Type and Effect System for Multi-staged Exceptions*

Hyunjun Eo[1], Ik-Soon Kim[2], and Kwangkeun Yi[1]

[1] Seoul National University, Korea
[2] École Polytechnique, France

Abstract. We present a type and effect system for a multi-staged language with exceptions. The proposed type and effect system checks if we safely synthesize complex controls with exceptions in multi-staged programming. The proposed exception constructs in multi-staged programming has no artificial restriction. Exception-raise and -handle expressions can appear in expressions of any stage, though they are executed only at stage 0. Exceptions can be raised during code composition and may escape before they are handled. Our effect type system support such features. We prove our type and effect system sound: empty effect means the input program has no uncaught exceptions during its execution.

1 Introduction

Staged computation, which explicitly divides a computation into separate stages, is a unifying framework for existing program generation systems: partial evaluation [5,1], run-time code generation [7,10], function inlining and macro expansion [11,3] are all instances of staged computation. The stage levels are determined by the nesting depth of program generations: stage 0 generates a program of stage 1 that generates a program of stage 2, and so on. The key aspect of multi-staged language is to have code templates (program fragments) as first-class objects. Code templates are freely passed, composed with code of other stages, and executed. At stage 0, computation include all normal computation plus generating code and executing generated code. At stage > 0, computation is just code-composition: it just visits expression's sub-expressions and substitutes code into code when appropriate.

Example 1. As a specializer example in multi-stage programming, consider a recursive `map` function:

```
fun map f nil = nil
  | map f (x::r) = (f x) :: (map f r)
```

* Eo and Yi were partially supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R& D Support Project of Korea Ministry of Information and Communication, by Korea National Security Research Institute, and by Microsoft Research Asia. Kim was supported by post-doctoral grants from École Polytechnique and École Normale Supérieure in France.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 61–78, 2006.

The `map` function applies function `f` to each element in the input list, and builds a list with the results returned by `f`. If we know which list is available, we can specialize `map` function with the input list. For example, if input list is `1::2::nil`, we specialize the `map` function to `fn f => (f 1)::(f 2):::nil`. The specialized function is more efficient than the original `map` function because it does not need to traverse the list structure. This specialization can be achieved by the following two functions in Lisp's quasi-quote syntax [11]:

```
fun map_ls nil = `nil
  | map_ls (x::r) = `((f ,x) :: ,(map_ls r))
fun smap ls = eval `(fn f => ,(map_ls ls))
```

At stage 0, the function `smap`, along with `map_ls`, traverses input list `ls` and generates a specialized function of stage 1: the stage increases by the number of surrounding backquotes (`), and decreases by the number of commas (,). Because the application `(f ,x)` in `map_ls` is at stage 1 (surrounded by one backquote), it will not be evaluated. However, the recursive call `(map_ls r)` will be evaluated because it is at stage 0 (surrounded by one backquote and one comma). ■

Exception handling allows the programmer to define, raise and handle exceptional conditions. Exceptional conditions are brought (by a raise expression) to the attention of another expression where the raised exceptions may be handled. Raised exceptions abort the usual program continuation, transfer ("long jump") the control to its handling point, and continue there with the handler expression. Hence by using exceptions programmers can divert any control structure to a point where the corresponding exception is handled. The exception facilities, however, can provide a hole for program safety. Programs can abruptly halt when an exception is raised and never handled.

In this paper we extend the Lisp-like multi-staged language λ^{sim}_{open} [6] with such exceptions and then present a sound type and effect system that statically estimate may-uncaught exceptions in the input programs.

The proposed exception facility in the multi-staged language has no artificial restriction. Lexically, exception-raise and -handle expressions can appear in expressions of any stage. Only restriction, which is natural, is on their dynamics: exceptions must be raised and handled only at stage 0 (at normal computation). Hence, the most interesting feature of our language is exceptions raised during code composition. During computation at stage > 0 (during code composition) an expression can be brought to stage 0 and evaluated there to return a code to substitute for the expression at the code composition. During this stage-0 evaluation an exception can be raised. This raised exception can be caught by a handler only at stage 0. Which handler is that? Any handler at stage 0 in the continuation of the raised exception. A handler that is installed during the stage-0 evaluation can catch it. Or, a handler that is installed at stage 0 before the code composition can catch it and continue.

Example 2. We explain this staged exception semantics by an example. The following function `f` gets a list `ls` and generates a code that multiplies free variable `a` with every element in `ls`.