C3 Notes

Peter Seymour

2 February 2011

Last updated: 02 February 2011

1 Introduction

C3 (compact character code) can be expressed using only base-64 characters and implements a variant of System F with sub-typing and objects. Programs in a wide variety of statically typed languages can be compiled to C3 easily. It supports continuations natively for concurrency and compiler implementation. Since it is a high-level intermediate language it is suitable for efficiently defining programs in dynamic languages with first class functions.

The code can parsed lexically without reference to the type system making it ideal for fast just-in-time compilation.

2 Definitions

Table 1: Type Generators

Symbol	Description
$A_k\tau_1 \dots \tau_k\tau$	Type quantification
G_k	Generator reference

Table 2: Type Expressions

Symbol	Description
U	Unit type
$F_k \overbrace{c_{64} \dots c_{64}}^{k}$	Foreign type with base-64 identifier
B	Bytes type
$O\tau^X\tau^X$	Object type (initial values for read/write and values for read-only)
$X_k\tau_1 \dots \tau_k$	Cartesian type (read-only elements)
$S_k\tau_1 \dots \tau_k$	Structure type (read/write elements)
$I_kG\tau_1 \dots \tau_k$	Instantiated type generator
$L_k\tau_1 \dots \tau_k\tau$	Lambda
$M_k\tau_1 \dots \tau_k\tau$	Member lambda
$N_k\tau_1 \dots \tau_k$	Continuation
T_n	Type reference

Table 3: Value Generators

Symbol	Description
$a_k \tau_1 \ldots \tau_k v$	Value quantification
g_n	Value generator reference

Table 4: Value Expressions

Symbol	Description
$b_k \ \overbrace{c_{64} \ldots c_{64}}^{min\{j\|4j \geq 3k\}}$	Bytes specified by base-64
$n_k \tau_1 \ldots \tau_k \beta$	Continuation ($e \notin \beta$ and returns C_0)
$l_k \tau_1 \ldots \tau_k \tau \beta$	Lambda
$e_k \alpha^{L_k} \alpha_1 \ldots \alpha_k$	Evaluate k-ary lambda
$h \alpha^{\tau} \alpha^{L_0 \tau}$	Handler for expression calling 0-ary lambda on exception
$i_k g \tau_1 \ldots \tau_k$	Instantiated value generator
$f_k \tau \alpha_1 \ldots \alpha_k$	Foreign k-ary function evaluation (identified by type)
$c_k t^O \alpha_1 \ldots \alpha_k$	Constructor for object type (requires instance read-only values, returns lambda from read/write values to object type)
$w_k \tau^S v$	Write k-th element
$r_k \tau^X v$	Read k-th element
t	Throw
$y_k \alpha^{L_k} \alpha_1 \ldots \alpha_k$	Curry k-ary lambda
$x_k \alpha_1 \ldots \alpha_k$	Cartesian constructor
$s_k \alpha_1 \ldots \alpha_k$	Structure constructor
u	Unit canonical instance
$k_k \alpha^O$	Read k-th element of read-only object value
α_n	Value reference

3 Sub-typing

$$C_k \sigma_1 \ldots \sigma_k <: L_k \tau_1 \ldots \tau_k C_0$$

$$X_j \sigma_1 \ldots \sigma_j <: X_k \tau_1 \ldots \tau_k \iff j \geq k \ and \ \forall i \ \sigma_i <: \tau_i$$

$$S_j \sigma_1 \ldots \sigma_j <: X_k \tau_1 \ldots \tau_k \iff j \geq k \ and \ \forall i \ \sigma_i := \tau_i$$

$$O \sigma \tau <: \sigma$$

$$\forall \tau \ \tau <: U$$

4 Event Model

Each process is a lambda or continuation returning a 0 argument function for the next step. Note that continuations must evaluate in finite time unless a foreign function fails to halt. A sparse matrix of callbacks implements a mapping of process and event pairs. The scheduler choses a matrix cell to next evaluate, queuing events if necessary. Note the scheduler cannot be a continuation as it evaluates the process continuations. Trusted applications can use co-operative threading (lambda) whereas untrusted applications use pre-emptive threading (continuation). All applications can call trusted code via an appropriate foreign function if made available.