# Lecture: RSC & I/O

Marco Patrignani

## 1 Languages

We extend the languages from the Backtranslation lecture notes with I/O primitives, specifically with a *print n* primitive that writes a number in an external buffer. That buffer is not readable from either the program of the context. That buffer is readable from the so-called *environment*.

All the recent additions such as heap, multiple calls etc are orthogonal to this development, so they are rolled back for simplicity.

### 1.1 Source

$$\mathsf{e} ::= \cdots \mid \mathsf{print\ e}$$
$$\mathsf{E} ::= \cdots \mid \mathsf{print\ E}$$

#### 1.1.1 Static Semantics (Typing)

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{Nat}}{\Gamma \vdash \mathsf{print\ e} : \mathsf{Nat}}\ (\mathsf{T\text{-}print})$$

#### 1.1.2 Dynamic Semantics

We need to accumulate the prints as they happen as the program executes. This means turning the regular semantics into a labelled semantics, where each reduction is labelled with a label $\lambda$ describing it. This is analogous to traces.

$$\lambda ::= \epsilon \mid \mathit{print}\ n$$

The semantics accumulates labels in a sequence which we call a behaviour $\overline{\lambda}$, where we implicitly assume that silent labels $\epsilon$ are pruned. The empty behaciour, i.e., those done by the program that never prints, is marked as $\varnothing$.

Judgement: $\mathsf{e} \xrightarrow{\overline{\lambda}} \mathsf{e'}$

$$\mathsf{e} \xrightarrow{\epsilon} \mathsf{e'} \text{ if } \mathsf{e} \hookrightarrow \mathsf{e'}$$
$$\mathsf{print\ n} \xrightarrow{\mathsf{print\ n}} \mathsf{n}$$

### 1.1.3 First Concern: Behaviour Origin

Since print e happens both in programs and context, the first concern is that we cannot blindly accumulate them.

A first solution to this is to assume this expression cannot occur in contexts.

Another solution is to annotate the label with an annotation of the form program or context and then eliminate the context ones when calculating behaviours.

In this document we assume the first solution. Both are equivalent, the only difference is whether the resulting model is a faithful model of a realistic scenario.

## 1.2 Target

The same primitive is added to the target and a semantic rule is added too.

## 1.3 Common

Plugging in unchanged. The notion of program equivalence is also unchanged: we still want to know if a *context* can tell two programs apart, not whether an *environment* can.

**Definition 1.1 (Contextual equivalence).**

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall O, \overline{\lambda}, \overline{\lambda'}.\ O\left[P_1\right] \xrightarrow{\overline{\lambda}} f \wedge O\left[P_2\right] \xrightarrow{\overline{\lambda'}} f$$

As for the previous case of RSC, we define the behaviour of a *single* program as what the *environment* can observe.

$$\texttt{Behav}\left(O[P]\right) \stackrel{\text{def}}{=} \left\{ \overline{\lambda} \;\middle|\; O[P] \xrightarrow{\overline{\lambda}} \_ \right\}$$

# 2 Compiler

The compiler translates source print to the target level analogous.

$$\ldots$$

$$[\![\text{print e}]\!]^{\mathsf{S}}_{\mathbf{T}} = \mathbf{print}\ [\![\text{e}]\!]^{\mathsf{S}}_{\mathbf{T}}$$

## 2.1 Compiler Properties

At this point, since we have added cases to the compiler, we need to add the missing cases to the compiler correctness proof and to the auxiliary lemmas. These additions are straightforward.

# 3 Backtranslation

In this case we have a single behaviour to backtranslate into a single source context.

Consider this program and its compilation

$\mathsf{f(x)} \mapsto \mathsf{let\ y{=}if\ x > 2\ then\ print\ 3\ else\ print\ 4\ in\ 0}$

$\mathbf{f(x) \mapsto if\ x\ has\ Nat\ then\ let\ y{=}if\ x > 2\ then\ print\ 3\ else\ print\ 4\ in\ 0\ else\ fail}$

Its behaviours are: $\{\varnothing, \mathbf{print\ 3}, \mathbf{print\ 4}\}$.

If we try to backtranslate based simply on the behaviours, we can create a context but we cannot be sure that the context will perform the right interaction with the source program. For example, the backtranslation of **print 3** should generate a context that calls function f with argument 3 (or more). On the other side, the backtranslation of **print 4** should generate a context that calls function f with argument 1 (or less). Unfortunately, the behaviours themselves do not carry any information that lets us know what argument to use.

While in this trivial example we could come up with a program analysis to address this issue, this is not true for arbitrary languages. In fact, in more complex programs, understanding which statement will lead to which behaviour is reconducible to the halting problem.

So, in order to address this issue, the most common trick is to emply so-called *informative traces*. This means extending behaviours to record context-program interaction as well as behaviours. The backtranslation then uses those informative traces.

Informative traces are conceptually like those we have already seen, so we do not present them here.