Fundamental Constructs in Programming Languages

Extracts for testing MathJax

Peter D. Mosses[1,2] [0000−0002−5826−7520]

[1] Delft University of Technology, The Netherlands
[2] Swansea University, United Kingdom
p.d.mosses@swansea.ac.uk

Abstract. The body of this document consists of fragments of the published article. It is for use in testing the formatting of mathematical formulae in paragraphs, inline displays, and floating figures by LaTeX and MathJax.

1 Introduction

Version control is superfluous for funcons; translations of language constructs to funcons, in contrast, may need to change when the specified language evolves. For example, the illustrative language Imp includes a plain old while-loop with a Boolean-valued condition: '`while(`*BExp*`)` *Block*'. The following rule translates it to the funcon while-true, which has exactly the required behaviour:

Rule $\quad \mathsf{execute}[\![\ \text{`}\texttt{while}\text{'}\ \text{`}\texttt{(}\text{'}\ \mathit{BExp}\ \text{`}\texttt{)}\text{'}\ \mathit{Block}\ ]\!] =$
$\qquad \mathsf{while\text{-}true}(\mathsf{eval\text{-}bool}[\![\ \mathit{BExp}\ ]\!], \mathsf{execute}[\![\ \mathit{Block}\ ]\!])$

The behaviour of the funcon while-true is fixed. But suppose the Imp language evolves, and a *Block* can now execute a statement '`break;`', which is supposed to terminate just the *closest* enclosing while-loop. We can extend the translation with the following rule:

Rule $\quad \mathsf{execute}[\![\ \text{`}\texttt{break}\text{'}\ \text{`}\texttt{;}\text{'}\ ]\!] = \mathsf{abrupt}(\mathsf{broken})$

The translation of '`while(true){break;}`' is while-true(true, abrupt(broken)). The funcon abrupt(V) terminates execution abruptly, signalling its argument value V as the reason for termination. However, the behaviour of while-true(true, X) is to terminate abruptly whenever X does – so this translation would lead to abrupt termination of *all* enclosing while-loops!

We cannot change the definition of while-true, so we are forced to change the translation rule. The following updated translation rule reflects the extension of the behaviour of while-loops with the intended handling of abrupt termination

due to break-statements, and that they propagate abrupt termination for any other reason:

Rule execute⟦ '`while`' '`(`' *BExp* '`)`' *Block* ⟧ =
handle-abrupt(
while-true(eval-bool⟦ *BExp* ⟧, execute⟦ *Block* ⟧),
if-true-else(is-equal(given, broken), null-value, abrupt(given)))

Computing null-value represents normal termination; given refers to the reason for the abrupt termination.

The specialised funcon handle-break can be used to specify the same behaviour more concisely:

Rule execute⟦ '`while`' '`(`' *BExp* '`)`' *Block* ⟧ =
handle-break(while-true(eval-bool⟦ *BExp* ⟧, execute⟦ *Block* ⟧))

Wrapping execute⟦ *Block* ⟧ in handle-continue would also support abrupt termination of the current *iteration* due to executing a continue-statement.

2 The Nature of Funcons

Funcons are often independent, but not always. For instance, the definition of the funcon while-true specifies the reduction of while-true(B, X) to a term involving the funcons if-true-else and sequential:

Funcon while-true$(B :\Rightarrow$ booleans$, X :\Rightarrow$ null-type$) :\Rightarrow$ null-type
$\rightsquigarrow$ if-true-else$(B$, sequential$(X$, while-true$(B, X))$, null-value$)$

3 Collections of Funcons

4 Facets of Funcons

5 Translation of Language Constructs to Funcons

The translation specification in Fig. 1 declares exp as a phrase sort, with the meta-variable *Exp* (possibly with subscripts and/or primes) ranging over phrases of that sort. The BNF-like production shows two language constructs of sort exp: an identifier of sort id (lexical tokens, here assumed to be specified elsewhere with meta-variable *Id*) and a function application written 'Exp_1 `(` Exp_2 `)`'.

The translation specification for function declarations in Fig. 2 assumes a translation function exec⟦*Block*⟧ for phrases *Block* of sort block. A block is a statement, which normally computes a null value; but here, as in many languages, a block can return an expression value by executing a return statement, which terminates the execution of the block abruptly.

Syntax *Exp* : exp ::= ··· | id | exp '(' exp ')' | ···

Semantics rval⟦ _ : exp ⟧ : ⇒ values

Rule rval⟦ *Id* ⟧ = assigned-value(bound-value(id⟦ *Id* ⟧))

Rule rval⟦ Exp_1 '(' Exp_2 ')' ⟧ = apply(rval⟦ Exp_1 ⟧, rval⟦ Exp_2 ⟧)

Fig. 1. Translation of identifiers and function applications in SIMPLE to funcons

Syntax *Decl* : decl ::= ··· | 'function' id '(' id ')' block

Semantics declare⟦ _ : decl ⟧ : ⇒ environments

Rule declare⟦ 'function' Id_1 '(' Id_2 ')' *Block* ⟧ =

```
bind-value(id⟦ Id1 ⟧,
  allocate-initialised-variable(functions(values, values),
    function(closure(
      scope(
        bind-value(id⟦ Id2 ⟧,
          allocate-initialised-variable(values, given)),
        handle-return(exec⟦ Block ⟧)))))))
```

Fig. 2. Translation of function declarations in SIMPLE to funcons

6 Defining and Implementing Funcons

The funcon signature in Fig. 3 specifies that scope takes two arguments. The first argument is required to be pre-evaluated to a value of type environments; the second argument should be unevaluated, as indicated by '$\Rightarrow T$'. Values computed by $\mathsf{scope}(\rho_1, X)$ are to have the same type (T) as the values computed by X.

Funcon $\mathsf{scope}(\_ : \mathsf{environments}, \_ : \Rightarrow T) : \Rightarrow T$

Rule $$\frac{\mathsf{environment}(\mathsf{map\text{-}override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \mathsf{scope}(\rho_1 : \mathsf{environments}, X) \longrightarrow \mathsf{scope}(\rho_1, X')}$$

Rule $\mathsf{scope}(\_ : \mathsf{environments}, V : T) \rightsquigarrow V$

Fig. 3. Definition of the funcon for expressing scopes of local declrations

The rules define how evaluation of $\mathsf{scope}(\rho_1, X)$ can proceed when the current bindings are represented by ρ_0. The premise of the first rule holds if X can make a transition to X' when ρ_1 overrides the current bindings ρ_0. Whether X' is a computed value or an intermediate term is irrelevant. When the premise holds, the conclusion is that $\mathsf{scope}(\rho_1, X)$ can make a transition to $\mathsf{scope}(\rho_1, X')$.

7 Related Work

8 Conclusion