1 Iterative Deepening for Functional Programming Enumeration

We present here an detailed exposition of an iterative deepening type algorithm for enumerating functional programs.

Algorithm 1 Cost bounded depth first search on typed functional programs.

Require: G a grammar of typed, weighted primitives. A weight γ associated with a new application. A requested type τ. A current substitution σ.

1: **function** CBSEARCH(G, b, τ, σ) ▷ A cost bounded depth first search.
2: $out = []$
3: $c_{min} \leftarrow \infty$
4: **for** $e, \eta, w \in G$ **do** ▷ τ is the type of e
5: $\sigma' \leftarrow unify(\eta, \tau, \sigma)$
6: **if** $null(\sigma)$) **then** ▷ If types can be unified.
7: continue
8: **else**
9: $out \leftarrow cons(out, (e, \sigma', w))$
10: **if** $w > b$ **then**
11: $c_{min} \leftarrow min(w, c_{\min})$
12: **end if**
13: **end if**
14: **end for**
15: **if** $\gamma > b$ **then**
16: $c_{min} \leftarrow min(\gamma, c_{\min})$
17: **return** (out, c_{min})
18: **else**
19: (lhss, k_ℓ) $\leftarrow$ DBSEARCH($G, b - \gamma, \eta \to \tau, \sigma$)
20: $c_{min} \leftarrow \gamma + k_\ell$
21: **for** $e_\ell, \sigma_\ell, w_\ell \in$ lhss **do**
22: **if** $\gamma + w_\ell \leq b$ **then**
23: (rhss, k_r) $\leftarrow$ DBSEARCH($G, b - \gamma - w_\ell, \eta, \sigma_\ell$)
24: **if** $\gamma + w_\ell + k_r > b$ **then**
25: $c_{min} \leftarrow min(c_{min}, \gamma + w_\ell + k_r)$
26: **end if**
27: **for** $e_r, \sigma_r, w_r \in$ rhss **do**
28: **if** $\gamma + w_\ell + w_r \leq b$ **then**
29: out $\leftarrow cons(out, (app(e_\ell, e_r), \sigma_r, \gamma + w_\ell + w_r))$
30: **end if**
31: **end for**
32: **end if**
33: **end for**
34: **return** (out, c_{min})
35: **end if**
36: **end function**

Algorithm 2 Cost bounded depth first search on typed functional programs, version 2. This version adds a modest bounding heuristic cost. Every expansion carries with it an induced cost. We modify the previous version in the following way: every time we proceed with an application, we reduce the available bound by the minimum required to complete the application if it comes back successfully. In this first version, we just reduce the bound by the smallest primitive in the grammar.

Require: G a grammar of typed, weighted primitives. A weight γ associated with a new application. A requested type τ. A current substitution σ.

```
1:  function cbSearch(G, b, τ, σ)                    ▷ A cost bounded depth first search.
2:      out = []
3:      c_min ← ∞
4:      for e, η, w ∈ G do                                  ▷ τ is the type of e
5:          σ' ← unify(η, τ, σ)
6:          if null(σ)) then                                ▷ If types can be unified.
7:              continue
8:          else
9:              out ← cons(out, (e, σ', w))
10:             if w > b then
11:                 c_min ← min(w, c_min)
12:             end if
13:         end if
14:     end for
15:     if γ + w_min > b then
16:         c_min ← min(γ + wMin, c_min)
17:         return (out, c_min)
18:     else
19:         w_min ← min{w|(_, _, w) ∈ G}
20:         (lhss, k_ℓ) ← dbSearch(G, b − γ − w_min, η → τ, σ)
21:         c_min ← γ + k_ℓ
22:         for e_ℓ, σ_ℓ, w_ℓ ∈ lhss do
23:             if γ + w_ℓ ≤ b then
24:                 (rhss, k_r) ← dbSearch(G, b − γ − w_ℓ, η, σ_ℓ)
25:                 if γ + w_ℓ + k_r > b then
26:                     c_min ← min(c_min, γ + w_ℓ + k_r)
27:                 end if
28:                 for e_r, σ_r, w_r ∈ rhss do
29:                     if γ + w_ℓ + w_r ≤ b then
30:                         out ← cons(out, (app(e_ℓ, e_r), σ_r, γ + w_ℓ + w_r))
31:                     end if
32:                 end for
33:             end if
34:         end for
35:         return (out, c_min)
36:     end if
37: end function
```