A Correspondence between Type Checking via Reduction and Type Checking via Evaluation

Accompanying code overview

Ilya Sergey
Dave Clarke

Report CW 617, January 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Correspondence between Type Checking via Reduction and Type Checking via Evaluation

Accompanying code overview

Ilya Sergey
Dave Clarke

Report CW 617, January 2012

Department of Computer Science, K.U.Leuven

Abstract

This is an accompanying technical report for the paper with the corresponding title, published in Information Processing Letters, volume 112, issues 1–2, pages 13–20. This document contains detailed listings of different semantic artifacts for type checking with explanations on the performed transformations.

Keywords : compositional evaluators, type checkers, continuation-passing style, defunctionalization, refunctionalization

A Correspondence between Type Checking via Reduction and Type Checking via Evaluation

Accompanying code overview

Ilya Sergey   Dave Clarke

Dept. Computer Science,
Katholieke Universiteit Leuven
{`firstname.lastname`}`@cs.kuleuven.be`

This is an accompanying technical report for the paper with the corresponding title [1]. This document contains detailed listings of different semantic artifacts for type checking with explanations on the performed transformations.

1 Introduction

This technical report provides a detailed implementation of the original reduction semantics for type checking and the corresponding semantics-preserving transformations. The report itself and the accompanying code is available from `http://people.cs.kuleuven.be/ilya.sergey/type-reduction/`.

2 A Reduction-Based Type Checker

In this section we provide the implementation of a hybrid language for the simply typed lambda calculus, a notion of closures in it and a corresponding reduction semantics via contraction as a starting point for further transformations.

The reduction-based normalization of hybrid terms is implemented by providing an abstract syntax, a notion of contraction and a reduction strategy. Then we provide a one-step reduction function that decomposes a non-value closure into a potential redex and a reduction context, contracts the potential redex, if it is actually one, and then recomposes the context with the contractum. Finally we define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value (i.e., an actual type of an expression) is reached.

2.1 Plain syntax

This section describes elements of the syntax of the traditional λ-calculus. The abstract syntax for $\lambda_{\mathcal{H}}$includes integer literals, identifiers, lambda-abstractions, applications as well as "hybrid" elements such as numeric types and arrows $\tau \to e$. Types are either numeric types or arrow types. The special value `T_ERROR s` is used for typing errors; it cannot be a constituent of any other type. Typing environments `TEnv` represent bindings of identifiers to types, which are values in the hybrid language. In order to keep to the uniform approach for different semantics for type inference, we leave environments parametrized by the type parameter `'a`, which is instantiated with `typ` in this case.

syntax.sml

```
structure Syn =
struct
datatype typ = T_NUM
             | T_ARR of typ * typ
             | T_ERROR of string

datatype term = LIT of int
              | IDE of string
              | LAM of string * typ * term
              | APP of term * term

end

structure TEnv =
struct
type 'a gamma = (string * 'a) list

val empty = []

fun extend (x, t, gamma) = (x, t) :: gamma

fun lookup (x, gamma)
    = let fun search []
              = NONE
            | search ((x', t) :: gamma)
              = if x = x' then SOME t else search gamma
      in search gamma
      end
end

(* Example terms for testing *)
local open Syn
in

(*  T_ARR (T_ARR (T_NUM,T_NUM),T_NUM)  *)
val term1 = LAM ("z", T_ARR(T_NUM, T_NUM), APP (IDE "z", LIT 42))

(*  T_ARR (T_ARR (T_NUM,T_NUM),T_ARR (T_NUM,T_NUM))  *)
val term2 = LAM ("y", T_ARR (T_NUM, T_NUM), IDE "y")

(*   T_ARR (T_NUM,T_NUM)  *)
val term3 = LAM("x", T_NUM, IDE "x")

(*  T_NUM  *)
val term4 = APP(term1, APP (term2, term3))
end
```

2.2 Hybrid syntax

We introduce closures into the hybrid language in order to represent the environment-based reduction system. A closure can either be a number, a ground closure pairing a term and an environment, a combination of closures, a closure for a hybrid arrow expression, or a closure for a value arrow element, namely an arrow type. A value in the hybrid language is either an integer or a function type. Environments bind identifiers to values. A context is a closure with a hole, represented inside-out in a zipper-like fashion.

hsyntax.sml

```
(* Hybrid syntax *)
structure HSyn =
struct
open Syn TEnv

datatype hterm = H_LIT of int
```

```
               | H_IDE of string
               | H_LAM of string * typ * hterm
               | H_APP of hterm * hterm
               | H_TARR of typ * hterm
               | H_TNUM

datatype closure = CLO_NUM
                 | CLO_GND of hterm * bindings
                 | CLO_APP of closure * closure
                 | CLO_ARR of typ * closure
                 | CLO_ARR_TYPE of typ
withtype bindings = typ TEnv.gamma

datatype hctx = CTX_MT
              | CTX_FUN of hctx * closure
              | CTX_ARG of typ * hctx
              | CTX_ARR of typ * hctx
end
```

2.3 Reduction semantics for type checking

A potential redex is either a numeric literal, a ground closure pairing an identifier and an environment, an application of a value to another value, a lambda-abstraction to be type-reduced, an arrow type, or a ground closure pairing a term application and an environment. A potential redex may trigger a contraction or it may get stuck.

The contraction function `contract` reflects the type-checking reduction rules for $\lambda_{\mathcal{H}}$. For instance, any integer literal contracts to a number type `T_NUM`, a lambda expression contracts to an arrow expression of the hybrid language, and the contraction of a potential redex `PR_APP` checks whether its first parameter is a function type and its parameter type matches the argument of the application. A non-value closure is stuck when an identifier does not occur in the current environment or non-function type is used in a function position or a function parameter's type does not correspond to the actual argument's type. Following the description of $\lambda_{\mathcal{H}}$'s reduction semantics we seek the left-most inner-most potential redex in a closure. In order to reduce a closure, it is first decomposed. The closure might be a value and not contain any potential redex or it can be decomposed into a potential redex and a reduction context.

A decomposition function recursively searches for the left-most inner-most redex in a closure. In our implementation we define decomposition (`decompose`) as a big-step abstract machine with two state-transition functions, `decompose_closure` and `decompose_context`. The former traverses a given closure and accumulates the reduction context until it finds a value and the latter dispatches over the accumulated context to determine whether the given closure is a value or a potential redex. The function `decompose` starts by decomposing a closure within an empty context. For the full definition of the decomposition functions, see the accompanying code. The recomposition function `recompose` takes a context and a value to embed, peels off context layers and iteratively constructs the resulting closure.

Reduction-based normalization is based on a function that iterates a one-step reduction function until it yields a value (i.e., it reaches a fixed point). At each iteration the normalization function inspects its argument. If it is a potential redex within some context it will be contracted using the function `contract` and then be recomposed. If during contraction an error occurs, it must be reported. The terms we want to type-check via reduction-based normalization are from the host language (and described by the data type `term`) whereas intermediate values of reductions are within the larger hybrid language (i.e., they are of type `hterm`). So we should first embed "plain" terms into "hybrid" ones using the function `term_to_hterm`. The function `type_check` runs the reduction-based normalization function `normalize` and processes an obtained result.

reductions.sml

```
use "syntax.sml";
use "hsyntax.sml";
```

```
structure TypeCheck_Reduct =
struct

open HSyn TEnv

fun type_to_closure T_NUM
    = CLO_NUM
  | type_to_closure (v as T_ARR (t1, t2))
    = CLO_ARR_TYPE v

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)


datatype potential_redex = PR_NUM
                         | PR_LAM of string * typ * hterm * bindings
                         | PR_APP of typ * typ
                         | PR_ARR of typ * typ
                         | PR_IDE of string * bindings
                         | PR_PROP of hterm * hterm * bindings

datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string
(*  contract : potential_redex -> contractum_or_error  *)
fun contract PR_NUM
    = CONTRACTUM CLO_NUM
  | contract (PR_ARR (t1, t2))
    = CONTRACTUM (type_to_closure (T_ARR (t1, t2)))
  | contract (PR_IDE (x, bs))
    = (case TEnv.lookup (x, bs)
        of NONE => ERROR "undeclared␣identifier"
         | (SOME v) => CONTRACTUM (type_to_closure v))
  | contract (PR_LAM (x, t, e, bs))
    = CONTRACTUM (CLO_GND (H_TARR (t, e), TEnv.extend (x, t, bs)))
  | contract (PR_APP (T_ARR (t1, t2), v))
    = if t1 = v
      then CONTRACTUM (type_to_closure t2)
      else ERROR "parameter␣type␣mismatch"
  | contract (PR_PROP (t0, t1, bs))
    = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)))
  | contract (PR_APP (t1, t2))
    = ERROR "non-function␣application"

datatype type_or_decomposition = VAL of typ
                               | DEC of potential_redex * hctx

(*  decompose_closure : closure * hctx -> type_or_decomposition *)
fun decompose_closure (CLO_NUM, C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_ARR_TYPE v, C)
    = decompose_context (C, v)
  | decompose_closure (CLO_GND (H_LIT n, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = DEC (PR_LAM (x, t, e, bs), C)
  | decompose_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_GND (H_TNUM, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_TARR (t, e), bs), C)
```

```
    = decompose_closure (CLO_GND (e, bs),
                         CTX_ARR (t, C))
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_ARR (v, c), C)
    = decompose_closure (c, CTX_ARR (v, C))

(*  decompose_context : hctx * typ -> type_or_decomposition *)
and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)
  | decompose_context (CTX_ARR (v0, C), v1)
    = DEC (PR_ARR (v0, v1), C)

(*  decompose : closure -> type_or_decomposition *)
fun decompose c
  = decompose_closure (c, CTX_MT)

(*  recompose : hctx * closure -> closure *)
fun recompose (CTX_MT, c)
    = c
  | recompose (CTX_FUN (C, c1), c0)
    = recompose (C, CLO_APP (c0, c1))
  | recompose (CTX_ARG (v0, C), c1)
    = recompose (C, CLO_APP (type_to_closure v0, c1))
  | recompose (CTX_ARR (v0, C), c1)
    = recompose (C, CLO_ARR (v0, c1))

datatype result = RESULT of typ
                | WRONG of string

(*  iterate : type_or_decomposition -> result *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM c')
           => iterate (decompose (recompose (C, c')))
         | (ERROR s)
           => WRONG s)

(*  normalize : term -> result *)
fun normalize t
    = iterate (decompose (CLO_GND (term_to_hterm t, TEnv.empty)))

(*  type_check : term -> typ *)
fun type_check t
    = case normalize t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s
end
```

3 From Reduction-Based to Compositional Type Checker

This section provides the code of each stage of the transformation of the initial reduction-based evaluator for type checking to the traditional recursive descent. Each stage is described in the accompanying code by the file `reduction`N`.sml`, where N is the number of one of the following subsections.

The overview of the program metamorphoses is shown in Figure 1. The reduction-based normalization function is transformed to a family of reduction-free normalization functions, i.e.,

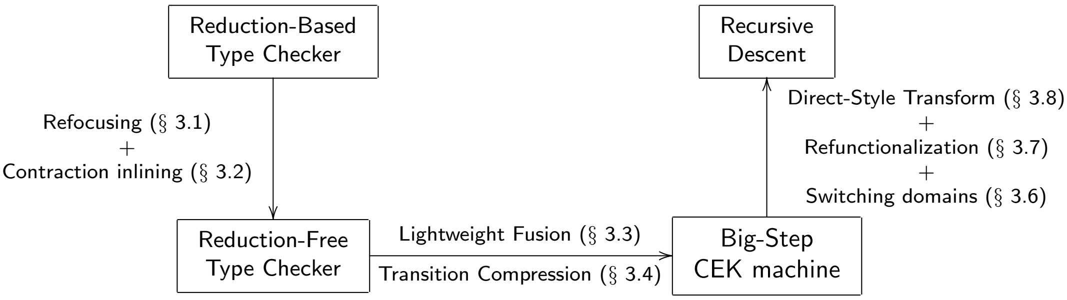

Figure 1: Inter-derivation

ones where no intermediate closure is ever constructed. In order to do so, we first refocus the reduction-based normalization function to obtain a small-step abstract machine implementing the iteration of the refocus function (Section 3.1). After inlining the contraction function (Section 3.2), we transform this small-step abstract machine into a big-step one applying a technique known as "lightweight fusion by fixed-point promotion" (Section 3.3). This machine exhibits a number of corridor transitions, which we compress (Section 3.4). We then flatten its configurations and rename its transition functions to something more intuitive (Section 3.5). We also switch domains of evaluator functions to factor out artifacts of the hybrid language (Section 3.6). The resulting abstract machine is in defunctionalized form, so we refunctionalize it (Section 3.7). The result is in continuation-passing style, so we transform it into direct style (Section 3.8). The final result is a traditional compositional type-checker.

3.1 Refocusing

The operation of decomposing and recomposing a term is usually referred as refocusing. By a simple observation, a refocusing function may be expressed via the `decompose_closure` function, mentioned in Section 2.

The new version of the type checker differs from the original one by the definition of the function `iterate1` using the function `refocus` instead the composition of `decompose` and `recompose`. The type checker is now reduction-free since no step-based reduction function is involved.

reductions1.sml

```
use "syntax.sml";
use "hsyntax.sml";

structure TypeCheck_Refocus =
struct

open HSyn TEnv

fun type_to_closure T_NUM
    = CLO_NUM
  | type_to_closure (v as T_ARR (t1, t2))
    = CLO_ARR_TYPE v

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)


datatype potential_redex = PR_NUM
                         | PR_LAM of string * typ * hterm * bindings
```

```
                         | PR_APP of typ * typ
                         | PR_ARR of typ * typ
                         | PR_IDE of string * bindings
                         | PR_PROP of hterm * hterm * bindings

datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string

fun contract PR_NUM
    = CONTRACTUM CLO_NUM
  | contract (PR_ARR (t1, t2))
    = CONTRACTUM (type_to_closure (T_ARR (t1, t2)))
  | contract (PR_IDE (x, bs))
    = (case TEnv.lookup (x, bs)
        of NONE
           => ERROR "undeclared␣identifier"
         | (SOME v) =>
           CONTRACTUM (type_to_closure v))
  | contract (PR_LAM (x, t, e, bs))
    = CONTRACTUM (CLO_GND (H_TARR (t, e),
                           TEnv.extend (x, t, bs)))
  | contract (PR_APP (T_ARR (t1, t2), v))
    = if t1 = v
      then CONTRACTUM (type_to_closure t2)
      else ERROR "parameter␣type␣mismatch"
  | contract (PR_PROP (t0, t1, bs))
    = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)))
  | contract (PR_APP (t1, t2))
    = ERROR "non-function␣application"

datatype type_or_decomposition = VAL of typ
                                | DEC of potential_redex * hctx

fun decompose_closure (CLO_NUM, C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_ARR_TYPE v, C)
    = decompose_context (C, v)
  | decompose_closure (CLO_GND (H_LIT n, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = DEC (PR_LAM (x, t, e, bs), C)
  | decompose_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_GND (H_TNUM, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_TARR (t, e), bs), C)
    = decompose_closure (CLO_GND (e, bs), CTX_ARR (t, C))
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_ARR (v, c), C)
    = decompose_closure (c, CTX_ARR (v, C))


and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)
  | decompose_context (CTX_ARR (v0, C), v1)
    = DEC (PR_ARR (v0, v1), C)

fun decompose c
  = decompose_closure (c, CTX_MT)

fun recompose (CTX_MT, c)
```

```
    = c
  | recompose (CTX_FUN (C, c1), c0)
    = recompose (C, CLO_APP (c0, c1))
  | recompose (CTX_ARG (v0, C), c1)
    = recompose (C, CLO_APP (type_to_closure v0, c1))
  | recompose (CTX_ARR (v0, C), c1)
    = recompose (C, CLO_ARR (v0, c1))

datatype result = RESULT of typ
                | WRONG of string

(*  refocus : closure * hctx -> type_or_decomposition  *)
fun refocus (c, C)
    = decompose_closure (c, C)

(*  iterate1 : type_or_decomposition -> result  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM c')
           => iterate1 (refocus (c', C))
         | (ERROR s)
           => WRONG s)

(*  normalize1 : term -> result  *)
fun normalize1 t
    = iterate1 (refocus (CLO_GND (term_to_hterm t,
                                  TEnv.empty), CTX_MT))

fun type_check t
    = case normalize1 t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s

end
```

3.2 Inlining the contraction function

We inline the function `contract` in the definition of `iterate1`. There are six cases in the definition of `contract`, so the `DEC` clause in the definition of `iterate1` is replaced by six `DEC` clauses. The resulting function is called `iterate2`.

reductions2.sml

```
use "syntax.sml";
use "hsyntax.sml";

(* Inline contraction function *)
structure TypeCheck_InlineContract =
struct

open HSyn TEnv

fun type_to_closure T_NUM
    = CLO_NUM
  | type_to_closure (v as T_ARR (t1, t2))
    = CLO_ARR_TYPE v

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
```

```
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)

datatype potential_redex = PR_NUM
                         | PR_LAM of string * typ * hterm * bindings
                         | PR_APP of typ * typ
                         | PR_ARR of typ * typ
                         | PR_IDE of string * bindings
                         | PR_PROP of hterm * hterm * bindings

datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string


datatype type_or_decomposition = VAL of typ
                               | DEC of potential_redex * hctx

fun decompose_closure (CLO_NUM, C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_ARR_TYPE v, C)
    = decompose_context (C, v)
  | decompose_closure (CLO_GND (H_LIT n, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = DEC (PR_LAM (x, t, e, bs), C)
  | decompose_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_GND (H_TNUM, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_TARR (t, e), bs), C)
    = decompose_closure (CLO_GND (e, bs), CTX_ARR (t, C))
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_ARR (v, c), C)
    = decompose_closure (c, CTX_ARR (v, C))


and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)
  | decompose_context (CTX_ARR (v0, C), v1)
    = DEC (PR_ARR (v0, v1), C)

datatype result = RESULT of typ
                | WRONG of string

fun refocus (c, C)
    = decompose_closure (c, C)

fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_NUM, C))
    = iterate2 (refocus (CLO_NUM, C))
  | iterate2 (DEC (PR_ARR (t1, t2), C))
    = iterate2 (refocus (type_to_closure
                         (T_ARR (t1, t2)), C))
  | iterate2 (DEC (PR_IDE (x, bs), C))
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared␣identifier"
         | (SOME v) =>
           iterate2 (refocus (type_to_closure v, C)))
  | iterate2 (DEC (PR_LAM (x, t, e, bs), C))
```

```
    = iterate2 (refocus
                  (CLO_GND (H_TARR (t, e),
                       TEnv.extend (x, t, bs)), C))
  | iterate2 (DEC (PR_APP (T_ARR (t1, t2), v), C))
    = if t1 = v
      then iterate2 (refocus (type_to_closure t2, C))
      else WRONG "parameter␣type␣mismatch"
  | iterate2 (DEC (PR_PROP (t0, t1, bs), C))
    = iterate2 (refocus (CLO_APP (CLO_GND (t0, bs),
                                  CLO_GND (t1, bs)), C))
  | iterate2 (DEC (PR_APP (t1, t2), C))
    = WRONG "non-function␣application"

fun normalize2 t
    = iterate2 (refocus (CLO_GND (term_to_hterm t, TEnv.empty), CTX_MT))

fun type_check t
    = case normalize2 t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s

end
```

3.3 Lightweight fusion: from small-step to big-step abstract machine

The next step is to *fuse* the definitions of `iterate2` and `refocus` from the previous section. The result of the fusion, called `iterate3`, is directly applied to the result of `decompose_closure` and `decompose_context`. The result is a big-step abstract machine consisting of three mutually tail-recursive state-transition functions:

- `refocus3_closure`, the composition of `iterate2` and `decompose_closure` and a clone of `decompose_closure`,
- `refocus3_context`, the composition of `iterate2` and `decompose_context`, which directly calls `iterate3` over the value of decomposition,
- `iterate3`, a clone of `iterate2` that calls the fused function `refocus3_closure`.

reductions3.sml

```
use "syntax.sml";
use "hsyntax.sml";

(* Lightweight Fusion *)
(* From small-step to big-step abstract machine *)
structure TypeCheck_Fusion =
struct

open HSyn TEnv

fun type_to_closure T_NUM
    = CLO_NUM
  | type_to_closure (v as T_ARR (t1, t2))
    = CLO_ARR_TYPE v

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)
```

```
datatype potential_redex = PR_NUM
                         | PR_LAM of string * typ * hterm * bindings
                         | PR_APP of typ * typ
                         | PR_ARR of typ * typ
                         | PR_IDE of string * bindings
                         | PR_PROP of hterm * hterm * bindings

datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string


datatype type_or_decomposition = VAL of typ
                               | DEC of potential_redex * hctx

datatype result = RESULT of typ
                | WRONG of string

(*  refocus3_closure : closure * hctx -> result *)
fun refocus3_closure (CLO_NUM, C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_ARR_TYPE v, C)
    = refocus3_context (C, v)
  | refocus3_closure (CLO_GND (H_LIT n, bs), C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_GND (H_IDE x, bs), C)
    = iterate3 (DEC (PR_IDE (x, bs), C))
  | refocus3_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = iterate3 (DEC (PR_LAM (x, t, e, bs), C))
  | refocus3_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = iterate3 (DEC (PR_PROP (t0, t1, bs), C))
  | refocus3_closure (CLO_GND (H_TNUM, bs), C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_GND (H_TARR (t, e), bs), C)
    = refocus3_closure (CLO_GND (e, bs),
                        CTX_ARR (t, C))
  | refocus3_closure (CLO_APP (c0, c1), C)
    = refocus3_closure (c0, CTX_FUN (C, c1))
  | refocus3_closure (CLO_ARR (v, c), C)
    = refocus3_closure (c, CTX_ARR (v, C))


(*  refocus3_context : hctx * typ -> result  *)
and refocus3_context (CTX_MT, v)
    = iterate3 (VAL v)
  | refocus3_context (CTX_FUN (C, c1), v0)
    = refocus3_closure (c1, CTX_ARG (v0, C))
  | refocus3_context (CTX_ARG (v0, C), v1)
    = iterate3 (DEC (PR_APP (v0, v1), C))
  | refocus3_context (CTX_ARR (v0, C), v1)
    = iterate3 (DEC (PR_ARR (v0, v1), C))

(*  iterate3 : type_or_decomposition -> result  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_NUM, C))
    = refocus3_closure (CLO_NUM, C)
  | iterate3 (DEC (PR_ARR (t1, t3), C))
    = refocus3_closure (type_to_closure
                 (T_ARR (t1, t3)), C)
  | iterate3 (DEC (PR_IDE (x, bs), C))
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared␣identifier"
         | (SOME v) =>
           refocus3_closure (type_to_closure v, C))
  | iterate3 (DEC (PR_LAM (x, t, e, bs), C))
    = refocus3_closure (CLO_GND (H_TARR (t, e),
```

```
                  TEnv.extend (x, t, bs)), C)
  | iterate3 (DEC (PR_APP (T_ARR (t1, t3), v), C))
    = if t1 = v
      then refocus3_closure ((type_to_closure t3), C)
      else WRONG "parameter␣type␣mismatch"
  | iterate3 (DEC (PR_PROP (t0, t1, bs), C))
    = refocus3_closure (CLO_APP (CLO_GND (t0, bs),
                                 CLO_GND (t1, bs)), C)
  | iterate3 (DEC (PR_APP (t1, t2), C))
    = WRONG "non-function␣application"


(*  normalize3 : term -> result  *)
fun normalize3 t
    = refocus3_closure (CLO_GND (term_to_hterm t,
                                 TEnv.empty), CTX_MT)

fun type_check t
    = case normalize3 t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s

end
```

3.4 Compressing corridor transitions

In the abstract machine from the previous section many transitions are *corridors*, i.e., they yield configurations for which there is a unique place for further consumption. In this section we *compress* these configurations. We copy the functions from the previous sections, changing their indices from 3 to 4. After this transformation *all* clauses of the function `refocus4_closure` for non-ground closures are now dead as well as the fact that all transition of `refocus4_closure` are now over ground closures, so we can flatten them by peeling off the “closure” part.

reductions4.sml

```
use "syntax.sml";
use "hsyntax.sml";

(* Compressing Corridor transitions *)
structure TypeCheck_Compress =
struct

open HSyn TEnv

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)


datatype result = RESULT of typ
                | WRONG of string

fun refocus4_closure (CLO_GND (H_LIT n, bs), C)
    = refocus4_context (C, T_NUM)
  | refocus4_closure (CLO_GND (H_IDE x, bs), C)
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared␣identifier"
         | (SOME v) =>
```

```
           refocus4_context (C, v))
  | refocus4_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = refocus4_closure (CLO_GND (H_TARR (t, e),
                                 TEnv.extend (x, t, bs)), C)
  | refocus4_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = refocus4_closure (CLO_GND (t0, bs), CTX_FUN (C, CLO_GND (t1, bs)))
  | refocus4_closure (CLO_GND (H_TNUM, bs), C)
    = refocus4_context (C, T_NUM)
  | refocus4_closure (CLO_GND (H_TARR (t, e), bs), C)
    = refocus4_closure (CLO_GND (e, bs), CTX_ARR (t, C))

and refocus4_context (CTX_MT, v)
    = RESULT v
  | refocus4_context (CTX_FUN (C, c1), v0)
    = refocus4_closure (c1, CTX_ARG (v0, C))
  | refocus4_context (CTX_ARG (v0, C), v1)
    = iterate4 (v0, v1, C)
  | refocus4_context (CTX_ARR (v0, C), v1)
    = refocus4_context (C, (T_ARR (v0, v1)))

and iterate4 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then refocus4_context (C, t4)
      else WRONG "parameter␣type␣mismatch"
  | iterate4 (t, v, C)
    = WRONG "non-function␣application"


fun normalize1 t
    = refocus4_closure (CLO_GND (term_to_hterm t, TEnv.empty), CTX_MT)

fun type_check t
    = case normalize1 t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s

end
```

3.5 Renaming transition functions and flattening configurations

The resulting simplified machine is a familiar ‘eval/apply/continue’ abstract machine operating over ground closures. For this section we rename `refocus4_closure` to `eval5`, `refocus4_context` to `continue5` and `iterate4` to `apply5`. We flatten the configuration of `refocus4_closure` as well as definitions of values and contexts.

reductions5.sml

```
use "syntax.sml";
use "hsyntax.sml";

(* Renaming transition functions and flattening configurations *)
structure TypeCheck_Renamed =
struct

open HSyn TEnv

fun term_to_hterm (IDE s)
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)
```

```
datatype context = CTX_MT
                 | CTX_FUN of context * hterm * bindings
                 | CTX_ARG of typ * context
                 | CTX_ARR of typ * context

datatype result = RESULT of typ
                | WRONG of string

(*  eval5 : hterm * (string * typ) list * context -> result  *)
fun eval5 (H_LIT n, gamma, C)
    = continue5 (C, T_NUM)
  | eval5 (H_IDE x, gamma, C)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => WRONG "undeclared␣identifier"
         | (SOME v) =>
           continue5 (C, v))
  | eval5 (H_LAM (x, t, e), gamma, C)
    = eval5 (H_TARR (t, e), TEnv.extend (x, t, gamma), C)
  | eval5 (H_APP (t0, t1), gamma, C)
    = eval5 (t0, gamma, CTX_FUN (C, t1, gamma))
  | eval5 (H_TNUM, gamma, C)
    = continue5 (C, T_NUM)
  | eval5 (H_TARR (t, e), gamma, C)
    = eval5 (e, gamma, CTX_ARR (t, C))

(*  continue5 : context * typ -> result  *)
and continue5 (CTX_MT, v)
    = RESULT v
  | continue5 (CTX_FUN (C, c1, gamma), v0)
    = eval5 (c1, gamma, CTX_ARG (v0, C))
  | continue5 (CTX_ARG (v0, C), v1)
    = apply5 (v0, v1, C)
  | continue5 (CTX_ARR (v0, C), v1)
    = continue5 (C, (T_ARR (v0, v1)))

(*  apply5 : typ * typ * context -> result  *)
and apply5 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then continue5 (C, t4)
      else WRONG "parameter␣type␣mismatch"
  | apply5 (t, v, C)
    = WRONG "non-function␣application"


(*  normalize5 : term -> result  *)
fun normalize5 t
    = eval5 (term_to_hterm t, TEnv.empty, CTX_MT)

(*  type_check : term -> typ  *)
fun type_check t
    = case normalize5 t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s

end
```

3.6 Removing hybrid artifacts and switching domains

The next simplification is to remove $\lambda_{\mathcal{H}}$-related artifacts from machine configurations. We copy functions from the previous section and perform some extra corridor transition compressions:

```
eval5 (H_LAM (x, t, e), gamma, C)
```

```
= (* by unfolding the definition of eval5 *)
eval5 (H_TARR (t, e), TEnv.extend (x, type_to_value t, gamma), C)
= (* by unfolding the definition of eval5 *)
eval5 (e, TEnv.extend (x, type_to_value t, gamma), CTX_ARR (type_to_value t, C))
```

As a result, there are no more clauses mentioning elements of the hybrid language such as `H_TNUM` (removed as an unused clause of `eval5`) and `H_TARR`. So now we can switch the domain of the `eval5`, `continue5` and `apply5` functions from `hterm` to `term`. The second observation is that algebraic data type `result` is in fact isomorphic to the data type `typ`, so we can switch the domain of values as well as follows:

$$\begin{array}{rcl}
\mathsf{RESULT\ (T\_NUM)} & \mapsto & \mathsf{T\_NUM} \\
\mathsf{RESULT\ (T\_ARR}\ (\tau_1, \tau_2)) & \mapsto & \mathsf{T\_ARR}\ (\tau_1, \tau_2) \\
\mathsf{WRONG}\ (s) & \mapsto & \mathsf{T\_ERROR}\ (s)
\end{array}$$

reductions6.sml

```
use "syntax.sml";
use "hsyntax.sml";

(*  Compressing hybrid syntax elements and switching domains  *)
structure TypeCheck_HybridCompress =
struct

open HSyn TEnv

datatype result = RESULT of typ
                | WRONG of string

datatype context = CTX_MT
                 | CTX_FUN of context * term * bindings
                 | CTX_ARG of typ * context
                 | CTX_ARR of typ * context
withtype bindings = typ gamma

(*  term * (string * typ) list * context -> typ  *)
fun eval6 (LIT n, gamma, C)
    = continue6 (C, T_NUM)
  | eval6 (IDE x, gamma, C)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => T_ERROR "undeclared␣identifier"
         | (SOME v) =>
           continue6 (C, v))
  | eval6 (LAM (x, t, e), gamma, C)
    = eval6 (e, TEnv.extend (x, t, gamma),
              CTX_ARR (t, C))
  | eval6 (APP (t0, t1), gamma, C)
    = eval6 (t0, gamma, CTX_FUN (C, t1, gamma))

(*  continue6 : context * typ -> typ  *)
and continue6 (CTX_MT, v)
    = v
  | continue6 (CTX_FUN (C, c1, gamma), v0)
    = eval6 (c1, gamma, CTX_ARG (v0, C))
  | continue6 (CTX_ARG (v0, C), v1)
    = apply6 (v0, v1, C)
  | continue6 (CTX_ARR (v0, C), v1)
    = continue6 (C, (T_ARR (v0, v1)))

(*  apply6 : typ * typ * context -> typ  *)
and apply6 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then continue6 (C, t4)
      else T_ERROR "parameter␣type␣mismatch"
  | apply6 (t, v, C)
```

```
    = T_ERROR "non-function␣application"

(*  term -> typ  *)
fun normalize6 t
    = eval6 (t, TEnv.empty, CTX_MT)

fun type_check t
    = normalize6 t

end
```

3.7 Refunctionalization

The abstract machine obtained in the previous section is in fact in defunctionalized form: the reduction contexts, together with `continue6`, are the first-order counterpart of continuations. To obtain the higher-order counterpart we use a technique known as *refunctionalization*. The resulting refunctionalized program is a compositional evaluation function in continuation-passing style.

`reductions7.sml`

```
use "syntax.sml";
use "hsyntax.sml";

(* Refunctionalization *)
structure TypeCheck_Refun =
struct

open Syn TEnv

(*  eval7 : term * (string * typ) list * (typ -> typ) -> typ  *)
fun eval7 (LIT n, gamma, k)
    = k T_NUM
  | eval7 (IDE x, gamma, k)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => T_ERROR "undeclared␣identifier"
         | (SOME v) =>
           k v)
  | eval7 (LAM (x, t, e), gamma, k)
    = eval7 (e, TEnv.extend (x, t, gamma),
          fn v => k (T_ARR (t, v)))

  | eval7 (APP (e0, e1), gamma, k)
    = eval7 (e0, gamma,
          fn t => case t
                   of T_ARR (t1, t2)
                      => eval7 (e1, gamma,
                             fn v1 =>
                                if t1 = v1
                                then k t2
                                else T_ERROR "parameter␣type␣mismatch")
                    | _ => T_ERROR "non-function␣application")

(*  normalize7 : term -> typ  *)
fun normalize7 t
    = eval7 (t, TEnv.empty, fn x => x)

fun type_check t
    = normalize7 t

end
```

3.8 Back to direct style

The refunctionalized definition from the previous section is in continuation-passing style: it has a functional accumulator and all of its calls are tail calls. To implement it in direct style in the presence of non-local returns in cases where typing error occurs, the library for undelimited continuations `SMLofNJ.Cont`, provided by Standard ML of New Jersey, is used.

reductions8.sml

```
use "syntax.sml";
use "hsyntax.sml";

(* Back to direct style *)
structure TypeCheck_Direct =
struct

open Syn TEnv

val callcc = SMLofNJ.Cont.callcc
val throw = SMLofNJ.Cont.throw

(*  normalize8 : term -> typ *)
fun normalize8 t
  = callcc (fn top =>
  let fun eval8 (LIT n, gamma)
        = T_NUM
     | eval8 (IDE x, gamma)
         = (case TEnv.lookup (x, gamma)
             of NONE
                => throw top (T_ERROR "undeclared␣identifier")
              | (SOME v) =>
                v)
     | eval8 (LAM (x, t, e), gamma)
       = T_ARR (t, eval8 (e, TEnv.extend (x, t, gamma)))
     | eval8 (APP (e0, e1), gamma)
       = let val t = eval8 (e0, gamma)
             val v1 = eval8 (e1, gamma)
         in (case t
              of T_ARR (t1, t2)
                 => if t1 = v1
                    then t2
                    else throw top (T_ERROR "parameter␣type␣mismatch")
               | _ => throw top (T_ERROR "non-function␣application"))
         end
  in eval8 (t, TEnv.empty)
  end)

(*  type_check : term -> typ *)
fun type_check t
    = normalize8 t

end
```

4 Conclusion

In this work we implemented a reduction semantics for type checking and a traditional recursive descent type checker as programs in SML. Through a series of behaviour-preserving program transformations we have shown that both these models are computationally equivalent and in fact just represent different ways to compute the same result. To the best of our knowledge, this is the first application of the study of the relation between reduction-free and reduction-based semantics to type systems. The result is a step towards reusing different computational models for type checking, whose equivalence is correct by construction.

References

[1] Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Information Processing Letters*, 112(1-2):13–20, January 2012.