Specification of the Stand-in Language

Wolfgang Jeltsch

Well-Typed LLP

wolfgang@well-typed.com

November 15, 2018

1 Introduction

We present a formal specification of the Stand-in Language (SIL) by Sam Griffin. Our specification closely resembles the version of the language implemented in Commit `966227a` of the GitHub repository `sfultong/stand-in-language`.

SIL actually consists of two languages: the surface language and the internal language. While the former constitutes the user-facing part, the latter is the one for which an operational semantics and a type system are defined.

2 Notation

Throughout this document, we describe syntax using a variant of Backus–Naur form. Our variant uses the following notations, listed here in increasing order of precedence:

- | denotes set union.
- \ denotes set difference.
- Mere juxtaposition denotes concatenation.
- $^?$, $^+$, and * denote optionality, repetition with at least one occurrence, and arbitrary repetition, respectively.
- ⟨ and ⟩ delimit subexpressions and are used for overriding default precedence.[1]
- Words in italics are nonterminals.

We introduce further custom notations for various things unrelated to syntax. Since such notations are tied to specific parts of the specification, we define them where they are needed.

[1] We do not use (and) for this purpose, since they are part of some of the languages we want to describe.

$$
\begin{aligned}
\mathit{Token}_\mathrm{i} &::= \mathit{Keyword}_\mathrm{i} \mid \mathit{Symbol}_\mathrm{i} \\
\mathit{Keyword}_\mathrm{i} &::= \textbf{left} \mid \textbf{right} \mid \textbf{withenv} \mid \textbf{env} \mid \textbf{gate} \mid \textbf{defer} \mid \textbf{abort} \mid \textbf{trace} \\
\mathit{Symbol}_\mathrm{i} &::= \{ \mid , \mid \} \mid \emptyset \mid (\mid)
\end{aligned}
$$

Figure 1: Token syntax of the internal language

$$
\begin{aligned}
\mathit{Prog}_\mathrm{i} &::= \mathit{Expr}_\mathrm{i} \\
\mathit{Expr}_\mathrm{i} &::= \mathit{Pair}_\mathrm{i} \mid \mathit{Left}_\mathrm{i} \mid \mathit{Right}_\mathrm{i} \mid \mathit{Zero}_\mathrm{i} \mid \mathit{WithEnv}_\mathrm{i} \mid \mathit{Env}_\mathrm{i} \mid \mathit{Gate}_\mathrm{i} \mid \mathit{Defer}_\mathrm{i} \mid \mathit{Abort}_\mathrm{i} \mid \mathit{Trace}_\mathrm{i} \mid (\mathit{Expr}_\mathrm{i}) \\
\mathit{Pair}_\mathrm{i} &::= \{\mathit{Expr}_\mathrm{i}, \mathit{Expr}_\mathrm{i}\} \\
\mathit{Left}_\mathrm{i} &::= \textbf{left}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Right}_\mathrm{i} &::= \textbf{right}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Zero}_\mathrm{i} &::= \emptyset \\
\mathit{WithEnv}_\mathrm{i} &::= \textbf{withenv}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Env}_\mathrm{i} &::= \textbf{env} \\
\mathit{Gate}_\mathrm{i} &::= \textbf{gate}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Defer}_\mathrm{i} &::= \textbf{defer}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Abort}_\mathrm{i} &::= \textbf{abort}\ \mathit{Expr}_\mathrm{i} \\
\mathit{Trace}_\mathrm{i} &::= \textbf{trace}\ \mathit{Expr}_\mathrm{i}
\end{aligned}
$$

Figure 2: Syntax of the internal language

3 The Internal Language

The internal language is a low-level language, which in particular has no built-in support for closures. It comes with an operational semantics and a type system.

3.1 Syntax

Figure 1 defines the syntax of tokens. Based on that, Figure 2 defines the syntax of the language.

3.2 Semantics

Common functional programming languages express dependencies on external data and function arguments by means of variables. The internal language does not have variables technically, but the expression **env** can be considered a single variable. Following this view, an expression **defer** *e* corresponds to an abstraction ("λ-expression") $\lambda\textbf{env}\,.\,e$. Because such an abstraction binds the only variable that exists, the body of a **defer** expression cannot refer to variable assignments outside the **defer** expression. This means that the internal language does not support closures natively.

It is customary in the functional programming world to evaluate function expressions to

$$\frac{\mathscr{E} \vdash e_1 \to e_1' \quad \mathscr{E} \vdash e_2 \to e_2'}{\mathscr{E} \vdash \{e_1, e_2\} \to \{e_1', e_2'\}}$$

$$\frac{\mathscr{E} \vdash e \to \{e_1, e_2\}}{\mathscr{E} \vdash \textbf{left}\ e \to e_1} \qquad \frac{\mathscr{E} \vdash e \to \{e_1, e_2\}}{\mathscr{E} \vdash \textbf{right}\ e \to e_2}$$

$$\frac{\mathscr{E} \vdash e \to e' \quad \forall e_1\ e_2\ .\ e' \neq \{e_1, e_2\}}{\mathscr{E} \vdash \textbf{left}\ e \to \emptyset} \qquad \frac{\mathscr{E} \vdash e \to e' \quad \forall e_1\ e_2\ .\ e' \neq \{e_1, e_2\}}{\mathscr{E} \vdash \textbf{right}\ e \to \emptyset}$$

$$\frac{}{\mathscr{E} \vdash \emptyset \to \emptyset}$$

$$\frac{\mathscr{E} \vdash e \to \{e_1, e_2\} \quad e_2 \vdash e_1 \to e'}{\mathscr{E} \vdash \textbf{withenv}\ e \to e'}$$

$$\frac{}{\mathscr{E} \vdash \textbf{env} \to \mathscr{E}}$$

$$\frac{\mathscr{E} \vdash e \to \emptyset}{\mathscr{E} \vdash \textbf{gate}\ e \to \textbf{left env}} \qquad \frac{\mathscr{E} \vdash e \to e' \quad e' \neq \emptyset}{\mathscr{E} \vdash \textbf{gate}\ e \to \textbf{right env}}$$

$$\frac{}{\mathscr{E} \vdash \textbf{defer}\ e \to e}$$

$$\frac{\mathscr{E} \vdash e \to \emptyset}{\mathscr{E} \vdash \textbf{abort}\ e \to \emptyset}$$

$$\frac{\mathscr{E} \vdash e \to e'}{\mathscr{E} \vdash \textbf{trace}\ e \to e'}$$

Figure 3: Operational semantics

abstractions. If we would do the same for the internal language, evaluation of any function expression would result in an abstraction that binds **env**, because there is no other variable. Since mentioning the bound variable **env** would not provide any information, we yield only the bodies of the abstractions as evaluation results.

Expression evaluation happens in an environment. The environment is an evaluated expression that characterizes the value of **env**. Figure 3 defines a relation $- \vdash - \to - \subseteq \mathit{Expr} \times \mathit{Expr} \times \mathit{Expr}$ that constitutes the big-step (evaluation) semantics of the internal language. A proposition $\mathscr{E} \vdash e \to e'$ is true if and only if evaluating the expression e in the environment $\mathscr{E}$ may yield the expression e'.

The relation $- \vdash - \to -$ is in fact a partial function. This means that evaluation is deterministic: it either fails or leads to a uniquely defined expression.

The semantics only covers the pure aspects of execution. The **trace** construct is actually impure, as it outputs diagnostic information. Our semantics ignores that, treating any expression of the form **trace** e like e.

$$
\begin{aligned}
\mathit{Type} &::= \mathit{Type}_0 \\
\mathit{Type}_0 &::= \mathit{FunType} \mid \mathit{Type}_1 \\
\mathit{FunType} &::= \mathit{Type}_1 \Rightarrow \mathit{Type}_0 \\
\mathit{Type}_1 &::= \mathit{PairType} \mid \mathit{ZeroType} \mid \mathit{TypeVar} \mid (\mathit{Type}_0) \\
\mathit{PairType} &::= \{\mathit{Type}_0, \mathit{Type}_0\} \\
\mathit{ZeroType} &::= \emptyset \\
\mathit{TypeVar} &::= \mathit{Letter}
\end{aligned}
$$

Figure 4: Type syntax

$$
\frac{}{\emptyset \approx \{\emptyset, \emptyset\}}
$$

$$
\frac{\tau_1 \approx \tau_1' \quad \tau_2 \approx \tau_2'}{\tau_1 \Rightarrow \tau_2 \approx \tau_1' \Rightarrow \tau_2'} \qquad \frac{\tau_1 \approx \tau_1' \quad \tau_2 \approx \tau_2'}{\{\tau_1, \tau_2\} \approx \{\tau_1', \tau_2'\}} \qquad \frac{}{\emptyset \approx \emptyset} \qquad \frac{\alpha \in \mathit{TypeVar}}{\alpha \approx \alpha}
$$

$$
\frac{\tau_1 \approx \tau_2}{\tau_2 \approx \tau_1} \qquad \frac{\tau_1 \approx \tau_2 \quad \tau_2 \approx \tau_3}{\tau_1 \approx \tau_3}
$$

Figure 5: Equivalence of types

3.3 Type System

The type system of the internal language is special in two ways:

- There are two typing relations, one for unevaluated and one for evaluated expressions. This is necessary, because function expressions are evaluated to expressions that do not describe the functions themselves but their results. If we would use the same typing relation for unevaluated and evaluated expressions, the type of an expression could change through evaluation.
- The type system does not distinguish between different types of nested pairs. Each nested pair type contains all nested pairs.

Figure 4 defines the syntax of types, and Figure 5 defines an equivalence relation on types that identifies all nested pair types. Based on that, Figure 6 defines the typing relation $- \vdash - : - \subseteq \mathit{Type} \times \mathit{Expr} \times \mathit{Type}$ for unevaluated expressions. A proposition $\mathscr{T} \vdash e : \tau$ is true if and only if the unevaluated expression e has type τ provided that the environment has type $\mathscr{T}$. Finally, Figure 7 defines the typing relation $- : - \subseteq \mathit{Expr} \times \mathit{Type}$ for evaluated expressions.

Conjecture 1 (Type preservation)**.** *If $\mathscr{E} : \mathscr{T}$, $\mathscr{T} \vdash e : \tau$, and $\mathscr{E} \vdash e \to e'$, then $e' : \tau$.*

$$\frac{\mathscr{T} \vdash e_1 : \tau_1 \quad \mathscr{T} \vdash e_2 : \tau_2}{\mathscr{T} \vdash \{e_1, e_2\} : \{\tau_1, \tau_2\}}$$

$$\frac{\mathscr{T} \vdash e : \{\tau_1, \tau_2\}}{\mathscr{T} \vdash \textbf{left}\ e : \tau_1} \qquad \frac{\mathscr{T} \vdash e : \{\tau_1, \tau_2\}}{\mathscr{T} \vdash \textbf{right}\ e : \tau_2}$$

$$\frac{}{\mathscr{T} \vdash \emptyset : \emptyset}$$

$$\frac{\mathscr{T} \vdash e : \{\tau_1 \Rightarrow \tau_2, \tau_1\}}{\mathscr{T} \vdash \textbf{withenv}\ e : \tau_2}$$

$$\frac{}{\mathscr{T} \vdash \textbf{env} : \mathscr{T}}$$

$$\frac{\mathscr{T} \vdash e : \emptyset}{\mathscr{T} \vdash \textbf{gate}\ e : \{\tau, \tau\} \Rightarrow \tau}$$

$$\frac{\tau_1 \vdash e : \tau_2}{\mathscr{T} \vdash \textbf{defer}\ e : \tau_1 \Rightarrow \tau_2}$$

$$\frac{\mathscr{T} \vdash e : \emptyset}{\mathscr{T} \vdash \textbf{abort}\ e : \emptyset}$$

$$\frac{\mathscr{T} \vdash e : \tau}{\mathscr{T} \vdash \textbf{trace}\ e : \tau}$$

$$\frac{\tau_1 \approx \tau_2 \quad \mathscr{T} \vdash e : \tau_1}{\mathscr{T} \vdash e : \tau_2}$$

Figure 6: Typing rules for unevaluated expressions

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\{e_1, e_2\} : \{\tau_1, \tau_2\}}$$

$$\frac{}{\emptyset : \emptyset}$$

$$\frac{\mathscr{T} \vdash e : \tau}{e : \mathscr{T} \Rightarrow \tau}$$

$$\frac{\tau_1 \approx \tau_2 \quad e : \tau_1}{e : \tau_2}$$

Figure 7: Typing rules for evaluated expressions

$$
\begin{aligned}
\mathit{Token} &::= \mathit{Ident} \mid \mathit{Str} \mid \mathit{Nat} \mid \mathit{Keyword} \mid \mathit{Symbol} \\
\mathit{Ident} &::= \mathit{Letter}\langle \mathit{Letter} \mid \mathit{Digit} \mid \_ \mid \prime \rangle^* \setminus \mathit{Keyword} \\
\mathit{Str} &::= \text{“}\mathit{Char}^*\text{”} \\
\mathit{Nat} &::= \mathit{Digit}^+ \\
\mathit{Keyword} &::= \textbf{let} \mid \textbf{in} \mid \textbf{if} \mid \textbf{then} \mid \textbf{else} \mid \textbf{left} \mid \textbf{right} \mid \textbf{trace} \\
\mathit{Symbol} &::= : \mid = \mid \backslash \mid \rightarrow \mid \# \mid [\mid] \mid , \mid \$ \mid \{ \mid \} \mid (\mid)
\end{aligned}
$$

Figure 8: Token syntax of the surface language

4 The Surface Language

The surface language is a high-level language, which in particular has support for closures. Programs in the surface language are translated to the internal language using a process called desugaring. The surface language does not have an operational semantics of its own; a surface language expression is evaluated by first desugaring it and then evaluating the resulting internal expression. Currently, the surface language does not have a type system.

4.1 Syntax

Figure 8 defines the syntax of tokens. Based on that, Figure 9 defines the syntax of the language. The nonterminals *CLam*, *TNat*, and *FNat* stand for “complete lambda”, “tuple natural”, and “function natural”, respectively.

4.2 Desugaring

For defining desugaring, we introduce the following notations:

- $\langle e \rangle_{x \mapsto e'}$ means the expression e with free occurrences of x replaced by e'.
- $c_{\#}$ means the natural number literal that represents the code of the character c.
- n_{+} means the natural number literal that represents the successor of n.
- $f^n x$ means the n-fold application of f to x, that is, $f \ \ldots \ f \ x$ where f occurs n times.

Desugaring of an expression happens in a context, which is the list of variables that are in scope, with variables bound closer to the expression being listed later. We write ε for the empty context and $\Delta \rhd x$ for the context Δ extended by x. An internal expression obtained by desugaring in a context Δ is supposed to be evaluated in the environment $\{e_n, \{e_{n-1}, \{\ldots, \{e_1, \emptyset\} \ldots\}\}\}$ where each e_i is the value of the i-th variable in Δ.

We introduce a helper function $\rho : \mathit{Var}^+ \times \mathit{Var} \to \mathit{Expr}_{\mathrm{i}}$ such that $\rho(\Delta, x)$ yields an internal expression that is evaluated to a pair whose first element is the value of x. The definition of ρ is as follows:

$$
\rho(\Delta \rhd x, y) = \begin{cases} \textbf{env} & \text{if } x = y \\ \textbf{right}\ \rho(\Delta, y) & \text{if } x \neq y \end{cases} \tag{1}
$$

$$
\begin{aligned}
\mathit{Prog} &::= \mathit{Assign}^{+} \\
\mathit{Assign} &::= \mathit{Ident}\ \langle : \mathit{Expr}_0 \rangle^{?} = \mathit{Expr}_0 \\
\mathit{Expr}_0 &::= \mathit{Let} \mid \mathit{If} \mid \mathit{Lam} \mid \mathit{CLam} \mid \mathit{Expr}_1 \\
\mathit{Let} &::= \textbf{let}\ \mathit{Assign}^{*}\ \textbf{in}\ \mathit{Expr}_0 \\
\mathit{If} &::= \textbf{if}\ \mathit{Expr}_0\ \textbf{then}\ \mathit{Expr}_0\ \textbf{else}\ \mathit{Expr}_0 \\
\mathit{Lam} &::= \backslash\ \mathit{Var}^{+} \rightarrow \mathit{Expr}_0 \\
\mathit{CLam} &::= \#\ \mathit{Var}^{+} \rightarrow \mathit{Expr}_0 \\
\mathit{Expr}_1 &::= \mathit{App} \mid \mathit{Expr}_2 \\
\mathit{App} &::= \mathit{Expr}_1\ \mathit{Expr}_2 \\
\mathit{Expr}_2 &::= \mathit{List} \mid \mathit{Str} \mid \mathit{TNat} \mid \mathit{FNat} \mid \mathit{Pair} \mid \mathit{Left} \mid \mathit{Right} \mid \mathit{Trace} \mid \mathit{Var} \mid (\mathit{Expr}_0) \\
\mathit{List} &::= [\,] \mid [\mathit{Expr}_0 \langle , \mathit{Expr}_0 \rangle^{*}] \\
\mathit{TNat} &::= \mathit{Nat} \\
\mathit{FNat} &::= \$\mathit{Nat} \\
\mathit{Pair} &::= \{\mathit{Expr}_0, \mathit{Expr}_0\} \\
\mathit{Left} &::= \textbf{left}\ \mathit{Expr}_2 \\
\mathit{Right} &::= \textbf{right}\ \mathit{Expr}_2 \\
\mathit{Trace} &::= \textbf{trace}\ \mathit{Expr}_2 \\
\mathit{Var} &::= \mathit{Ident}
\end{aligned}
$$

Figure 9: Syntax of the surface language

Figure 10 defines a function $\lfloor - \rfloor_{\_} : \mathit{Expr} \times \mathit{Var}^* \to \mathit{Expr}_{\mathrm{i}}$ that describes desugaring of expressions. A term $\lfloor e \rfloor_{\Delta}$ gives the internal expression that corresponds to the surface expression e in the context Δ. Based on $\lfloor - \rfloor_{\_}$ we define a function $\lfloor - \rfloor : \mathit{Prog} \to \mathit{Prog}_{\mathrm{i}}$ that describes desugaring of programs:

$$\lfloor p \rfloor = \lfloor \mathbf{let}\ p\ \mathbf{in}\ \mathit{main} \rfloor_{\varepsilon} \tag{2}$$

$$
\begin{aligned}
\lfloor \textbf{let in } e \rfloor_\Delta &= \lfloor e \rfloor_\Delta \\
\lfloor \textbf{let } x = e' : e'' \ \bar{a} \textbf{ in } e \rfloor_\Delta &= \lfloor \textbf{let } x = e' \ \bar{a} \textbf{ in } e \rfloor_\Delta \\
\lfloor \textbf{let } x = e' \ \bar{a} \textbf{ in } e \rfloor_\Delta &= \lfloor \langle \textbf{let } \bar{a} \textbf{ in } e \rangle_{x \mapsto e'} \rfloor_\Delta \\
\lfloor \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \rfloor_\Delta &= \textbf{withenv } \{\textbf{gate } \lfloor e \rfloor_\Delta, \{\lfloor e_1 \rfloor_\Delta, \lfloor e_2 \rfloor_\Delta\}\} \\
\lfloor \backslash\ \bar{x} \to e \rfloor_\Delta &= \{\textbf{defer } \lfloor e \rfloor_{\Delta \circ \bar{x}}, \textbf{env}\} \\
\lfloor \#\ \bar{x} \to e \rfloor_\Delta &= \{\textbf{defer } \lfloor e \rfloor_{\bar{x}}, \emptyset\} \\
\lfloor e_1\ e_2 \rfloor_\Delta &= \textbf{withenv} \\
&\qquad \textbf{withenv } \{ \\
&\qquad\quad \textbf{defer } \{\textbf{left right env}, \{\textbf{left env}, \textbf{right right env}\}\}, \\
&\qquad\quad \{\lfloor e_2 \rfloor_\Delta, \lfloor e_1 \rfloor_\Delta\} \\
&\qquad \} \\
\lfloor [] \rfloor_\Delta &= \emptyset \\
\lfloor [e, l] \rfloor_\Delta &= \{\lfloor e \rfloor_\Delta, \lfloor [l] \rfloor_\Delta\} \\
\lfloor \text{“”} \rfloor_\Delta &= \emptyset \\
\lfloor \text{“}c\bar{c}\text{”} \rfloor_\Delta &= \{\lfloor c_\# \rfloor_\Delta, \lfloor \text{“}\bar{c}\text{”} \rfloor_\Delta\} \\
\lfloor 0 \rfloor_\Delta &= \emptyset \\
\lfloor n_+ \rfloor_\Delta &= \{\lfloor n \rfloor_\Delta, \emptyset\} \\
\lfloor \$n \rfloor_\Delta &= \lfloor \#\ f\ x \to f^n x \rfloor_\Delta \\
\lfloor \{e_1, e_2\} \rfloor_\Delta &= \{\lfloor e_1 \rfloor_\Delta, \lfloor e_2 \rfloor_\Delta\} \\
\lfloor \textbf{left } e \rfloor_\Delta &= \textbf{left } \lfloor e \rfloor_\Delta \\
\lfloor \textbf{right } e \rfloor_\Delta &= \textbf{right } \lfloor e \rfloor_\Delta \\
\lfloor \textbf{trace } e \rfloor_\Delta &= \textbf{trace } \lfloor e \rfloor_\Delta \\
\lfloor x \rfloor_\Delta &= \textbf{left } \rho(\Delta, x) \\
\lfloor (e) \rfloor_\Delta &= \lfloor e \rfloor_\Delta
\end{aligned}
$$

Figure 10: Desugaring of expressions