq \Sigma$, we get $\Gamma[y{:}T_1!] \mid \Sigma_1 \vdash e_2 : T$. Since $l_y \notin dom(\rho_1)$, Lemma 9 implies $\Gamma \mid \Sigma_1[l_y{:}T_1] \vdash e_2[y := l_y] : T$. Moreover $\Sigma_1[l_y{:}T_1] \models \rho_1[l_y \mapsto v_y]$. Therefore, from 2, we have that$\Sigma_1[l_y{:}T_1] \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y]\rangle\diamond$. 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 derive that $\models v{:}T$, and $\Sigma' \models \rho_*$ for some $\Sigma' \supseteq \Sigma_1[l_y{:}T']$. This proves the result. If T is a function type, then the inductive hypotheses implies 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 $\langle e_1 \mid \sigma' \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ and $\sigma' = \sigma[w_j{:}T_j \mapsto (f_j, \sigma)]_{1\leq j\leq m}$ where f_j = `let rec` $\overline{x}{:}\overline{T}=\overline{F}$ `in` F_j. From Lemma 10, clause 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 $\Sigma' \supseteq \Sigma$. If T is a function type, then the inductive hypotheses implies 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 10, clause 12, we have that $\Gamma \mid \Sigma \vdash e_1 : T$. Therefore, from 2 and 3 we derive $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. 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 $\Sigma_1 \supseteq \Sigma$.
Lemma 10, clause 12, implies $\Sigma(l) = T$, and $\Sigma_1(l) = T$. From $\models v{:}T$ we get $\Sigma_1 \models \rho_1[l \mapsto v]$.
If T is a function type, then the inductive hypotheses implies that v has the required shape.

Rule (CthVal-F) In this case e=`try` e_1 `with` x -> e_2, and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$. From 1, and Lemma 10, clause 14, we have that $\Gamma \mid \Sigma \vdash e_1 : T$. Therefore, from 2 and 3 we derive $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. 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 $\Sigma_1 \supseteq \Sigma$.
If T is a function type, then the inductive hypotheses implies that v has the required shape.

Rule (CthExc-F) In this case e=`try` e_1 `with` x -> e_2 and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_2\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v_1) \mid \rho_1\rangle$, and $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle$. From 1, and Lemma 10, clause 14, we have that $\Gamma \mid \Sigma \vdash e_1 : T$ and $\Gamma[x{:}T_{\mathsf{E}}] \mid \Sigma \vdash e_2 : T$. Therefore, from 2 and 3 we derive $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. From Lemma 2 we have that $\models v_1{:}T_{\mathsf{E}}$, and $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$. From 2, and $\models v_1{:}T_{\mathsf{E}}$ we derive $\models \sigma[x{:}T_{\mathsf{E}} \mapsto v_1]\diamond$. From $\Gamma[x{:}T_{\mathsf{E}}] \mid \Sigma \vdash e_2 : T$, and $\Sigma_1 \supseteq \Sigma$ we have $\Gamma[x{:}T_{\mathsf{E}}] \mid \Sigma_1 \vdash e_2 : T$. Therefore, applying the induction hypothesis to $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle$ we derive that $\models v{:}T$, and $\Sigma_2 \models \rho_2$ for some $\Sigma_2 \supseteq \Sigma_1$. If T is a function type, then the inductive hypotheses implies that v has the required shape. □

Proof of Lemma 2 (Type Preservation for Exceptions):
Let $\langle e \mid \sigma \mid \rho\rangle$ be such that $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$. From Definition 2, clause 4, we get that

1. $env(\sigma) \mid \Sigma \vdash e : T$ 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 $\Downarrow$. Consider the last rule applied in the derivation.
If the rule is (Throw-F), then e=`raise` e_1 and $\langle$`raise` $e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho'\rangle$. From 1, and Lemma 10, clause 13, we have that $\Gamma \mid \Sigma \vdash e_1 : T_{\mathsf{E}}$. Therefore, from 2 and 3 we derive $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Lemma 1 applied to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ implies that $\models v{:}T_{\mathsf{E}}$, and $\Sigma' \models \rho'$ for some $\Sigma' \supseteq \Sigma$.
All other cases derive by induction hypothesis as for the proof of Lemma 1, by noticing that $env(\sigma) \mid \Sigma \vdash$ `raise` $e' : T$ for all T. □

B Proof of Consistency of Semantics and Progress for F#

In this appendix we prove the Consistency and Progress lemmas. Since we give a coinductive interpretation to the rules of Fig. 5, the proof of the Progress lemma use coinduction.

Proof of Lemma 3 (Consistency):
Consider a configuration $\langle e \mid \sigma \mid \rho\rangle$. By structural induction on 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 only rules of Fig. 3 are applicable.
For the structured expressions we only show the proof for application. The others are similar and simpler.

Let $\underline{e{=}e_1\ e_2}$ for some e_1, e_2. Assume that $\langle e \mid \sigma \mid \rho\rangle\Downarrow$, and rule (App-F) of Fig. 3 was applied. Then

1. $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle$(`fun` $x{:}T'$->e_b, σ') $\mid \rho_1\rangle$,
2. $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$, and
3. $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$.

By inductive hypothesis

a. neither $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v_e) \mid \rho_e\rangle$ nor $\langle e_1 \mid \sigma \mid \rho\rangle \Uparrow$

b. neither $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle \mathsf{E}(v_e) \mid \rho_e\rangle$ nor $\langle e_2 \mid \sigma \mid \rho_1\rangle \Uparrow$, and

c. neither $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle \mathsf{E}(v_e) \mid \rho_e\rangle$ nor $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle \Uparrow$.

Therefore, neither rule (PrApp) of Fig. 4 nor rule (App-$\Uparrow$) of Fig. 5 are applicable.
If $\langle e \mid \sigma \mid \rho\rangle \Downarrow$ and rule (PrApp) of Fig. 4 was applied, then either

i. $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$ or

ii. $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T\texttt{->}e, \sigma') \mid \rho_1\rangle$ and $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$ or

iii. $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle (\texttt{fun}\ x{:}T\texttt{->}e, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v \mid \rho_2\rangle$, and $\langle e \mid \sigma'[x \mapsto v] \mid \rho_2\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho'\rangle$.

By inductive hypothesis, if either one of i., ii. and .iii holds, then neither rule (App-F) of Fig. 3 nor rule (App-$\Uparrow$) of Fig. 5 is applicable. □

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 $e{=}n$, $e{=}\texttt{tr}$, $e{=}\texttt{fls}$, $e{=}\texttt{fun}\ x{:}T_1\texttt{->}e_1$, $e = x$, and $e = l$ we have that $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho\rangle$ for some v.
For the structured expressions we only show the proof for application and let mutable. The others are similar and simpler.

Let $e{=}e_1\ e_2$ for some e_1, e_2. 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'.
By excluded middle, either $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow$ or not.
In the latter case the judgment $\langle e \mid \sigma \mid \rho\rangle \Uparrow$ follows from rule (App-$\Uparrow$) of Fig. 5 and the coinductive hypothesis $\langle e_1 \mid \sigma \mid \rho\rangle \Uparrow$, using the first disjunct of the premises.

Assume that $\langle e_1 \mid \sigma \mid \rho\rangle\Downarrow$. If $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$, from Lemma 1, we get $v_1 = (\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma')$ 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 $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$ or not.
In the latter case the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (App-$\Uparrow$) of Fig. 5, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v_1 \mid \rho_1\rangle$, and the coinductive hypothesis $\langle e_2 \mid \sigma \mid \rho_1\rangle\Uparrow$. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ is false.
If $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$, then, from Lemma 1, 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. 2 we have $env(\sigma'), x{:}T' \mid \emptyset \vdash e_b : T$, and $\models \sigma'\diamond$. Therefore, from $\models v_x{:}T'$ follows that $\models \sigma'[x{:}T' \mapsto v_x]\diamond$, so $\Sigma_2 \models \langle e_b \mid \sigma'[x{:}T' \mapsto v_x] \mid \rho_2\rangle\diamond$.
By excluded middle, either $\langle e_b \mid \sigma'[x{:}T' \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$ or not. In the latter case the judgment $\langle e \mid \sigma \mid \rho\rangle\Uparrow$ follows from rule (App-$\Uparrow$) of Fig. 5, $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle(\texttt{fun}\ x{:}T'\texttt{->}e_b, \sigma') \mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$, and the coinductive hypothesis $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle\Uparrow$. Moreover, $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ 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. 3 we derive $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$.
We can give a similar proof for the case in which the configuration converges using rule (PrApp) of Fig. 4 □

C Proof of Lemma 7

Let $\langle e \mid \sigma \mid \rho\rangle \approx \langle se \mid \rho'\rangle$ *w.r.t.* $\Sigma = \overline{l}{:}\overline{T}$, *the variables* $\overline{x}$, $\overline{y}$, $\overline{w}$, z, *the locations* $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^w$, l_z, *and the* F# *expression* e°. *If* $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$, *then* $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$ *where*

(*A*) $v \cong \langle v' \mid \rho'_*\rangle$, *and* $v' = \rho'_*(l_z)$,

(*B*) $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ *implies* $\rho'(l) = \rho'_*(l)$, *and*

(*C*) $l \in dom(\rho_*)$ *implies* $\rho_*(l) \cong \langle\rho'_*(l) \mid \rho'_*\rangle$.

Proof Let $\langle e \mid \sigma \mid \rho\rangle \approx \langle se \mid \rho'\rangle$. From Definition 6, clauses 5 and 6, we have that

(α) for all i, $1 \le i \le n$, if $x_i{:}T_i \mapsto v_i \in \sigma$, then $v_i \cong \langle\rho'(l_i^x) \mid \rho'\rangle$, and

(β) $l \in dom(\rho)$ implies $\rho(l) \cong \langle \rho'(l) \mid \rho' \rangle$.

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`, `let rec`, assignment, `raise`, and the `try` $-$ `with` in which the exception is caught. The other cases are similar. Consider the last rule of Fig. 3 applied in the derivation.

Rule (Var-F) In this case $e{=}x$ and, from Definition 6, clause 2, for some i such that $x = x_i$ we have that $x{\mapsto}v_i \in \sigma$. From Definition 6, clause 3 and Definition 4, clause 1, we have that se = `def` $l_z{:=}l_i^x$. Let $\langle x \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho \rangle$ where $lkp(x, \sigma) = v$. Applying rule (Def) of Fig. 8 we have that $\langle$`def` $l_z{:=}l_i^x \mid \rho' \rangle \Downarrow_{sq} \langle \rho'(l_i^x) \mid \rho'_* \rangle$ where $\rho'_* = \rho'[l_z \mapsto \rho'(l_i^x)]$. If $v = v_i$, i.e., the value is not a recursively defined function, then, from (α) we have that $v \cong (\rho'_*(l_i^x), \rho'_*)$, and $\rho'_*(l_z) = \rho'_*(l_i^x)$. Therefore (A) holds.
If $v_i = (\texttt{let rec } \overline{w}{:}\overline{T}{=}\overline{F} \texttt{ in } F_k, \sigma)$ for some k, then

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

Since, from (α), we have that $v_i \cong (\rho'(l_i^x), \rho')$, then, from Lemma 6 we also have that $v \cong (\rho'(l_i^x), \rho'_*)$, and $\rho'_*(l_z) = \rho'_*(l_i^x)$. Therefore (A) holds.
Since $\rho_* = \rho$ and $l \in dom(\rho') - \{l_z\}$ implies $\rho'(l) = \rho'_*(l)$, then clauses (B) and (C) hold.

Rule (Fn-Val-F) In this case $e{=}$`fun` $x{:}T{\to}e'$, and $\langle e \mid \sigma \mid \rho \rangle \Downarrow \langle$(`fun` $x{:}T{\to}e', \sigma$) $\mid \rho \rangle$. From Definition 6, clause 2, e° is such that $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ for some T. From rule (TyAbs) of Fig. 2 we have that no $y \in \{\overline{y}\}$ can be free in e°, so $e^\circ = e$. From Definition 4, clause 2, we get $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}}{=}($ `def` $z{:=}$`fun` $x{\to}\{se^\circ\}, z\ )$, where se° is such that $[\![e']\!]^{\{\overline{x},x\},\{\overline{y}\}}{=}(\ se^\circ, z'\ )$ for some z'. From Definition 6, clause 3, we have that $se{=}$`def` $l_z{:=}$`fun` $x{\to}\{se'\}$ where $se' = se^\circ[\overline{x} := \overline{l}^x]$. Applying rule (Def) of Fig. 8 we derive $\langle$`def` $l_z{:=}$`fun` $x{\to}\{se'\} \mid \rho' \rangle \Downarrow_{sq} \langle$`fun` $x{\to}\{se'\} \mid \rho'_* \rangle$ where $\rho'_* = \rho'[l_z \mapsto$ `fun` $x{\to}\{se'\}]$.
Consider the value ($\texttt{fun } x{:}T{\to}e', \sigma$), the configuration $\langle$`fun` $x{\to}\{se'\} \mid \rho'_* \rangle$, and the sequence of locations $\overline{l}^x$. Since $x \notin \{\overline{x}\}$ we have that `fun` $x{\to}\{se'\}$ = (`fun` $x{\to}\{se^\circ\}$)$[\overline{x} := \overline{l}^x]$. From ($\alpha$) and the fact that $\rho'_*(l_i^x) = \rho'(l_i^x)$ $(1 \le i \le n)$ we derive that: for all i, $1 \le i \le n$, $x_i{:}T_i{\mapsto}v_i \in \sigma$ implies $v_i \cong \langle \rho'_*(l_i^x) \mid \rho' \rangle$. Therefore ($\texttt{fun } x{:}T{\to}e', \sigma$) $\cong$

$\langle$`fun` x->$\{se'\} \mid \rho'_*\rangle$, and $\rho'_*(l_z) =$ `fun` x->$\{se'\}$. So (A) holds.
Since $\rho_* = \rho$ and $l \in dom(\rho') - \{l_z\}$ implies $\rho'(l) = \rho'_*(l)$, then clauses (B) and (C) hold.

Rule (Loc-F) In this case $e{=}l$ for some l. From $env(\sigma), \overline{y}{:}\overline{T}^y! \mid \emptyset \vdash e^\circ : T$ we derive that e° may not contain locations. Therefore, for some j, we have that $e^\circ = y_j$, $e = l_j^y$, and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \rho(l_j^y) \mid \rho\rangle$. From Definition 6, clause 3, and Definition 4, clause 1, we have that $se =$ `def` $l_z{:=}l_j^y$. Applying rule (Def) of Fig. 8, with rule (Loc) on the premise, we derive $\langle se \mid \rho'\rangle \Downarrow_{sq} \langle \rho'(l_j^y) \mid \rho'_*\rangle$ where $\rho'_* = \rho'[l_z \mapsto \rho'(l_j^y)]$. From ($\beta$) we get $\rho(l_j^y) \cong (\rho'(l_j^y), \rho')$, and $\rho(l_j^y) \cong (\rho'_*(l_j^y), \rho'_*)$. Therefore, (A) holds.
Since $\rho_* = \rho$ and $l \in dom(\rho') - \{l_z\}$ implies $\rho'(l) = \rho'_*(l)$, then clauses (B) and (C) 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 $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle$(`fun` $x{:}T_1$->e_b, σ') $\mid \rho_1\rangle$, $\langle e_2 \mid \sigma \mid \rho_1\rangle \Downarrow \langle v_x \mid \rho_2\rangle$, and $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2\rangle \Downarrow \langle v \mid \rho_*\rangle$. Since $e = e^\circ[\overline{y} := \overline{l}^y]$ we have that $e^\circ{=}e_1^\circ\ e_2^\circ$, where $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$ $(1 \le i \le 2)$. From Definition 4, clause 4, we derive $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se^\circ, z\ )$, where $se^\circ = se_1^\circ; se_2^\circ;$ `def` $z{:=}z_1\ z_2$, $[\![e_i^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se_i^\circ, z_i\ )$ $(1 \le i \le 2)$, and z is a fresh variable. From Definition 6, clause 3, we have that se is

$$se_1; se_2; \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2})$$

where $se_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{w}^i\ z_i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{w_i}\ l_{z_i}]$, and $\{\overline{l}^x, \overline{l}^y\,\overline{l}^w, l_z\} \subseteq dom(\rho')$. Moreover $def(se^\circ) = def(se_1^\circ) \cup def(se_2^\circ) \cup \{z\}$, $def(se_1^\circ) \cap def(se_2^\circ) = \emptyset$, and $\{\overline{w}^i\} = def(se_i^\circ) - \{z_i\}$.
Consider the configurations $\langle e_1 \mid \sigma \mid \rho\rangle$, and $\langle se_1 \mid \rho'\rangle$. Since e_1 is subterm of e, from $\Sigma \models \langle e \mid \sigma \mid \rho\rangle\diamond$ we get $\Sigma \models \langle e_1 \mid \sigma \mid \rho\rangle\diamond$. Similarly $\langle se_1 \mid \rho'\rangle$ is well-formed. From the previous considerations, (α), and (β) we can see that clauses $2 \div 6$ of Definition 6 hold. Therefore we get

$$\langle e_1 \mid \sigma \mid \rho\rangle \approx \langle se_1 \mid \rho'\rangle$$

w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}^1$, z_1, the locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^{w_1}$, l_{z_1}, and the F# expression e_1°. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle$(`fun` $x{:}T$->e_b, σ') $\mid \rho_1\rangle$ we get

$$\langle se_1 \mid \rho'\rangle \Downarrow_{sq} \langle v'' \mid \rho'_1\rangle. \tag{3}$$

Moreover, the following hold:

(A_1) $(\texttt{fun}\ x{:}T_1\texttt{->}e_b, \sigma') \cong \langle v'' \mid \rho_1' \rangle$ and $v'' = \rho_1'(l_{z_1})$,
(B_1) $l \notin \{\overline{l}^y, \overline{l}^{w_1}, l_{z_1}\}$ implies $\rho'(l) = \rho_1'(l)$, and
(C_1) $l \in dom(\rho_1)$ implies $\rho_1(l) \cong \langle \rho_1'(l) \mid \rho_1' \rangle$.

From Lemma 1, and Definition 2(c) we have that

(D_1) no location l may occur in e_b.

Consider the configurations $\langle e_2 \mid \sigma \mid \rho_1 \rangle$, and $\langle se_2 \mid \rho_1' \rangle$. We want to show that these configurations satisfy the conditions of Definition 6. From Lemma 1 we have that $\Sigma_1 \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$. From clauses 2 and 3 of Definition 6, and typing rule (TyApp) of Fig. 2 we get $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_2 : T_1$ for some T_1. Since $\overline{l}^y{:}\overline{T}^y \subseteq \Sigma_1$ and $\models \sigma\diamond$ we derive $\Sigma_1 \models \langle e_2 \mid \sigma \mid \rho_1 \rangle\diamond$. From Theorem 2 we have that $\langle v'' \mid \rho_1' \rangle$ is well-formed. From $Loc(se_2) \subseteq dom(\rho') \subseteq dom(\rho_1')$ we get that $\langle se_2 \mid \rho_1' \rangle$ is well-formed. Therefore $\langle e \mid \sigma \mid \rho \rangle \approx \langle se \mid \rho' \rangle$, clauses $2 \div 4$ of Definition 6 hold with $\overline{x}$, $\overline{y}$, $\overline{w}^2$, z_2, $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^{w_2}$, l_{z_2}, and e_2°. From (B_1), for all $l \in \{\overline{l}^x\}$ we have that $\rho'(l) = \rho_1'(l)$. So, from (α) we have that:

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

Therefore, from (C_1) we derive $\langle e_2 \mid \sigma \mid \rho_1 \rangle \approx \langle se_2 \mid \rho_1' \rangle$ w.r.t. Σ_1, $\overline{x}$, $\overline{y}$, $\overline{w}^2$, z_2, $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^{w_2}$, l_{z_2}, and e_2°. Applying the inductive hypothesis to $\langle e_2 \mid \sigma \mid \rho_1 \rangle \Downarrow \langle v_x \mid \rho_2 \rangle$ we get

$$\langle se_2 \mid \rho_1' \rangle \Downarrow_{sq} \langle v_x' \mid \rho_2' \rangle \tag{4}$$

where

(A_2) $v_x \cong \langle v_x' \mid \rho_2' \rangle$ and $v_x' = \rho_2'(l_{z_2})$,
(B_2) $l \notin \{\overline{l}^y, \overline{l}^{w_2}, l_{z_2}\}$ implies $\rho_1'(l) = \rho_2'(l)$, and
(C_2) $l \in dom(\rho_2)$ implies $\rho_2(l) \cong \langle \rho_2'(l) \mid \rho_2' \rangle$.

Let $\overline{x}' = dom(\sigma')$, from ($A_1$), and Definition 5 we have that

(P_1) $v'' = (\texttt{fun}\ x\texttt{->}\{se_3^\circ[\overline{x}' := \overline{l}^{x'}]\})$ where $\{\overline{l}^{x'}\} \subseteq dom(\rho_1')$,
(P_2) $[\![e_b]\!]^{dom(\sigma')\cup\{x\},\{\overline{y}\}} = (\, se_3^\circ, z' \,)$ for some $\overline{y}$, z', and
(P_3) for all i, $1 \le i \le n'$, if $x_i'{:}T_i' \mapsto v_i' \in \sigma'$, then $v_i' \cong \langle \rho_1'(l_i^{x'}) \mid \rho_1' \rangle$.

From (B_2), and (P_3) we derive

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

Now consider the configurations $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle$, and $\langle se' \mid \rho_2'' \rangle$ where $\rho_2'' = \rho_2'[l_x \mapsto v_x', \overline{l}^{w'} \mapsto \overline{?}, l_{z'} \mapsto ?]$ and $se' = se_3^\circ[\overline{x}'\, x\, \overline{w}'\, z' := \overline{l}^{x'}\, l_x\, \overline{l}^{w'}\, l_{z'}]$ with $\{\overline{w}'\} = def(se_3^\circ) - \{z'\}$. We want to show that these configurations satisfy the conditions of Definition 6.
From Lemma 1 we have that $\models (\texttt{fun}\ x{:}T_1 \texttt{->} e_b, \sigma'){:}T_1 \to T$. From Definition 2(c), and rule (TyAbs) preceding Definition 2 we derive that $env(\sigma')[x{:}T_1] \mid \emptyset \vdash e_b : T$, and $\models \sigma'\diamond$. Applying Lemma 1 to $\langle e_2 \mid \sigma \mid \rho_1 \rangle \Downarrow \langle v_x \mid \rho_2 \rangle$ we have $\models v_x{:}T_1$, and $\Sigma_2 \models \rho_2$ for some $\Sigma_2 \supseteq \Sigma_1$. Therefore, we get $\models \sigma'[x{:}T_1 \mapsto v_x]\diamond$, and $\Sigma_2 \models \langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle\diamond$.
From (3), and Theorem 2, we derive that $\langle v'' \mid \rho_1' \rangle$ is well-formed. From (P_1), (P_2), and definition of se' we have that $FV(se') = \emptyset$, and $Loc(se') \subseteq dom(\rho_2'')$. From (4), and Theorem 2 we have that ρ_2' is location closed. Therefore, $\langle se' \mid \rho_2'' \rangle$ is well-formed.
From (P_2), (C_2), and definitions of se' and ρ_2'' we can derive that $\langle e_b \mid \sigma'[x{:}T_1 \mapsto v_x] \mid \rho_2 \rangle \approx \langle se' \mid \rho_2'' \rangle$ w.r.t. Σ_2, the sequences of disjoint variables $\overline{x}'\, x$, $\overline{w}'$, z', the sequences of disjoint locations $\overline{l}^{x'}$, $\overline{l}^{w'}$, $l_{z'}$, and the F# expression e_b.
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

$$\langle se' \mid \rho_2'' \rangle \Downarrow_{sq} \langle v' \mid \rho_*'' \rangle$$

where

(A_3) $v \cong \langle v' \mid \rho_*'' \rangle$ and $v' = \rho_*''(l_{z'})$,

(B_3) $l \notin \{\overline{l}^{y}, \overline{l}^{w'}, l_{z'}\}$ implies $\rho_2''(l) = \rho_*''(l)$,

(C_3) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho_*''(l) \mid \rho_*' \rangle$.

Applying rule (Block) of Fig. 8 we derive $\langle \{se_3^\circ[\overline{x}' := \overline{l}^{x'}]\} \mid \rho_2'[l_x \mapsto v_x']\rangle \Downarrow_{bl} \langle v' \mid \rho_*'' \rangle$. Consider the evaluation of the configuration $\langle \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho_2' \rangle$. From ($A_1$), and ($B_2$) we derive $\rho_2'(l_{z_1}) = \texttt{fun}\ x\texttt{->}\{se_3^\circ[\overline{x}' := \overline{l}^{x'}]\}$, and from ($A_2$) we get $\rho_2'(l_{z_2}) = v_x'$. Therefore, applying rule (App) of Fig. 8, we derive that $\langle (l_{z_1}\ l_{z_2}) \mid \rho_2' \rangle \Downarrow_{ex} \langle v' \mid \rho_*'' \rangle$. By applying (Def) we have

$$\langle \texttt{def}\ l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho_2' \rangle \Downarrow_{st} \langle v' \mid \rho_*''[l_z := v'] \rangle. \tag{5}$$

Let $\rho'_* = \rho''_*[l_z := v']$. From (3), (4), and (5), applying rule (SEQ) twice we get

$$\langle se_1; se_2; \texttt{def } l_z{:=}(l_{z_1}\ l_{z_2}) \mid \rho' \rangle \Downarrow_{st} \langle v' \mid \rho'_* \rangle.$$

From (A_3), and definition of ρ'_* we have

(A') $v \cong \langle v' \mid \rho'_* \rangle$ and $v' = \rho'_*(l_z)$.

From $\{\overline{l}^{w}\} \supseteq \{\overline{l}^{w_1}, \overline{l}^{w_2}, l_{z_1}, l_{z_2}, \}$, $\{\overline{l}^{w'}, l_{z'}\} \cap dom(\rho') = \emptyset$, (B_1), and (B_2) we derive that

(B') $l \notin \{\overline{l}^{y}, \overline{l}^{w}, l_z\}$ implies $\rho'(l) = \rho'_*(l)$.

Finally, from (C_3), and the fact that $l_z \notin dom(\rho_*)$, we get that

(C') $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_* \rangle$.

This concludes the proof.

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 $l_y \notin dom(\rho_1)$.
Since $e = e^{\circ}[\overline{y} := \overline{l}^{y}]$, then $e^{\circ}{=}\texttt{let mutable } y{:}T{=}e_1^{\circ} \texttt{ in } e_2^{\circ}$ where $e_i = e_i^{\circ}[\overline{y} := \overline{l}^{y}]$. From Definition 4, clause 8(a), $[\![e^{\circ}]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se^{\circ}, z_2\,)$ where $se^{\circ} = se_1^{\circ}; \texttt{def } y{:=}z_1; se_2^{\circ}$, and $[\![e_i^{\circ}]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se_i^{\circ}, z_i\,)$ $(1 \le i \le 2)$. From Definition 6, clause 3, we have that se is $se_1; \texttt{def } l_y{:=}l_{z_1}; se_2$ where $se_i = se_i^{\circ}[\overline{x}\,\overline{y}\,\overline{w}^{i}\ z_i := \overline{l}^{x}\,\overline{l}^{y}\,\overline{l}^{w_i}\ l_{z_i}]$, and $\{\overline{l}^{x}, \overline{l}^{y}\,\overline{l}^{w}, l_z\} \subseteq dom(\rho')$. Moreover, $def(se^{\circ}) = def(se_1^{\circ}) \cup def(se_2^{\circ})$, $def(se_1^{\circ}) \cap def(se_2^{\circ}) = \emptyset$, and $\{\overline{w}^{i}\} = def(se_i^{\circ}) - \{z_i\}$.
As for the proof of the case of application we can prove that

$$\langle e_1 \mid \sigma \mid \rho \rangle \approx \langle se_1 \mid \rho' \rangle$$

w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}^1$, z_1, the locations $\overline{l}^{x}$, $\overline{l}^{y}$, $\overline{l}^{w_1}$, l_{z_1}, and the F# expression e_1°. Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle v_y \mid \rho_1 \rangle$ we get

$$\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle v'_y \mid \rho'_1 \rangle. \tag{6}$$

Moreover, the following hold:

(A_1) $v_y \cong \langle v'_y \mid \rho'_1 \rangle$ and $v'_y = \rho'_1(l_{z_1})$

(B_1) $l \notin \{\overline{l}^y, \overline{l}^{w_1}, l_{z_1}\}$ implies $\rho'(l) = \rho'_1(l)$, and

(C_1) $l \in dom(\rho_1)$ implies $\rho_1(l) \cong \langle \rho'_1(l) \mid \rho'_1 \rangle$.

From rule (Def) of Fig. 8 we have

$$\langle \texttt{def}\ l_y{:=}l_{z_1} \mid \rho'_1 \rangle \Downarrow_{sq} \langle v'_y \mid \rho'_1[l_y := v'_y] \rangle. \tag{7}$$

Consider the configurations $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle$, and $\langle se_2 \mid \rho'_1[l_y := v'_y] \rangle$.
From $\Sigma \models \langle e_1 \mid \sigma \mid \rho \rangle \diamond$ and Lemma 1 we have that: $\Sigma' \models \rho_1$ (for some $\Sigma' \supseteq \Sigma$), $\models \sigma \diamond$, and $\models v_y{:}T'$ (since $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y \vdash e_1 : T'$).
Let $\Sigma_1 = \Sigma', l_y{:}T$, then $\Sigma_1 \models \rho_1[l_y \mapsto v_y]$, and $env(\sigma) \mid \overline{l}^y{:}\overline{T}^y, l_y{:}T \vdash e_2 : T$. Therefore $\Sigma_1 \models \langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle \diamond$.
From Theorem 2 we have that $\langle v'_y \mid \rho'_1 \rangle$ is well-formed, and, since $Loc(se_2) \subseteq dom(\rho') \subseteq dom(\rho'_1)$, we get that $\langle se_2 \mid \rho'_1[l_y := v'_y] \rangle$ is well-formed.
From $\langle e \mid \sigma \mid \rho \rangle \approx \langle se \mid \rho' \rangle$ we derive that clauses $2 \div 4$ of Definition 6 hold for the configurations $\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle$ and $\langle se_2 \mid \rho'_1[l_y := v'_y] \rangle$ with $\overline{x}$, $\overline{y}\, y$, $\overline{w}^2$, z_2, $\overline{l}^x$, $\overline{l}^y\, l_y$, $\overline{l}^{w_2}$, l_{z_2}, and e_2°. Therefore

$$\langle e_2[y := l_y] \mid \sigma \mid \rho_1[l_y \mapsto v_y] \rangle \approx \langle se_2 \mid \rho'_1[l_y := v'_y] \rangle.$$

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

$$\langle se_2 \mid \rho'_1[l_y := v'] \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle \tag{8}$$

where

(A_2) $v \cong \langle v' \mid \rho'_* \rangle$ and $v' = \rho'_*(l_{z_2})$,

(B_2) $l \notin \{\overline{l}^y, \overline{l}^{w_2}, l_{z_2}\}$ implies $(\rho'_1[l_y := v'_y])(l) = \rho'_*(l)$, and

(C_2) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_* \rangle$.

From $\{\overline{l}^w\} \supseteq \{\overline{l}^{w_1}, \overline{l}^{w_2}, l_{z_1}, l_{z_2}, \}$, ($B_1$), and ($B_2$) we derive that

(B') $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ implies $\rho'(l) = \rho'_*(l)$.

Therefore (A_2), (B'), and (C_2) prove the result.

Rule (LetRec-F) In this case $e{=}\texttt{let rec}\ \overline{w}'{=}\overline{F}\ \texttt{in}\ e_0$. From rule (LetRec-F), if $\langle e_0 \mid \sigma[w'_j{:}T_j{\mapsto}(\texttt{let rec}\ \overline{w}'{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j, \sigma)]_{1\le j\le m} \mid \rho \rangle \Downarrow \langle v \mid \rho_1 \rangle$, then $e \Downarrow \langle v \mid \rho_1 \rangle$. Assume that $\{\overline{w}'\} \cap \{\overline{x}, \overline{y}\} = \emptyset$. From $\Sigma \models$

$\langle e \mid \sigma \mid \rho\rangle\diamond$, and typing rules (TyRec) and (TyAbs) of Fig. 2 we have that for all j, $1 \leq j \leq m$, $FV(F_j) \cap \{\overline{y}\} = \emptyset$. From Definition 6, since $e = e^{\circ}[\overline{y} := \overline{l}^{y}]$, we derive e°=`let rec` $\overline{w}'$=$\overline{F}$ `in` e_0° where $e_0 = e_0^{\circ}[\overline{y} := \overline{l}^{y}]$. Let F_j = `fun` x:T_j->e_j $(1 \leq j \leq m)$. From Definition 4, clause 9(a), $[\![e^{\circ}]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se^{\circ}, z_0\,)$. So se° is $se_1^{\circ}; \cdots ; se_m^{\circ}$; `def` w_1':=z_1; $\cdots$; `def` w_m':=z_m; se_0°, where $\{\overline{x}\} = dom(\sigma)$,

- (P_1) $[\![F_j]\!]^{\{\overline{x},\overline{w}'\},\{\overline{y}\}}$=$(\, se_j^{\circ}, z_j\,)$, i.e., se_j° = `def` z_j:=`fun` x->$\{se_j'\}$ where $[\![e_j]\!]^{\{\overline{x},x,\overline{w}'\},\{\overline{y}\}}$=$(\, se_j', z_j\,)$ $(1 \leq j \leq m)$,
- (P_2) $[\![e_0^{\circ}]\!]^{\{\overline{x},\overline{w}'\},\{\overline{y}\}}$=$(\, se_0^{\circ}, z_0\,)$,
- (P_3) $se = se_1; \cdots ; se_m$; `def` $l_{w_1'}$:=l_{z_1}; $\cdots$; `def` $l_{w_m'}$:=l_{z_m}; se_0 where $se_k = se_k^{\circ}[\overline{x}\,\overline{y}\,\overline{w}\,z_k := \overline{l}^{x}\,\overline{l}^{y}\,\overline{l}^{w}\,l_{z_k}]$ $(0 \leq k \leq m)$, $\overline{w} = \overline{w}' z_1 \cdots z_m \overline{w}^0$, and $\overline{w}^0 = def(se_0) - \{z_0\}$,
- (P_4) $\rho' = \rho^I + \rho^M$, $dom(\rho^M) = dom(\rho)$, and $\{\overline{l}^{x}, \overline{l}^{w}, l_{z_0}\} \subseteq dom(\rho^I)$.

From the definition of the translation of functions (Definition 4, clause 2) we have that for all j, $1 \leq j \leq m$, the only defined variable in se_j° is z_j, and z_j cannot occur in the body of the function. Therefore, from (P_3) we have se_j = `def` l_{z_j}:=F_j' where F_j' = `fun` x->$\{se_j'[\overline{x}\,\overline{w}' := \overline{l}^{x}\,\overline{l}^{w'}]\}$ $(1 \leq j \leq m)$.
Applying rules (Seq), and (Def) of Fig. 8 we have that

$$\langle se_1; \cdots ; se_m; \texttt{def}\ l_{w_1'}:=l_{z_1}; \cdots ; \texttt{def}\ l_{w_m'}:=l_{z_m} \mid \rho'\rangle \Downarrow_{sq} \langle F_m' \mid \rho'_1\rangle$$

where $\rho'_1 = \rho'[l_{w_j'}, l_{z_j} \mapsto F_j']_{1\leq j\leq m}$.
Let $v_k^R = (\texttt{let rec}\ \overline{w}'{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_k, \sigma)$, and $\sigma' = \sigma[w_k'{:}T_k \mapsto v_k^R]_{1\leq k\leq m}$. We want to show that $\langle e_0 \mid \sigma' \mid \rho\rangle \approx \langle se_0 \mid \rho'_1\rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}\,\overline{w}'$ $(dom(\sigma'))$, $\overline{y}$, $\overline{w}^0$, z_0, the locations $\overline{l}^{x}\,\overline{l}^{w'}$, $\overline{l}^{y}$, $\overline{l}^{w_0}$, l_{z_0}, and the F# expression e_0°.
From $\Sigma \models \langle \texttt{let rec}\ \overline{w}'{=}\overline{F}\ \texttt{in}\ e_0 \mid \sigma \mid \rho\rangle\diamond$ we have that $\Sigma \models \rho$, and from rule (TyRec) of Fig. 2 we get

1. $\Gamma[\overline{w}'{:}\overline{T}] \mid \Sigma \vdash F_j : T_j$ $(1 \leq j \leq m)$, and
2. $\Gamma[\overline{w}'{:}\overline{T}] \mid \Sigma \vdash e_0 : T$.

From 1. and rule (TyRec) we have that $\Gamma \mid \Sigma \vdash \texttt{let rec}\ \overline{w}'{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j : T_j$ $(1 \leq j \leq m)$, and $\models \sigma'\diamond$. Therefore $\Sigma \models \langle e_0 \mid \sigma' \mid \rho\rangle\diamond$.
From $\langle se \mid \rho'\rangle$ well-formed and Theorem 2 we derive that $\langle F_m' \mid$

$\rho'[l_{w'_j}, l_{z_j} \mapsto F'_j]_{1\le j\le m}\rangle$ and $\langle se_0 \mid \rho'[l_{w'_j}, l_{z_j} \mapsto F'_j]_{1\le j\le m}\rangle$ are well-formed.
From $\Gamma[\overline{w}'{:}\overline{T}] \mid \Sigma \vdash e_0 : T$, (P_2), and (P_3), and $\{\overline{w}'\} \subseteq \{\overline{w}\}$, and $\{\overline{z}\} \cap \{\overline{w}_0\} = \emptyset$ we derive clauses 2 and 3 of Definition 6. From (P_4) and $\{\overline{w}_0\} \subseteq \{\overline{w}\}$ we have that $\rho' = \rho^I + \rho^M$, $dom(\rho^M) = dom(\rho)$, and $\{\overline{l}^x, \overline{l}^{w_0}, l_{z_0}\} \subseteq dom(\rho^I)$.
Let $x'{:}T' \mapsto v'' \in \sigma'$. If $x'{:}T' \mapsto v'' \in \sigma$, then, from (α) and $\rho'(l_{x'}) = \rho'_1(l_{x'})$ we have that $v'' \cong \langle \rho'_1(l_{x'}) \mid \rho'_1\rangle$. If instead $x'{:}T' \mapsto v'' \notin \sigma$, then for some j, $1 \le j \le m$, $x' = w'_j$, $v'' = (\texttt{let rec}\ \overline{w}'{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j, \sigma)$, and $\rho'_1(l_{x'}) = F'_j$. From $\rho'_1(l_{w'_k}) = F'_k$ $(1 \le k \le m)$, and (P_1), we have that $(\texttt{let rec}\ \overline{w}'{:}\overline{T}{=}\overline{F}\ \texttt{in}\ F_j, \sigma) \cong \langle F'_k \mid \rho'_1\rangle$. Therefore, we can conclude that $\langle e_0 \mid \sigma' \mid \rho\rangle \approx \langle se_0 \mid \rho'_1\rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}\,\overline{w}'$ $(dom(\sigma'))$, $\overline{y}$, $\overline{w}^0$, and z_0, the locations $\overline{l}^x\,\overline{l}^{w'}$, $\overline{l}^y$, $\overline{l}^{w_0}$, and l_{z_0}, and the F# expression e°_0.
Applying the inductive hypothesis to $\langle e_0 \mid \sigma' \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ we have that

$$(\star) \qquad \langle se_0 \mid \rho'_1\rangle \Downarrow_{sq} \langle v' \mid \rho'_*\rangle$$

where

- (A_1) $v \cong \langle v' \mid \rho'_*\rangle$ and $v' = \rho'_*(l_{z_0})$,
- (B_1) $l \notin \{\overline{l}^y, \overline{l}^{w_0}, l_{z_0}\}$ implies $\rho'(l) = \rho'_*(l)$, and
- (C_1) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$.

From (B_1), $\{\overline{w}_0\} \subseteq \{\overline{w}\}$, and $l \notin \{\overline{l}^z, \overline{l}^{w'}\}$ implies $\rho'(l) = \rho'_1(l)$, we derive that

- (B'_1) $l \notin \{\overline{l}^y, \overline{l}^w, l_{z_0}\}$ implies $\rho'(l) = \rho'_*(l)$.

Therefore (A_1), (B'_1), and (C_1) prove the result.

Rule (Ass-F) In this case $e{=}l{\leftarrow}e_1$ and $\langle l{\leftarrow}e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$, and $\rho_* = \rho_1[l \mapsto v]$. Since $e = e^\circ[\overline{y} := \overline{l}^y]$, then $e^\circ{=}y{\leftarrow}e^\circ_1$ where $y \in \{\overline{y}\}$, and $e_1 = e^\circ_i[\overline{y} := \overline{l}^y]$.
From Definition 4, clause 10, we have that $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ_1; x{\leftarrow}z, z\,)$ where $[\![e^\circ_1]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se^\circ_1, z\,)$. From Definition 6, clause 3, we derive that se is $se_1; l_y{\leftarrow}l_z$ where $se_1 = se^\circ_1[\overline{x}\,\overline{y}\,\overline{w}\,z := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,l_z]$, $\{\overline{w}\} = def(se^\circ_1) - \{l_z\}$, and $\{\overline{l}^x, \overline{l}^y\,\overline{l}^w, l_z\} \subseteq dom(\rho')$. Therefore $\langle e_1 \mid \sigma \mid \rho\rangle \approx \langle se_1 \mid \rho'\rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}$, z, the locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^w$, l_z, and the F# expression e°_1.

Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_1\rangle$ we get $\langle se_1 \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_1\rangle$ where

(A_1) $v \cong \langle v' \mid \rho'_1\rangle$ and $v' = \rho'_1(l_z)$,
(B_1) $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ implies $\rho'(l) = \rho'_1(l)$, and
(C_1) $l \in dom(\rho_1)$ implies $\rho_*(l) \cong \langle \rho'_1(l) \mid \rho'_1\rangle$.

From $\langle se_1 \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_1\rangle$, applying rule (SEQ) and (ASS), we get

$$\langle se_1; l_y \mathrel{<\!\!-} l_z \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$$

where $\rho'_* = \rho'_1[l_y \mapsto v']$.
From (A_1) and the definition of ρ'_* we get

(A'_1) $v \cong \langle v' \mid \rho'_*\rangle$, and $v' = \rho'_*(l_z)$.

Since ρ'_1 and ρ'_* differs only for the location $l_y \in \{\overline{l}^y\}$, from ($B_1$) we have that

(B'_1) $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ implies $\rho'(l) = \rho'_*(l)$.

Finally, from (C_1), $\rho_* = \rho_1[l \mapsto v]$ and (A'_1) we have that

(C'_1) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$.

Therefore (A'_1), (B'_1), and (C'_1) prove the result.

Rule (Throw-F) In this case e=`raise` e_1, and $\langle e \mid \sigma \mid \rho\rangle \Downarrow \langle \mathsf{E}(v) \mid \rho_*\rangle$ where $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$.
From Definition 4, clause 11, we get that $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se_1^\circ; \texttt{raise}\ z, z\,)$ where $[\![e_1^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\, se_1^\circ, z\,)$. From Definition 6, clause 3, we have that se is $se_1; \texttt{raise}\ l_z$ where $se_1 = se_1^\circ[\overline{x}\,\overline{y}\,\overline{w}\,z := \overline{l}^x\,\overline{l}^y\,\overline{l}^w\,l_z]$, $\{\overline{w}\} = def(se_1^\circ) - \{l_z\}$, and $\{\overline{l}^x, \overline{l}^y\,\overline{l}^w, l_z\} \subseteq dom(\rho')$. Therefore $\langle e_1 \mid \sigma \mid \rho\rangle \approx \langle se_1 \mid \rho'\rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}$, z, the locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^w$, l_z, and the F# expression e_1°.

Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho\rangle \Downarrow \langle v \mid \rho_*\rangle$ we get $\langle se_1 \mid \rho'\rangle \Downarrow_{sq}\langle v' \mid \rho'_*\rangle$ where

(A_1) $v \cong \langle v' \mid \rho'_*\rangle$ and $v' = \rho'_*(l_z)$,
(B_1) $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ implies $\rho'(l) = \rho'_*(l)$, and
(C_1) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho'_*(l) \mid \rho'_*\rangle$.

From $\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$, applying rule (Seq) and (Throw), we get

$$\langle se_1; \texttt{raise}\ l_z \mid \rho' \rangle \Downarrow_{sq} \langle v' \mid \rho'_* \rangle$$

Therefore (A_1), (B_1), and (C_1) prove the result.

Rule (CthExc-F) In this case $e = \texttt{try}\ e_1\ \texttt{with}\ x \texttt{ -> } e_2$, and $\langle \texttt{try}\ e_1\ \texttt{with}\ x \texttt{ -> } e_2 \mid \sigma \mid \rho \rangle \Downarrow \langle v \mid \rho_* \rangle$ where $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle \mathsf{E}(v_1) \mid \rho_1 \rangle$, and $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle \Downarrow \langle v \mid \rho_* \rangle$.
Since $e = e^\circ[\overline{y} := \overline{l}^y]$, then $e^\circ = \texttt{try}\ e_1^\circ\ \texttt{with}\ x \texttt{ -> } e_2^\circ$, where $e_i = e_i^\circ[\overline{y} := \overline{l}^y]$. From Definition 4, clause 12, we have that $[\![e^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se^\circ, z_2\ )$ where $se^\circ = (\ \texttt{def}\ z{:=}\_; \texttt{try}\ \{se_1; z{<-}z_1\}\ \texttt{catch}\,(\ x\ )\ \{se_2; z{<-}z_2\}, z\ )$, and $[\![e_i^\circ]\!]^{\{\overline{x}\},\{\overline{y}\}} = (\ se_i^\circ, z_i\ )$ $(1 \le i \le 2)$. Moreover, since z is fresh, we have that $z \notin FV(se_i^\circ)$ $(1 \le i \le 2)$.
From Definition 6, clause 3, we have that se is

$$\texttt{def}\ l_z{:=}\_; \texttt{try}\ \{se_1; l_z{<-}l_{z_1}\}\ \texttt{catch}\,(\ x\ )\ \{se_2; l_z{<-}l_{z_2}\}$$

where $se_i = se_i^\circ[\overline{x}\,\overline{y}\,\overline{w}^i\ z_i := \overline{l}^x\,\overline{l}^y\,\overline{l}^{w_i}\ l_{z_i}]$, and $\{\overline{l}^x, \overline{l}^y\,\overline{l}^w, l_z\} \subseteq dom(\rho')$. Moreover: $def(se^\circ) = def(se_1^\circ) \cup def(se_2^\circ)$, $def(se_1^\circ) \cap def(se_2^\circ) = \emptyset$, $\{\overline{w}^i\} = def(se_i^\circ) - \{z_i\}$, and l_z is not in se_i $(1 \le i \le 2)$.
As for the proof for application and let-mutable we can show that from $\langle e \mid \sigma \mid \rho \rangle \approx \langle se \mid \rho' \rangle$ we derive $\langle e_1 \mid \sigma \mid \rho \rangle \approx \langle se_1 \mid \rho' \rangle$ w.r.t. $\Sigma = \overline{l}{:}\overline{T}$, the variables $\overline{x}$, $\overline{y}$, $\overline{w}^1$, z_1, the locations $\overline{l}^x$, $\overline{l}^y$, $\overline{l}^{w_1}$, l_{z_1}, and the F# expression e_1°.
Applying the inductive hypothesis to $\langle e_1 \mid \sigma \mid \rho \rangle \Downarrow \langle \mathsf{E}(v_1) \mid \rho_1 \rangle$ we get $\langle se_1 \mid \rho' \rangle \Downarrow_{sq} \langle \mathsf{E}(v_1') \mid \rho_1' \rangle$, and the following hold:

- (A_1) $v_1 \cong \langle v_1' \mid \rho_1' \rangle$ and $v_1' = \rho_1'(l_{z_1})$,
- (B_1) $l \notin \{\overline{l}^y, \overline{l}^{w_1}, l_{z_1}\}$ implies $\rho'(l) = \rho_1'(l)$, and
- (C_1) $l \in dom(\rho_1)$ implies $\rho_1(l) \cong \langle \rho_1'(l) \mid \rho_1' \rangle$.

From rule (PrSeq) of Fig. 9 and propagation of exceptions for blocks we have that

$$\langle \{se_1; l_z{<-}l_{z_1}\} \mid \rho' \rangle \Downarrow_{bl} \langle \mathsf{E}(v_1') \mid \rho_1' \rangle. \tag{9}$$

Consider the configurations $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle$ and $\langle se_2[x := l_x] \mid \rho_1'[l_x := v_1'] \rangle$.
From $\Sigma \models \langle e_1 \mid \sigma \mid \rho \rangle \diamond$ and Lemma 2 we have that $\Sigma' \models \rho_1$ for some $\Sigma_1 \supseteq \Sigma$, $\models \sigma \diamond$, and $\models v_1{:}T_{\mathsf{E}}$. Therefore $\models \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \diamond$, and

$\Sigma_1 \models \langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle \diamond$.
From Theorem 2 we derive that configurations $\langle \mathsf{E}(v_1') \mid \rho_1' \rangle$ and $\langle v_1' \mid \rho_1' \rangle$ are well-formed. From $Loc(se_2) \subseteq dom(\rho') \subseteq dom(\rho_1')$ we get that $\langle se_2[x := l_x] \mid \rho_1'[l_x := v_1'] \rangle$ is well-formed.
From $\langle e \mid \sigma \mid \rho \rangle \approx \langle se \mid \rho' \rangle$ we derive that clauses $2 \div 4$ of Definition 6 hold for the configurations $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle$ and $\langle se_2[x := l_x] \mid \rho_1'[l_x := v_1'] \rangle$ with $\overline{x}$, $\overline{y}\, y$, $\overline{w}^2$, z_2, $\overline{l}^x$, $\overline{l}^y\, l_y$, $\overline{l}^{w_2}$, l_{z_2}, and e_2°. Therefore

$$\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle \approx \langle se_2[x := l_x] \mid \rho_1'[l_x := v_1'] \rangle.$$

Applying the inductive hypothesis to $\langle e_2 \mid \sigma[x{:}T_{\mathsf{E}} \mapsto v_1] \mid \rho_1 \rangle \Downarrow \langle v \mid \rho_* \rangle$ we get $\langle se_2[x := l_x] \mid \rho_1'[l_x := v_1'] \rangle \Downarrow_{sq} \langle v' \mid \rho_2' \rangle$ where

- (A_2) $v \cong \langle v' \mid \rho_2' \rangle$ and $v' = \rho_2'(l_{z_2})$,
- (B_2) $l \notin \{\overline{l}^y, \overline{l}^{w_2}, l_{z_2}\}$ implies $\rho_1'(l) = \rho_2'(l)$, and
- (C_2) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho_2'(l) \mid \rho_2' \rangle$.

Let $\rho_*' = \rho_2'[l_z \mapsto \rho_1'(l_{z_2})]$. From rules (Seq) and (Block) of Fig. 8 we have that

$$\langle \{se_2[x := l_x]; l_z{<}{-}l_{z_1}\} \mid \rho_1' \rangle \Downarrow_{bl} \langle v' \mid \rho_*' \rangle. \tag{10}$$

From (9), (10), and rule (CthExc) of Fig. 9 we derive that

$$\langle \texttt{try}\ \{se_1; l_z{<}{-}l_{z_1}\}\ \texttt{catch}\,(\,x\,)\ \{se_2[x := l_x]; l_z{<}{-}l_{z_1}\} \mid \rho' \rangle \Downarrow_{st} \langle v' \mid \rho_*' \rangle.$$

From rules (Def) and (Seq) of Fig. 8 we have that $\langle se \mid \rho' \rangle \Downarrow_{st} \langle v' \mid \rho_*' \rangle$. From (A_2), (B_1), (B_2), (C_2), the definition of ρ_*', and $l_z \notin dom(\rho_*)$ we have that

- (A_3) $v \cong \langle v' \mid \rho_*' \rangle$ and $v' = \rho_*'(l_z)$,
- (B_3) $l \notin \{\overline{l}^y, \overline{l}^w, l_z\}$ implies $\rho'(l) = \rho_*'(l)$, and
- (C_3) $l \in dom(\rho_*)$ implies $\rho_*(l) \cong \langle \rho_*'(l) \mid \rho_*' \rangle$.

This concludes the proof. □