Jetsam: A Sound Binding Generator for TypeScript to Pyret

Alexander Vo

Contents

1 Introduction

This paper presents a specification for the binding generator *Jetsam* [2]: a program that reads TypeScript code and outputs binding code in Pyret.

Bindings are a kind of foreign function interface (FFI) mechanism. It is code inserted between a program written in one language and another that allows code to be utilized between the two languages in a defined matter.

FFIs help broaden the libraries available to a particular language. This is especially important for Pyret because its ecosystem is unfortunately tiny in comparison to TypeScript (which in turn has access to the JavaScript ecosystem). TypeScript's impressive collections of libraries include: *three.js* [7] (a 3D rendering library), *matter.js* [5] (a 2D physics engine), and *chart.js* [1] (charting library).

Pyret already has the facilities to interface with TypeScript code through manually written bindings. However, this manual labor includes writing type information in a Pyret-compatible format, inserting conversion routines, and setting up runtime scaffolding functions. Needless to say, this can be a tedious and erroneous process. Verifying the correctness of the FFI boundary must be done manually as well, and any API changes in the TypeScript code must be continuously handled and reviewed.

Jetsam seeks to automate the creation of the FFI boundary as well as the verification of correctness. In order to do this however, Jetsam needs to be aware of the semantics of both Pyret and TypeScript. In the spirit of the Lambda Cube, this paper presents various points along a "TypeScript Polyhedron" and "Pyret Polyhedron", subsets of their respective languages with increasingly complex features. Then, taking direct inspiration from Patterson and Ahmed [6], this paper uses source-type indexed logical relations to formally define how data "appears" at runtime. From there, this paper can define how they can be used by the other language through a convertibility relation.

This paper proves that a small variety of binding styles written in a point along the "Pyret Polyhedron" is semantically sound when wrapping code of a particular point along the "TypeScript Polyhedron". In turn, Jetsam will only accept TypeScript programs of those verified subsets and emit only one of the verified bindings styles. As a result, any code that Jetsam generates can be safely used from within Pyret.

2 Core Subset of TypeScript (TS_{NUM})

A core subset of TypeScript (TS_{NUM}) represented as a simply typed lambda calculus augmented with:

1. Double-precision 64-bit binary floats (format IEEE 754)
2. Float addition (corresponds directly to JS '+' operator)

2.1 Terms

$t ::=$

x	Variable
$float$	TS Number
$t_1 + t_2$	Addition
$\lambda x : T.t_1$	Abstraction
$t_1 t_2$	Application

2.2 Values

v ::=

$float$	Double-precision 64-bit binary format IEEE 754
$\lambda x : T.t$	Abstraction value

2.3 Types

T ::=

$Number$	Type of floats
$T \to T$	Type of functions

2.4 Evaluation

$$\text{(E-ADD1)}\ \frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2}$$

$$\text{(E-ADD1)}\ \frac{t_2 \to t_2'}{v_1 + t_2 \to v_1 + t_2'}$$

$$\text{(E-ADD3)}\ \frac{float_1 + float_2 = float_3}{float_1 + float_2 \to float_3}$$

$$\text{(E-APP1)}\ \frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$$

$$\text{(E-APP2)}\ \frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'}$$

$$\text{(E-APPABS)}\ \frac{}{(\lambda x : T.t_1) v_2 \to [x \mapsto v_2] t_1}$$

2.5 Typing

$$\text{(T-NUMBER)}\ \frac{}{\Gamma \vdash float : Number}$$

$$\text{(T-ADD)}\ \frac{\Gamma \vdash t_1 : Number \qquad \Gamma \vdash t_2 : Number}{\Gamma \vdash t_1 + t_2 : Number}$$

$$\text{(T-VAR)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{(T-ABS)}\ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x\colon T_1.t : T_1 \to T_2}$$

$$\text{(T-APP)}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

3 Core Subset of Pyret (PY_{NUM})

A core subset of Pyret (PY_{NUM}) represented as a simply typed lambda calculus augmented with:

1. "Unlimited" precision rational numbers (as defined in js-number.js [3])
2. Rational addition (corresponds to the add() function found in js-number.js [4])

3.1 Terms

t ::=

x	Variable
rational	Pyret Number
$t_1 + t_2$	Addition
$\lambda x\colon T.t_1$	Abstraction
$t_1 t_2$	Application

3.2 Values

v ::=

rational	"Unlimited" precision rational
$\lambda x\colon T.t$	Abstraction value

3.3 Types

T ::=

$Number$	Type of Pyret rationals
$T \to T$	Type of functions

Evaluation and typing rules are nearly identical to TS_{NUM} (save for a difference in the numeric type) and are omitted for brevity.

4 Target Language L_{NUM}

A common target language for TS_{NUM} and PY_{NUM} represented as a simply typed lambda calculus augmented with:

1. "Unlimited" precision rational numbers (as found in PY_{NUM})
2. Rational addition (as found in PY_{NUM})
3. Double-precision 64-bit binary floats (as found in TS_{NUM})
4. Float addition (as found in TS_{NUM})

4.1 Terms

t ::=

$float$	TS Number
$rational$	Pyret Number
$t_1 + t_2$	TS Addition
$t_1 +^{PY} t_2$	Pyret Addition
$t_1 t_2$	Application
$\lambda x\colon T.t$	Abstraction
FLAT t_1	Rational Evaluation (lossy)
RATt_1	Rational Creation

4.2 Values

v ::=

$float$	TS Number
$rational$	Pyret Number
$\lambda x.\tau$	Abstraction
$fail$	Explicit failure value

Note that the 'fail' value is NOT a 'NaN' found with floating point numbers.

4.3 Types

T ::=

num_f	Type of floats
num_r	Type of rationals
$T \to T$	Function type

4.4 Evaluation

$$\text{(E-ADD1)}\ \frac{t_1 \to t_1' \qquad t_1' \neq fail}{t_1 + t_2 \to t_1' + t_2}$$

$$\text{(E-ADD2)}\ \frac{t_2 \to t_2' \qquad t_2' \neq fail}{v_1 + t_2 \to v_1 + t_2'}$$

$$\text{(E-ADD3)}\ \frac{float_1 + float_2 = float_3}{float_1 + float_2 \to float_3}$$

$$\text{(E-PYADD1)}\ \frac{t_1 \to t_1' \qquad t_1' \neq fail}{t_1 +^{PY} t_2 \to t_1' +^{PY} t_2}$$

$$\text{(E-PYADD2)}\ \frac{t_2 \to t_2' \qquad t_2' \neq fail}{v_1 +^{PY} t_2 \to v_1 +^{PY} t_2'}$$

$$\text{(E-PYADD3)}\ \frac{rational_1 + rational_2 = rational_3}{rational_1 +^{PY} rational_2 \to rational_3}$$

$$\text{(E-APP1)}\ \frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$$

$$\text{(E-APP2)}\ \frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'}$$

$$\text{(E-APPABS)}\ \frac{}{(\lambda x\colon T.t_1)v_2 \to [x \mapsto v_2]t_1}$$

$$\text{(E-FLAT1)}\ \frac{t_1 \to t_1' \qquad t_1' \neq fail}{\text{FLAT } t_1 \to \text{FLAT } t_1'}$$

$$\text{(E-FLAT2)}\ \frac{rational \text{ as float} = f}{\text{FLAT } rational \to f}$$

$$\text{(E-RAT1)}\ \frac{t_1 \to t_1' \qquad t_1' \neq fail}{\text{RAT } t_1 \to \text{RAT } t_1'}$$

$$\text{(E-RAT2)}\ \frac{float \text{ as rational} = r}{\text{RAT } float \to r}$$

The behavior of terms with 'fail' values have been omitted for brevity. Assume evaluating any sub-term to 'fail' returns a 'fail'. Under the current formulation, no term may evaluate to the explicit 'fail' value. The value itself will be used to signal an FFI failure.

Also note that the implementation of '$+^{PY}$' (and all other rational operators provided by js-numbers.js) can handle adding any combination of rational and floating point numbers. For this sketch of a specification, this behavior will be ignored in the evaluation and typing rules. However, this behavior will be considered/used later on when considering the FFI boundary.

4.5 Typing

$$\text{(T-FAIL)}\ \frac{}{\Gamma \vdash fail : T}$$

$$\text{(T-FLOAT)}\ \frac{}{\Gamma \vdash float : num_f}$$

$$\text{(T-RATIONAL)}\ \frac{}{\Gamma \vdash rational : num_r}$$

$$\text{(T-VAR)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{(T-ADD)}\ \frac{\Gamma \vdash t_1 : num_f \qquad \Gamma \vdash t_2 : num_f}{\Gamma \vdash t_1 + t_2 : num_f}$$

$$\text{(T-PYADD)}\ \frac{\Gamma \vdash t_1 : num_r \qquad \Gamma \vdash t_2 : num_r}{\Gamma \vdash t_1 +^{PY} t_2 : num_r}$$

$$\text{(T-ABS)}\ \frac{\Gamma, x : T_1 \vdash \tau : T_2}{\Gamma \vdash \lambda x : T_1 . \tau : T_1 \to T_2}$$

$$\text{(T-APP)}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

$$\text{(T-FLAT)}\ \frac{\Gamma \vdash t_1 : num_r}{\Gamma \vdash \text{FLAT } t_1 : num_f}$$

$$\text{(T-RAT)}\ \frac{\Gamma \vdash t_1 : num_f}{\Gamma \vdash \text{RAT } t_1 : num_r}$$

4.6 Compiler TS_{NUM} to Target L_{NUM}

Denoted as $C_{TS \mapsto L}$

4.6.1 Types

$$\begin{array}{rcl} Number & = & num_f \\ T_1 \to T_2 & = & T_1' \to T_2' \end{array}$$

4.6.2 Terms

$$\begin{array}{rcl} x : T & = & x : T' \\ float : Number & = & float : num_f \\ t_1 + t_2 : Number & = & t_1' + t_2' : num_f \\ t_1 t_2 : T & = & t_1' t_2' : T' \\ \lambda x\colon T_1.t : T_1 \to T_2 & = & \lambda x : T_1'.t' : T_1' \to T_2' \end{array}$$

4.7 Compiler PY_{NUM} to Target L_{NUM}

Denoted as $C_{PY \mapsto L}$. Identical to the compiler of TS_{NUM} to L_{NUM} save for translating the "$Number$" type to "num_r".

5 The Source-Type Indexed Logical Relationship

Let τ be any type from TS_{NUM}. Let ρ be any type from PY_{NUM}. Let T be any type from either language.

Let $\mathcal{V}[\![T]\!]$ be the set of all possible values of type T as represented in the operational semantics of the target language L_{NUM}.

Let $\mathcal{E}[\![T]\!]$ be the set of well-typed source-language terms t with type T that, after compilation to L_{NUM}, evaluate to value v of the correct type T as represented in L_{NUM} or $fail$:

$$\forall t \in \mathcal{E}[\![T]\!].((C_{source \mapsto L_{NUM}}(t) \Downarrow v \implies v \in \mathcal{V}[\![T]\!]) \vee ((C_{source \mapsto L_{NUM}}(t) \Downarrow fail)$$

TS_{NUM}

$$\begin{array}{rcl} \mathcal{V}[\![Number]\!] & = & \{n \mid n \text{ is any float}\} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \{(\lambda x\colon \tau_1.t) \mid \forall v \in \mathcal{V}[\![\tau_1]\!].[x \mapsto v]t \in \mathcal{E}[\![\tau_2]\!]\} \end{array}$$

PY_{NUM}

$$\begin{array}{rcl} \mathcal{V}[\![Number]\!] & = & \{n \mid n \text{ is any rational}\} \\ \mathcal{V}[\![\rho_1 \to \rho_2]\!] & = & \{(\lambda x : \rho_1.t) \mid \forall v \in \mathcal{V}[\![\rho_1]\!].[x \mapsto v]t \in \mathcal{E}[\![\rho_2]\!]\} \end{array}$$

6 Soundness

Given a well-typed term in either source language, show that it is semantically well-typed after compiling to L_{NUM} (i.e. evaluates to a value of the right semantic type or terminates with 'fail'). More formally:

1. Given any well-typed term t in TS_{NUM}, show:
   $\Gamma \vdash t : \tau \implies \forall y \in \mathcal{G}[\![\Gamma]\!].y(C_{TS \mapsto L_{NUM}}(t)) \in \mathcal{E}[\![\tau]\!]$
2. Given any well-typed term t in PY_{NUM}, show:
   $\Gamma \vdash t : \rho \implies \forall y \in \mathcal{G}[\![\Gamma]\!].y(C_{PY \mapsto L_{NUM}}(t)) \in \mathcal{E}[\![\rho]\!]$

where $\mathcal{G}[\![\Gamma]\!]$ is the set of all typing contexts Γ' such that given $\Gamma \vdash t : T$ in the source language, assigns the type $T' = C_{source \mapsto L_{NUM}}(T)$ to term $t' = C_{source \mapsto L_{NUM}}(t)$.

6.1 TS_{NUM}

(Compatibility $Number$). $\Gamma \vdash t : Number \implies \forall y \in \mathcal{G}[\![\Gamma]\!].y(C_{TS \mapsto L_{NUM}}(t)) \in \mathcal{E}[\![Number]\!]$

Proof by induction on typing derivations:

- Case T-NUMBER: $t = float$:
  1. Let $t' = C_{TS \mapsto L_{NUM}}(t) = float \in \mathcal{V}[\![Number]\!]$
  2. $t' \in \mathcal{V}[\![Number]\!] \implies t' \in \mathcal{E}[\![Number]\!]$ by definition
- Case T-VAR: $t = x$; $x : Number \in Gamma$
  1. Let $t' = C_{TS \mapsto L_{NUM}}(t) = x$
  2. x is a value, $x : Number \implies x = float \implies t \in \mathcal{E}[\![Number]\!]$ by definition
- Case T-ADD: $t = t_1 + t_2$; $\Gamma \vdash t_1 : Number$; $\Gamma \vdash t_2 : Number$
  1. Let $t'_1 = C_{TS \mapsto L_{NUM}}(t_1)$, $t'_2 = C_{TS \mapsto L_{NUM}}(t_2)$
  2. Let $t' = C_{TS \mapsto L_{NUM}}(t) = C_{TS \mapsto NUM}(t_1) + C_{TS \mapsto L}(t_2) = t'_1 + t'_2$
  3. By the IH, $t'_1 \in \mathcal{E}[\![Number]\!]$ and $t'_2 \in \mathcal{E}[\![Number]\!]$
  4. If $(t'_1 \Downarrow fail)$ or $(t'_2 \Downarrow fail)$, then by the convention noted in L_{NUM}'s evaluation rules:
     $t' \Downarrow fail \implies t' \in \mathcal{E}[\![Number]\!]$
  5. If $(t'_1 \Downarrow float_1)$ and $(t'_2 \Downarrow float_2)$:
     $t' \Downarrow float \implies t' \in \mathcal{E}[\![Number]\!]$
- Case T-APP: $t = t_1 t_2$; $\Gamma \vdash t_1 : T_1 \to Number$; $\Gamma \vdash t_2 : T_1$
  1. Let $t'_1 = C_{TS \mapsto L_{NUM}}(t_1)$, $t'_2 = C_{TS \mapsto L_{NUM}}(t_2)$

2. Let $t' = C_{TS \mapsto L_{NUM}}(t) = C_{TS \mapsto L}(t_1) C_{TS \mapsto L}(t_2) = t'_1 t'_2$
3. By the IH, $t'_1 \in \mathcal{E}[\![T_1 \to Number]\!]$ and $t'_2 \in \mathcal{E}[\![T_1]\!]$
4. If $(t'_1 \Downarrow fail)$ or $(t'_2 \Downarrow fail)$, then by the convention noted in L_{NUM}'s evaluation rules:
   $t' \Downarrow fail \implies t' \in \mathcal{E}[\![Number]\!]$
5. If $(t'_1 \Downarrow (\lambda x : T'_1.t_3))$ and $(t'_2 \Downarrow v \in \mathcal{V}[\![T_1]\!], v \neq fail)$:

   (a) $[x \mapsto v]t_3 \Downarrow float \implies t' \Downarrow float \implies t' \in \mathcal{E}[\![Number]\!]$
   (b) $[x \mapsto v]t_3 \Downarrow fail \implies t' \Downarrow fail \implies t' \in \mathcal{E}[\![Number]\!]$

(Compatibility *Abstraction***).** $\Gamma \vdash t : T_1 \to T_2 \implies \forall y \in \mathcal{G}[\![\Gamma]\!].y(C_{TS \mapsto L_{NUM}}(t)) \in \mathcal{E}[\![T_1 \to T_2]\!]$

Proof by induction on typing derivations:

- Case T-ABS: $t = (\lambda x : T_1.t_2)$; $\Gamma, x : T_1 \vdash t_2 : T_2$
  1. Let $t' = C_{TS \mapsto L_{NUM}}(t) = (\lambda x : T'_1.t'_2)$
  2. Let $v \in \mathcal{V}[\![T_1]\!], v \neq fail$
  3. By the IH, $t'_2 \in \mathcal{E}[\![T_2]\!]$
  4. $v \neq fail$ because per the evaluation conventions for L_{NUM}, x could only be bound to a value during application. If the argument evaluated to $fail$, substitution will never occur.
  5. $[x \mapsto v]t'_2 \Downarrow v' \in \mathcal{V}[\![T_2]\!] \implies t' \in \mathcal{E}[\![T_1 \to T_2]\!]$
- Case T-VAR: $t = x = (\lambda x : T_1.t_2)$; $x : T_1 \to T_2 \in Gamma$
  1. Let $t' = C_{TS \mapsto L_{NUM}}(t) = x'$
  2. x' is a value, $x : T'_1 \to T'_2 \implies x' = (\lambda x.T'_1.t'_2)$
  3. By the IH, $t'_2 \in \mathcal{E}[\![T_2]\!] \implies t \in \mathcal{E}[\![T_1 \to T_2]\!]$ by definition

6.2 PY_{NUM}

The proof is identical to TS_{NUM} save for using the *rational* values and type num_r instead of $float$ values and type num_f.

7 The Convertibility Relationship

Let τ be any type from TS_{NUM}. Let ρ be any type from PY_{NUM}. Let T be any type from either language.

The convertibility of types between PY_{NUM} and TS_{NUM} is written as the symmetric relationship $T_1 \sim T_2$.

Two types ρ and τ can safely be converted between each other iff:

- There exists a translator $C_{\rho \mapsto \tau}$ such that:

$$\forall t \in \mathcal{E}[\![\rho]\!].C_{\rho \mapsto \tau}(t) \in \mathcal{E}[\![\tau]\!]$$

- There exists a translator $C_{\tau \mapsto \rho}$ such that:

$$\forall t \in \mathcal{E}[\![\tau]\!].C_{\tau \mapsto \rho}(t) \in \mathcal{E}[\![\rho]\!]$$

This is the convertibility soundness lemma. These translators may inject runtime code that performs conversions and these conversions can fail (where it would return the 'fail' value).

For TS_{NUM} and PY_{NUM}, we define the following convertibility relationships:

$$\frac{}{(C_{Number_{PY_{NUM}} \mapsto Number_{TS_{NUM}}}, C_{Number_{TS_{NUM}} \mapsto Number_{PY_{NUM}}}) : Number_{PY_{NUM}} \sim Number_{TS_{NUM}}}$$

where

$$C_{Number_{PY_{NUM}} \mapsto Number_{TS_{NUM}}}(e) = \text{FLAT } e$$
$$C_{Number_{TS_{NUM}} \mapsto Number_{PY_{NUM}}}(e) = \text{RAT } e$$

Proof of convertibility soundness is trivial.

$$\frac{\rho_1 \sim \tau_1 \qquad \rho_2 \sim \tau_2}{(C_{(\rho_1+\rho_2)\mapsto(\tau_1+\tau_2)}, ...) : PY_{NUM}(\rho_1 + \rho_2) \sim TS_{NUM}(\tau_1 + \tau_2)}$$

where

$$C_{(\rho_1+\rho_2)\mapsto(\tau_1+\tau_2)}(e_1 + e_2) = C_{\rho_1 \mapsto \tau_1}(e_1) + C_{\rho_2 \mapsto tau_2}(e_2)$$
$$C_{(\tau_1+\tau_2)\mapsto(\rho_1+\rho_2)}(e_1 + e_2) = C_{\tau_1 \mapsto \rho_1}(e_1) +^{PY} C_{\tau_2 \mapsto \rho_2}(e_2)$$

etc...

8 Using the Convertibility Relationship

A complete Pyret program is the tuple (P, FFI, C) where:

1. P is the Pyret program
2. FFI is the set of all pairs $(name, t : \tau)$ where:
   (a) t is a term in TS_{NUM} which P wishes to use

(b) *name* is a binding for t which P will reference to use t

3. C is the set of all compilers defined above (note: this allows different compilers such as $C_{NUMBER_{TS_{NUM}} \mapsto C_{NUMBER_{PY_{NUM}}}}$ which can turn into the identity b/c all floats are binary compatible with rationals)

To type check P with terms referencing name n in FFI bound to $t : \tau$ **expected type** ρ, search C for the compiler $C_{\tau \mapsto \rho}$. If it does not exist, it is a compilation error. If it does exist, wrap program P in an abstraction with parameter $n : \rho$ to make P'. Do this for all elements in FFI.

To execute P', compile P' to L_{NUM} and add the argument $C_{\tau \mapsto \rho}(t)$ for each FFI element.

References

[1] [n. d.] Chart.js. `https://www.chartjs.org/`.

[2] [n. d.] Jetsam. `https://github.com/InnPatron/jetsam`.

[3] [n. d.] js-numbers.js. `https://github.com/brownplt/pyret-lang/blob/7e09ae29d40ce33a2f3fce1e5f2290d1a8b36170/src/runtime/js-numbers.js#L1496`.

[4] [n. d.] js-numbers.js. `https://github.com/brownplt/pyret-lang/blob/7e09ae29d40ce33a2f3fce1e5f2290d1a8b36170/src/runtime/js-numbers.js#L358-L370`.

[5] [n. d.] Matter.js. `https://brm.io/matter-js/`.

[6] Daniel Patterson and Amal Ahmed. 2020. Foreign function typing: Semantic type soundness for FFIs. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20)*. ACM. New York, NY, USA, 12.

[7] [n. d.] Three.js. `https://threejs.org/`.