Many-to-many, Source-to-source, transpilation infrastructure

Francesco Bertolotti

Università degli Studi di Milano, Computer Science Department

T-LADIES Kick-off, Pisa, July 6th 2022

Joint work with Walter Cazzola

Disclaimer#1

This is a work in its very early stages.

So early that this is just an idea.

- If you have any suggestions please let us know.
- If you do not think this is a good idea please let us know.
- Or, if you know similar tools.

Disclaimer#2

I am going to use a little bit of notation from:

- BNF grammars, and
- denotational semantics

With a pinch of abuse.

However, I am no expert with these formalism.

Again, if you see any error let us know.

Library ecosystem

A Library ecosystem represent the libraries available to developers for a specific language.

- Example: Numpy is part of the Python library ecosystem.
- Example: Apache commons is part of the Java library ecosystem.

Most of these libraries are tied to one or a few languages.

Library ecosystems are not interchangeable.

A library from an ecosystem cannot be used in another one.

- Example: You cannot use Numpy in Java.
- Example: You cannot use Apache commons in Python.

At least, not without ad-hoc bindings.

Library ecosystems are overlapping.

Software from different ecosystems offer similar functionalities.

- Example: Java Random class is similar to Python random module.
- Example: Java Nd4j is similar to Python Numpy.

This means that there is a lot of replication.

Library ecosystems development take time.

A mature software ecosystems takes years of community development.

- Example: Numpy development lasted more than 15 years of community work.
- Example: Java still lacks a mature autodiff. library.

New programming languages need a mature ecosystem before becoming compelling.

Library ecosystems are different

Changing programming language means learning a new library ecosystem.

Which takes

- time, and
- practice.

Migratable library ecosystem

With a transpiler, we can render libraries for an ecosystem available to another.

It can be done systematically.

However, we need to build transpilers between languages.

We need many of them.

Migratable library ecosystem

We need an infrastructure that allows for:

- modular, and
- reusable development.

Preliminaries

Languages are formed by stacking language features together.

A language feature is a piece of syntax with a piece of semantics.

A piece of syntax represents form.
(add ← expr "+" expr).

A piece of semantics represents computation.
($[\![a+b]\!](\sigma)) = [\![a]\!](\sigma) + [\![b]\!](\sigma)$).

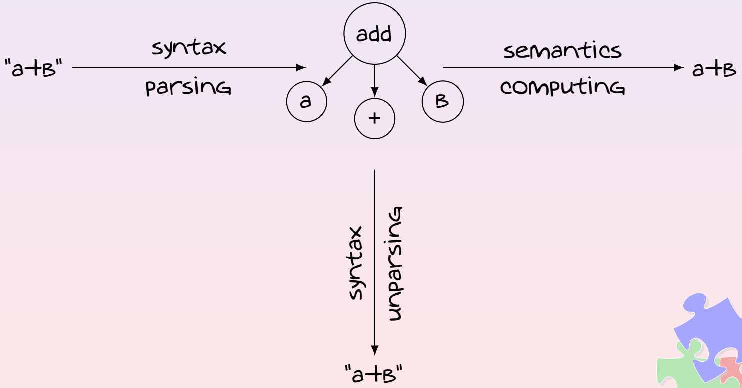
Preliminaries

Many-to-many, Source-to-source, transpilation infrastructure

Francesco Bertolotti

The syntax tells how to parse text.

The semantics tells how to evaluate the parsed text.



Abstract Syntax Tree (AST) represents both:

- the sources, and
- the computation.

Preliminaries

There are infinite languages that could have generated a given AST.

If a language $\mathcal{L}$ can generate the AST T, then we say that T belongs $\mathcal{L}$

$$(T \in \mathcal{L})$$

Problem statement and proposed solution

Problem: Library availability is language-dependent.

Objective: Translating libraries from any language to any language.

Idea

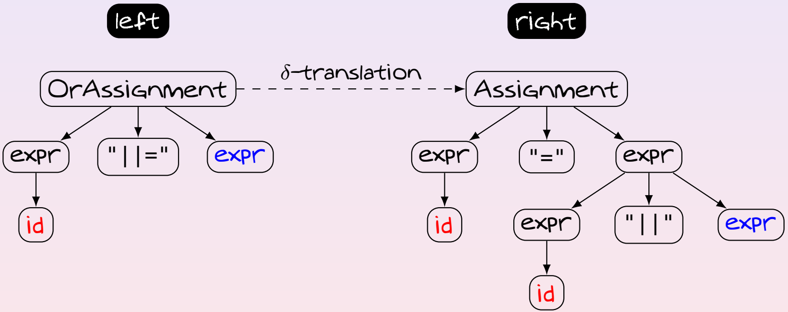
Build a transpilation infrastructure made of small translation functions.

Each function translate a small piece of the AST.

The functions are used by a system to find a transpilation from one language to the other.

δ-translation

A δ-translation is represented as a directed connection between trees. E.G.:



A δ-translation δ on an AST T, $\delta(T)$:

- it pattern matches the left tree.
- it replace the match with the right tree.

δ-translation

A δ-translation is:

- Modular. (one does not affect the other).
- Reusable. (it can be reused in other scenarios).
- Composable. (it can be chained).

δ–translation

A translation is computationally invariant iff:

$\forall x \in id,$
$\forall y \in expr,$
$\forall \sigma \in \Sigma :$
$[\![\delta(OrAss(x, y))]\!](\sigma) = [\![OrAss(x, y)]\!](\sigma)$

Computational invariance cannot be verified.

It is the developer responsibility to write and use computation invariant δs

δ-translation

Applying a δ-translation can change AST language by changing: a language-feature, a μ-language, a sub-language or the entire language.

$$T \in \mathcal{L} \not\Rightarrow \delta(T) \in \mathcal{L}$$

(But it does not change the outcome of execution.)

if-while language

stmt	← *stmt* ";" *stmt*	$[\![x; y]\!](\sigma) = [\![x]\!]([\![y]\!](\sigma))$
	← *assgn*	—
	← *while*	—
	← *if*	—
if	← "if" "(" *expr* ")" "{" *stmt* "}"	$[\![if(x)\{y\}]\!](\sigma) = [\![y]\!](\sigma)$ `if` $x \neq \mathbf{0}$ `else` σ
while	← "while" "(" *expr* ")" "{" *stmt*"}"	$[\![while(x)\{y\}]\!](\sigma) = [\![while(x)\{y\}]\!]([\![y]\!](\sigma))$ `if` $x \neq \mathbf{0}$ `else` σ
assgn	← *id* "=" *expr*	$[\![x = y]\!](\sigma) = \sigma[x \leftarrow y]$
expr	← *expr* "+" *expr*	$[\![x + y]\!](\sigma) = [\![x]\!](\sigma) + [\![y]\!](\sigma)$
	← *expr* "-" *expr*	$[\![x - y]\!](\sigma) = [\![x]\!](\sigma) - [\![y]\!](\sigma)$
	← *expr* "==" *expr*	$[\![x == y]\!](\sigma) = \mathbf{1}$ `if` $x == y$ `else` $\mathbf{0}$
	← *id*	—
	← *int*	—
id	← [a..za..z]+	$[\![x]\!](\sigma) = lit(x)$
int	← [0..9]+	$[\![x]\!](\sigma) = int(x)$

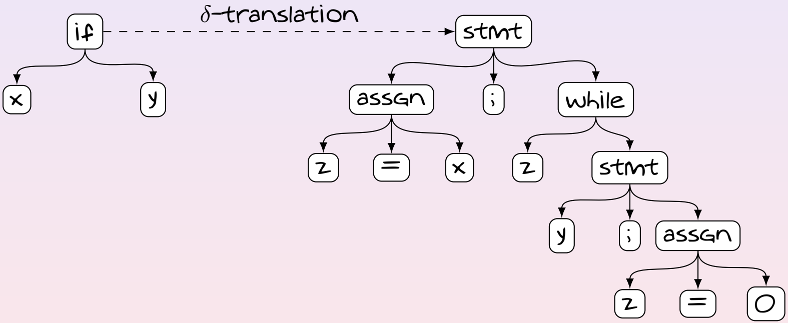
While language

stmt	$\leftarrow$ *stmt* ";" *stmt*	$[\![x; y]\!](\sigma) = [\![x]\!]([\![y]\!](\sigma))$
	$\leftarrow$ *assgn*	—
	$\leftarrow$ *while*	—
while	$\leftarrow$ "while" "(" *expr* ")" "{" *stmt* "}"	$[\![while(x)\{y\}]\!](\sigma) = [\![while(x)\{y\}]\!]([\![y]\!](\sigma))$ `if` $x \neq \mathbf{0}$ `else` σ
assgn	$\leftarrow$ *id* "=" *expr*	$[\![x = y]\!](\sigma) = \sigma[x \leftarrow y]$
expr	$\leftarrow$ *expr* "+" *expr*	$[\![x + y]\!](\sigma) = [\![x]\!](\sigma) + [\![y]\!](\sigma)$
	$\leftarrow$ *expr* "-" *expr*	$[\![x - y]\!](\sigma) = [\![x]\!](\sigma) - [\![y]\!](\sigma)$
	$\leftarrow$ *expr* "==" *expr*	$[\![x == y]\!](\sigma) = \mathbf{1}$ `if` x `==` y `else` $\mathbf{0}$
	$\leftarrow$ *id*	—
	$\leftarrow$ *int*	—
id	$\leftarrow$ `[a..zA..Z]+`	$[\![x]\!](\sigma) = lit(x)$
int	$\leftarrow$ `[0..9]+`	$[\![x]\!](\sigma) = int(x)$

if language

stmt	$\leftarrow$ *stmt* ";" *stmt*	$[\![x; y]\!](\sigma) = [\![x]\!]([\![y]\!](\sigma))$
	$\leftarrow$ *assgn*	—
if	$\leftarrow$ "if" "(" *expr* ")" "{" *stmt* "}"	$[\![if(x)\{y\}]\!](\sigma) = [\![y]\!](\sigma)$ if $x \neq \mathbf{0}$ else σ
if	$\leftarrow$ "if" "(" *expr* ")" "{" *stmt* "}"	$[\![if(x)\{y\}]\!](\sigma) = [\![y]\!](\sigma)$ if $x \neq \mathbf{0}$ else σ
assgn	$\leftarrow$ *id* "=" *expr*	$[\![x = y]\!](\sigma) = \sigma[x \leftarrow y]$
expr	$\leftarrow$ *expr* "+" *expr*	$[\![x + y]\!](\sigma) = [\![x]\!](\sigma) + [\![y]\!](\sigma)$
	$\leftarrow$ *expr* "-" *expr*	$[\![x - y]\!](\sigma) = [\![x]\!](\sigma) - [\![y]\!](\sigma)$
	$\leftarrow$ *expr* "==" *expr*	$[\![x == y]\!](\sigma) = \mathbf{1}$ if $x == y$ else $\mathbf{0}$
	$\leftarrow$ *id*	—
	$\leftarrow$ *int*	—
id	$\leftarrow$ [a..za..z]+	$[\![x]\!](\sigma) = lit(x)$
int	$\leftarrow$ [0..9]+	$[\![x]\!](\sigma) = int(x)$

(if → while)-translation



(z must be a unique id)

if-while snippet to while snippet


```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

if-while snippet to while snippet


```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;

    z = x == 1;
    while(z) {
        flag = 0;
        z = 0;
    }
}
```

if-while snippet to while snippet


```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;

    z = x == 1;
    while(z) {
        flag = 0;
        z = 0;
    }
}
```

while

if-while

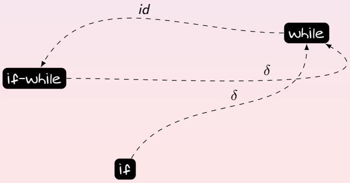
if

if-while snippet to while snippet


```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;

    z = x == 1;
    while(z) {
        flag = 0;
        z = 0;
    }
}
```

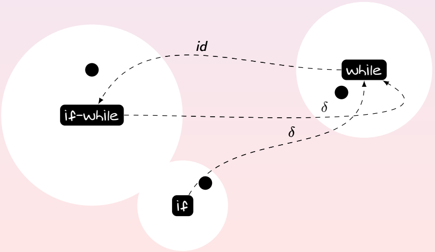


if-while snippet to while snippet

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;

    z = x == 1;
    while(z) {
        flag = 0;
        z = 0;
    }
}
```

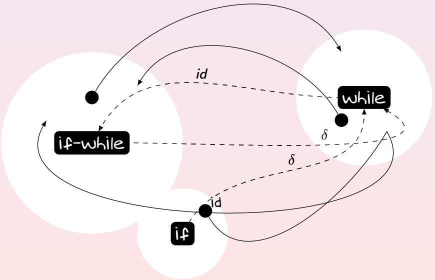


if-while snippet to while snippet

```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;
    if (x == 1) {flag = 0};
}
```

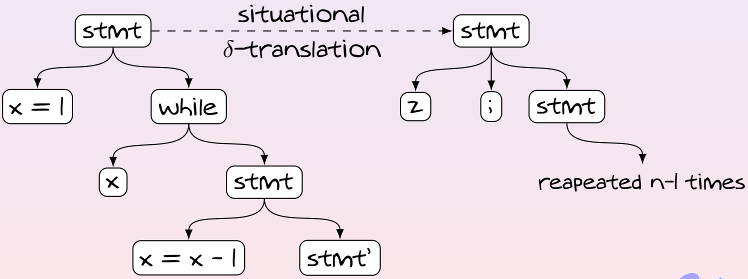
```
flag = 1;
x = 10;
while(flag) {
    x = x - 1;

    z = x == 1;
    while(z) {
        flag = 0;
        z = 0;
    }
}
```



Situational δ-translation

Situational δ-translations are translations that are only aplicable under few circumstances. For example:



Provided that stmt' does not modify x.

Situational δ-translation

Situational δ-translations are not reliable to translate one language into another.

Situational δ-translation may succeed into translating only in certain situations.

Situational δ-translation


```
x = 3;
y = 0;
while(x) {
    x = x - 1;
    y = y + 2;
}
```

Situational δ-translation

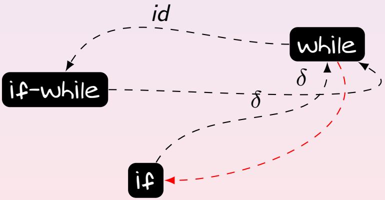

```
x = 3;
y = 0;
while(x) {
    x = x - 1;
    y = y + 2;
}
```

```
y = 0;
y = y + 2;
y = y + 2;
y = y + 2;
```

Situational δ-translation

```
x = 3;
y = 0;
while(x) {
    x = x - 1;
    y = y + 2;
}
```

```
y = 0;
y = y + 2;
y = y + 2;
y = y + 2;
```



δ-alternatives

A δ-translation may have alternatives that are user dependent.

For example:

- The user may have a preference on the generation of the identifier z.
- Or, it may want to use a different translation pattern.

Transpilation Product Lines

We need to model the δ-variability that can occur.

Different δ-translation lead to different products.

Feature models seems a good model for this kind of variability.

Conclusion

δs can be used to translate any language to any other language.

δs are reusable, modular but have alternatives.

δs can be chained to build new transpilers.

The target language can be expressed declaratively.

δs Product lines can be used to model alternatives.

Thank you for your attention.