From Type Checking by Recursive Descent to Type Checking with an Abstract Machine[*]

Ilya Sergey
DistriNet & IBBT, Dept. Computer Science
Katholieke Universiteit Leuven, Belgium
ilya.sergey@cs.kuleuven.be

Dave Clarke
DistriNet & IBBT, Dept. Computer Science
Katholieke Universiteit Leuven, Belgium
dave.clarke@cs.kuleuven.be

ABSTRACT

Modern type systems for programming languages usually incorporate additional information useful for program analysis, e.g., effects, control flow, non-interference, strictness etc. When designing a typing predicate for such systems, a form of logical derivation rules is normally taken. Despite the expressivity of this approach, the straightforward implementation of an appropriate type checker is usually inefficient in terms of stack consumption and further optimisations. This leads to a significant gap between an analysis and program implementing the analysis.

In this paper we demonstrate an application of techniques investigated by Danvy et al. to derive an abstract machine for typing from the traditional recursive descent approach. All used techniques are off-the-shelf and no appropriate correspondence theorems between an initial type system and the derived abstract machine needs to be proven: they are instead corollaries of the correctness of inter-derivation and of the initial specification. Whereas a recursive descent is something straightforward to implement based on declarative typing rules, the derived abstract machine exposes behaviour similar to Landin's SECD machine and gives a solid basis for further optimizations using abstract interpretation.

Categories and Subject Descriptors

D.1.1 [**Software**]: Programming techniques—*Applicative (Functional) Programming*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*; F.3.1 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms

Algorithms, Types, Theory

[*]This work was carried out while the first author was visiting the BRICS PhD School of Aarhus University in September 2010

Keywords

abstract machines, continuations, continuation-passing style (CPS), CPS transformation, defunctionalization, SECD machine, type systems, type checking

1. INTRODUCTION

Type systems in programming languages are a well-established way to ensure fundamental properties of programs. The principle "well-typed programs do not go wrong" introduced by Milner [19] and followed by "well-typed programs do not get stuck" by Wright and Felleisen [27] give an idea of some very basic of these properties, such a program execution progress. However, modern type systems are also targeted to infer more specific program properties such as possible computational effects, non-interference, control-flow information and strictness [21, 26, 28, 14]. These enhancements inevitably affect the implementation of a type inference algorithm, making it significantly harder to evaluate and to reason about. Therefore, in program analysis one should always distinguish between *an analysis* and *a program to implement the analysis*. The goal of this paper is to bridge this gap and establish a systematic transition from an expressive definition to an effective implementation.

1.1 Motivation

Traditionally, type systems are described in terms of logical derivation rules and can be implemented in the form of *recursive descent* as a part of production quality compilers. In this paper we consider type-checking procedure for simply typed lambda calculus (STLC) as an underlying algorithm for different implementations. Figure 1 describes well-known typing rules. The computational procedure for type checking using these rules is a recursive descent, where a given λ-term is recursively traversed, so its type is derived if no typing errors have been occurred.

Such an approach, however, is difficult to implement effectively in a presence of multiple computations involving inferred types and iterative typing fixed-point expressions. For example, in the system for abstract non-interference [28], types have denotational meaning

$$\text{[t-lam]}\ \frac{\Gamma[x:\tau_1] \vdash e : \tau_2}{\Gamma : \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2} \qquad \text{[t-var]}\ \frac{(x : \tau \in \Gamma)}{\Gamma \vdash x : \tau}$$

$$\text{[t-app]}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \text{[t-num]}\ \Gamma \vdash \mathit{number} : \mathsf{num}$$

Figure 1: Type system for the simply typed lambda calculus

$$\begin{array}{rcl}
\langle S,E,num:C\rangle & \Rightarrow_t & \langle \mathsf{num}:S,E,C\rangle \\
\langle S,E[x\Rightarrow\tau],x:C\rangle & \Rightarrow_t & \langle \tau:S,E[x\Rightarrow\tau],C\rangle \\
\langle S,E,(\lambda x:\tau.e):C\rangle & \Rightarrow_t & \langle \mathsf{nil},E\sqcup\{x\Rightarrow\tau\},e:Lam(\tau,S):C\rangle \\
\langle S,E,(e_1e_2):C\rangle & \Rightarrow_t & \langle S,E,e1:Fun(e_2):C\rangle \\
\langle \tau_2:S,E,Lam(\tau_1,S'):C\rangle & \Rightarrow_t & \langle (\tau_1\to\tau_2):S',E,C\rangle \\
\langle (\tau_1\to\tau_2):S,E,Fun(e_2):C\rangle & \Rightarrow_t & \langle (\tau_1\to\tau_2):S,E,e_2:Arg(\tau_1,\tau_2):C\rangle \\
\langle \tau_1:x:S,E,Arg(\tau_1,\tau_2):C\rangle & \Rightarrow_t & \langle \tau_2:S,E,C\rangle
\end{array}$$

Figure 2: A small-step transition system for type checking

associated with appropriate abstract domains. As a consequence, the type derivation rule for lambda-abstractions demands for exhaustive iteration through all elements of a possibly infinite abstract domain. Even more, the rule for a fixed-point expression computes a least-fixed point of the sub-derivation for its argument. To implement this iteration effectively and optimize it easily by switching domains, one should choose a more *operational* representation.

The first step towards the connection of type inference by recursive descent and type inference via abstract machines is due Hankin and Le Métayer [14]. They provide a description of an abstract machine-like formalism for implementing type checking and type inference systems. The described technique is in the spirit of Hannan and Miller [15] and yields in several stages an abstract machine based on the given type derivation rules. That machine strongly resembles Landin's SECD machine [18]. The only difference between the resulting machine and the original SECD machine is that the former has no "D" component since there is no "dump" in the corresponding evaluator, so we call the respective artifact *SEC machine* following the tradition to name machines after their control strings. A simplified version of the small-step machine, defined by its transition relation, is given in Figure 2. The following theorem has been proven for soundness and completeness of the derived machine:

THEOREM 1.1. [14] (Soundness and Completeness for $\Rightarrow_t$) $\Gamma\vdash e:\tau$ *iff* $\langle S,\Gamma,e:C\rangle\Rightarrow_t\langle\tau:S,\Gamma,C\rangle$.

One can see that the third component of the abstract machine (i.e., "C" for *control*) contains λ-terms as control elements, but also specific tokens, such as *Lam*, *Fun* and *Arg*, with extra bits of context information. Intuitively it is clear that these elements correspond somehow to combination of type constituents in derivation rules. However the question that remains open is what is a formal meaning of this correspondence?

The contribution of this paper is a *mechanical* inter-derivation of the two above mentioned type inference procedures via the program transformations used in Reynolds's functional correspondence between evaluators and big-step abstract machines [1, 23] and in Danvy et al.'s work on the systematic deconstruction of Landin's SECD machine [9]. The correspondence between a traditional type system and a SEC machine for type inference is provided by the construction and inter-derivation of their computational counterparts. The pleasant consequence is that no soundness and completeness theorems need to be proven: they are instead corollaries of the correctness of inter-derivation and of the initial specification [5].

1.2 Paper outline

The remainder of the paper is structured as follows. Section 2 gives an overview of our method, enumerating the techniques involved. Section 3 provides the implementation of type checking simply typed lambda calculus and describes initial setting for further functional transformations. Section 4 describes the set of program transformations corresponding to the construction of an abstract machine for type inference from the traditional type inference procedure in the form of a recursive descent. Section 5 starts a discussion and provides a brief survey of related work. Section 6 concludes.

2. METHOD OVERVIEW

A diagram with an overview of program metamorphoses is shown in Figure 3. In the following sections we show that SEC machine can be derived methodologically from the canonical compositional type checker by a sequence of meaning-preserving program transformations, such as the continuation-passing style transformation and defunctionalization [8, 11, 23].

We start by providing an implementation of a traditional type checker for the STLC in the form of a recursive descent as a starting point for further transformations (Section 3). We successively refactor it into a stack-threading *callee-save* evaluator, i.e., one that pushes its results on an explicit local stack, which is passed around as a parameter — the component "S" of a control string (Section 4.1). The obtained evaluator is in non-tail call form, so we transform it into continuation-passing style (Section 4.2) and then defunctionalize it (Section 4.3), which leads to the big-step stack-threading CEK machine. The type environment is still a part of some defunctionalized contexts, so we extract it as an explicit parameter of the evaluator, i.e., the component "E" of the control string (Section 4.4). We introduce an explicit control stack (the component "C" of the control string) in order to merge together several mutually recursive transition functions (Section 4.5), which yields a big-step SEC machine. Finally, we rework the big-step machine into a small-step one by extracting an *iteration* function (Section 4.6). The final machine is Landin's SECD machine lacking the "D" component of its control string, since no explicit control flow management with *dumps* is needed for type-checking.

Standard ML (SML) [20] is used as a metalanguage for the implementation and transformations. SML is a statically-typed, call-by-value language with computational effects. For the sake of brevity we omit some of the program artifacts, keeping only essential parts to demonstrate the corresponding program transformation.[1] At each transformation stage the trailing index of all involved functions is incremented.

3. INITIAL SETTING

This section provides the initial implementation of a type checking procedure for the simply typed lambda calculus, which will be used for further transformations in Section 4.

3.1 Terms and types

The abstract syntax of simply typed lambda calculus includes integer literals, identifiers, lambda-abstractions and applications. Types are either numeric types or arrow types. Special value `T_ERROR` s

[1]The accompanying code is available from
`http://people.cs.kuleuven.be/ilya.sergey/types-sec.zip`

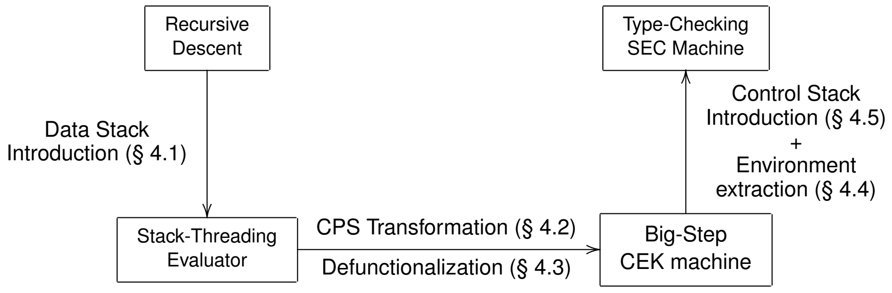
Recursive Descent
Data Stack Introduction (§ 4.1)
Stack-Threading Evaluator
CPS Transformation (§ 4.2)
Defunctionalization (§ 4.3)
Big-Step CEK machine
Control Stack Introduction (§ 4.5) + Environment extraction (§ 4.4)
Type-Checking SEC Machine

Figure 3: Inter-derivation

is used for typing errors and cannot be a substituent of any other type. We implement terms and types with the following SML data types:

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

datatype typ = T_NUM
             | T_ARR of typ * typ
             | T_ERROR of string
```

3.2 Type checking procedure

Typing environments `TEnv` represent bindings of identifiers to types. They carry typing assumptions about free variables in λ-terms. The value `empty` corresponds to an empty environment, `extend` extends an environment with a new binding of a variable into a type and, finally, the function `lookup` extracts the typing assumption, associated with a particular variable. A lookup may fail, which is reflected by its return type `'a option`[2].

```
signature TEnv =
sig
  type 'a gamma
  val empty : (string * 'a) gamma
  val extend : string * 'a * (string * 'a) gamma ->
               (string * 'a) gamma
  val lookup : string * (string * 'a) gamma -> 'a option
end
```

The canonical procedure for type checking [22, pages 113-116] is implemented as a recursive descent.

```
exception TYPING_ERROR of string

(*  check0 : term * typ gamma -> typ  *)
fun check0 (LIT n, gamma)
    = T_NUM
  | check0 (IDE x, gamma)
    = (case TEnv.lookup(x, gamma) of (SOME t) => t)
  | check0 (LAM (x, arg_type, body), gamma)
    = let val body_type = check0 (body,
          (TEnv.extend (x, arg_type, gamma)))
      in T_ARR (arg_type, body_type)
      end
  | check0 (APP (e1, e2), gamma)
    = let val T_ARR (t1, t2) = check0 (e1, gamma)
          val arg_type = check0 (e2, gamma)
      in if arg_type = t1
         then t2
         else raise (TYPING_ERROR "type␣mismatch")
      end

(*  type_check : term -> typ  *)
fun type_check t = check0 (t, TEnv.empty)
```

[2]In order to keep to the uniform approach for different semantics for type inference [17, 25], we leave environments parametrized by the type parameter `'a`, which is instantiated with `typ` in this case.

3.3 Representation of typing errors

Three kinds of typing errors might occur during type checking:

- ***Undefined identifier in an environment***, corresponds to the `MatchError` exception of SML raised in the second clause of `check0` function.
- ***Non-arrow type in a function position***, is represented by a `MatchError` raised at the top-level of `check0` function
- ***A type mismatch between a function parameter type and an argument type***, a `TYPING_ERROR` exception is raised at the last clause of `check0` function.

Moreover, there are at least three different ways to represent typing errors in practice and propagate the information about them to the client of the type checker.

1. *"Bubbling up"*: for results of recursive calls, the type-checking procedure checks explicitly whether the result is a type error. If it is, this information is returned immediately to an "upper level".
2. *Using exceptions*: when a type error occurs, the necessary information for a client of the type checker can be put into an exception, which is immediately raised. This approach is employed in the described implementation of the function `check0`.
3. *Continuation dropping*: if the type-checker is in continuation-passing style, one can interrupt the current control flow in case of a typing error. Then an error value will be returned instead of applying the continuation to the result [7].

In the following series of transformations we will be switching between the second and the third approaches.

4. FROM RECURSIVE DESCENT TO SEC MACHINE

In this section we describe a systematic approach to the construction of an abstract machine for type inference from a traditional type inference procedure in the form of recursive descent. The approach takes advantage of Reynolds's functional correspondence between different ways to represent semantic artifacts [23] and more recent work by Danvy et al. on the deconstruction of Landin's SECD machine [9].

4.1 Extracting a result stack

In the canonical implementation of a type checker the results of nested calls of the `check0` function are allocated on local stack frames of callees. We represent this model explicitly by introducing local result stacks and passing them around as an explicit parameter of `check1` function. A data stack, which is the "S" component of a control string for the machine presented in Section 1, stores intermediate values after they have been computed but before they are used. Computing an expression leaves its value on top of the data stack. Applications expect to find their argument and a function on top of this data stack. In case of nested calls the immutable part of the stack is saved by a callee, whereas, a caller is invoked with a reduced or fresh stack. This kind of evaluator is classified as a *callee-save*, explicit stack-threading one according to Danvy and Millikin [9, Appendix D].

The implementation of the function `type_check` is changed correspondingly to take the head of the result list as the result of a computation.

```
(*  check1 : term * typ list * typ gamma -> typ list  *)
fun check1 (LIT n, s, e)
    = T_NUM :: s
  | check1 (IDE x, s, e)
    = (case TEnv.lookup(x, e) of (SOME t) => t :: s)
  | check1 (LAM (x, arg_type, body), s, e)
    = let val (body_type :: _) =
              check1 (body, nil,
                      (TEnv.extend (x, arg_type, e)))
      in T_ARR (arg_type, body_type) :: s
      end
  | check1 (APP (e1, e2), s, e)
    = let val s0 as (T_ARR (t1, t2) :: _) =
              check1 (e1, nil, e)
            val (arg_type :: x :: _) = check1 (e2, s0, e)
      in if arg_type = t1
         then t2 :: s
         else raise (TYPING_ERROR "type␣mismatch")
      end

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) =  check1 (t, nil, TEnv.empty)
    in v end
```

4.2 CPS transformation

The function `check1` from the previous section is transformed into continuation-passing style (CPS). This is done in three steps, as described in Danvy's report [4]. Briefly, each intermediate result of a computation is extracted into a new local variable, their computations are sequentialized and a new formal parameter, namely, a continuation is introduced. Thus the intermediate results are named by the formal parameters of each of the lambda-abstractions that define the continuation.

```
(*  check2 : term * typ list * typ gamma *
             (typ list -> typ list) -> typ list  *)
fun check2 (LIT n, s, e, k) = k (T_NUM :: s)
  | check2 (IDE x, s, e, k)
    = k (case TEnv.lookup(x, e) of (SOME t) => t :: s)
  | check2 (LAM (x, arg_type, body), s, e, k)
    = check2 (body, nil, (TEnv.extend (x, arg_type, e)),
           fn (body_type :: s0) =>
              k (T_ARR (arg_type, body_type) :: s))
  | check2 (APP (e1, e2), s, e, k)
    = check2 (e1, nil, e,
         fn (s0 as (T_ARR (t1, t2) :: _)) =>
           check2 (e2, s0, e,
             fn (arg_type :: x :: _) =>
                if arg_type = t1
                then k (t2 :: s)
                else (T_ERROR "type␣mismatch") :: nil))

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) =  check2 (t, nil,
                                TEnv.empty, fn x => x)
    in v end
```

Since we CPS-transformed our program, we may replace exception raising by non-local returns, as it is done now in the last clause of `check2` function: a `T_ERROR` is returned directly if a typing error occurs. This small transformation corresponds to the switching between the second and third methods of typing error representation described in Section 3. The resulting procedure, considered as an interpreter of λ-terms, is a traditional continuation-passing one.

4.3 Defunctionalization

The next step is to defunctionalize the continuations in the implementation of the type checker from Section 4.2. The function space of the considered program is inhabited by the four function values that arise from considering four function abstractions from the definitions of functions `check2` and `type_check`: one initial continuation in `type_check` and three more in two last clauses of `check2`. We therefore partition the function space into four summands and represent it as the following first-order data type:

```
datatype cont = CONT_MT
              | CONT_LAM of typ * cont *  typ list
              | CONT_FUN of cont * term * typ gamma
              | CONT_ARG of typ * typ * cont
```

Those defunctionalized continuations represent first-order *evaluation contexts* of type computations on top of the abstract syntax of the calculus. Contexts are produced at places of former lambda-abstractions (the initial call of the function `type_check` and third and forth clauses of the function `check2`) and consumed by a "dispatcher"-like function `continue3`.

```
(*  check3 : term * typ list * typ gamma * cont ->
             typ list *)
fun check3 (LIT n, s, e, C)
    = continue3 (C, (T_NUM :: s))
  | check3 (IDE x, s, e, C)
    = continue3 (C, case TEnv.lookup(x, e)
                     of (SOME t) => t :: s)
  | check3 (LAM (x, arg_type, body), s, e, C)
    = check3 (body, nil, (TEnv.extend (x, arg_type, e)),
              CONT_LAM (arg_type, C, s))
  | check3 (APP (e1, e2), s, e, C)
    = check3 (e1, s, e, CONT_FUN (C, e2, e))

(*  continue3 : cont * typ list -> typ list  *)
and continue3 (CONT_MT, s)
    = s
  | continue3 (CONT_LAM (arg_type, C, s),
               (body_type :: s0))
    = continue3 (C, T_ARR (arg_type, body_type) :: s)
  | continue3 (CONT_FUN (C, e2, e),
               s0 as (T_ARR (t1, t2) :: _))
    = check3 (e2, s0, e, CONT_ARG (t1, t2, C))
  | continue3 (CONT_ARG (t1, t2, C), (arg_type :: x :: s1))
    = if arg_type = t1
      then continue3 (C, t2 :: s1)
      else (T_ERROR "type␣mismatch") :: nil

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) = check3 (t, nil,
                               TEnv.empty, CONT_MT)
    in v end
```

The resulting machine is an analogue of the well-known environment-based CEK machine with an explicit component `s` for the result stack [13]. Each tail call implements a state transition of the machine.

4.4 Extracting an environment to a parameter

One can notice that a type environment is part of the data type of evaluation contexts. We massage the type checking machine by extracting an environment to a separate explicit parameter of the function continue4. It will correspond to the component "E" in the control string of the final abstract machine. Now the constructor CONT_FUN, which is consumed by continue4, does not contain an environment as a parameter. We also rearrange parameters of the data type cont to give it a list-like shape. The data type of contexts is now as follows:

```
datatype cont = CONT_MT
              | CONT_LAM of typ * typ list * cont
              | CONT_FUN of term * cont
              | CONT_ARG of typ * typ * cont
```

A next natural step is to take advantage of the list-like structure of contexts represented by cont.

4.5 Adding an explicit control stack

In this section we introduce the last component of the control string of the abstract machine, namely, the *control stack* "C". The defunctionalized contexts from the Section 4.4 expose a stack-like structure with CONT_MT as the "empty" element. The structure can be refactored into a stack of control tokens, corresponding to particular summands of cont. To unify the structure of states we also introduce one more extra control token for terms. Control stack tokens are represented by the following data structure:

```
datatype control_element = C_ARG of typ * typ
                         | C_FUN of term
                         | C_LAM of typ * typ list
                         | C_TERM of term
```

Former CONT_MT element corresponds now to an empty control stack. Since the domain of control elements is now "lifted" to control_element, we may safely merge continue4 and check4 functions to get the unified function check5.

```
(*  check5 : typ list * typ gamma * control_element list
             -> typ list  *)
fun check5 (s, e, C_TERM (LIT n) :: C)
    = check5 (T_NUM :: s, e, C)
  | check5 (s, e, C_TERM (IDE x) :: C)
    = check5 (case TEnv.lookup(x, e)
               of (SOME t) => t :: s, e, C)
  | check5 (s, e, C_TERM (LAM (x, arg_type, body)) :: C)
    = check5 (nil, TEnv.extend (x, arg_type, e),
              C_TERM body :: C_LAM (arg_type, s) :: C)
  | check5 (s, e, C_TERM (APP (e1, e2)) :: C)
    = check5 (s, e, C_TERM e1 :: C_FUN e2 :: C)
  | check5 ((body_type :: s0), e,
            C_LAM (arg_type, s) :: C)
    = check5 (T_ARR (arg_type, body_type) :: s, e, C)
  | check5 (s0 as (T_ARR (t1, t2) :: _), e, C_FUN e2 :: C)
    = check5 (s0, e, C_TERM e2 :: C_ARG (t1, t2) :: C)
  | check5 (v2 :: x :: s1, e,
            C_ARG (arg_type, result_type) :: C)
    = if v2 = arg_type
      then check5 (result_type :: s1, e, C)
      else T_ERROR "parameter␣type␣mismatch" :: nil
  | check5 (s, e, nil)
    = s

(*  type_check : term -> typ  *)
and type_check t
  = let val (v :: s) =  check5 (nil, TEnv.empty,
                                C_TERM t :: nil)
    in v end
```

The resulting interpreter is a big-step SEC machine where each tail call of check5 corresponds to a transition. Now we are going to turn it into a small-step machine by introducing an explicit driver-loop function.

4.6 From a big-step to a small-step SEC machine

Since the big-step SEC machine from Section 4.5 has only one type of control string, it is straightforward to transform it into a small-step machine by introducing a dedicated driver-loop function iterate6:

```
type state = typ list * typ gamma * control_element list

(*  step6 : state -> state  *)
fun step6 (s, e, C_TERM (LIT n) :: C)
    = (T_NUM :: s, e, C)
  | step6 (s, e, C_TERM (IDE x) :: C)
    = (case TEnv.lookup(x, e) of (SOME t) => t :: s, e, C)
  | step6 (s, e, C_TERM (LAM (x, arg_type, body)) :: C)
    = (nil, TEnv.extend (x, arg_type, e),
       C_TERM body :: C_LAM (arg_type, s) :: C)
  | step6 (s, e, C_TERM (APP (e1, e2)) :: C)
    = (s, e, C_TERM e1 :: C_FUN e2 :: C)
  | step6 ((body_type :: s0), e, C_LAM (arg_type, s) :: C)
    = (T_ARR (arg_type, body_type) :: s, e, C)
  | step6 (s0 as (T_ARR (t1, t2) :: _), e, C_FUN e2 :: C)
    = (s0, e, C_TERM e2 :: C_ARG (t1, t2) :: C)
  | step6 (v2 :: x :: s1, e,
           C_ARG (arg_type, result_type) :: C)
    = if v2 = arg_type
      then (result_type :: s1, e, C)
      else raise (TYPING_ERROR "type␣mismatch")

(*  iterate6 : state -> typ  *)
fun iterate6 (v :: s, _, nil)
    = v
  | iterate6 state
    = iterate6 (step6 state)

(*  type_check : term -> typ  *)
fun type_check term
  = iterate6  (nil, TEnv.empty, C_TERM term :: nil)
```

At each step of the execution the machine performs a transition to a new state and the function iterate6 checks the termination condition. This last transition completes our chain of transformations. The transition function of the described small-step SEC machine corresponds directly to the set of transition rules, given in Figure 2. Arrow types are consumed implicitly in the last transition rule being popped from the result stack S with no additional check. However, the necessary check of type correspondence is performed thanks to the control element $Arg(\tau_1, \tau_2)$.

5. DISCUSSION

The approach described in this paper allows one to derive mechanically abstract machine operational representation for type derivations from their computational counterparts, implemented in the form of recursive descent. However all presented artifacts are hand-crafted, the implementation of the *automatic* transformation is to be addressed in the future work. All described transitions, except environment extraction as we described it, are well-known for implementors of interpreters for functional programming languages. In general, the present technique scales for implementation of many static analyses defined compositionally in the form of derivation rules: all one needs to do is to provide a straightforward initial implementation of the appropriate recursive descent. This section discusses properties of the presented correspondence, its applications and related work.

5.1 Applications

The transition system described in Figure 2 exhibits some generic elements, which can be adjusted according to the specific procedure of computations involving types. As it has been shown through Sections 4.2—4.5, the control stack elements Lam, Fun and Arg are derived from defunctionalized continuations. They trigger system-specific computations involving combinations of previously obtained types, stored in the result stack *S*.

Hankin and Le Métayer [14] in their work on lazy types derive an abstract machine similar to the one we have constructed in this paper. The type system they consider is augmented with Jensen's strictness logic [16]. As a consequence, a computational counterpart for the typing for lambda-abstractions involves iterating through multiple *abstract values* of the formal parameter's type, that leads to the exponential complexity of the derivation algorithm. In our transition system this possible pitfall would correspond to the computation of the third and fifth transition rules in Figure 2. The abstract machine-like representation allows one to coarsen the result of a type derivation by choosing different abstract domains to iterate through when the control element *Lam* is processed.

The similar idea is applicable to a more recent work on a type system for security and abstract non-interference by Zanardini [28]. Non-interference refers to the possibility that two computations can be distinguished by observing some public parts of data. Types in the described system have denotational meaning and are defined in terms of abstract value domains and identify properties which domains cannot distinguish. Parity or the sign of an integer, are the simplest examples of such abstract properties. The appropriate type system is encoded originally in terms of derivation rules, which involve iterations through possibly infinite semantic domains. From the abstract machine point of view, such an iteration would be triggered by control stack elements. It does not change the nature of type to be computed but makes it more precise depending on chosen semantic domains. Thus, an abstract machine-like representation would give an effective way to control the precision of the type-based analysis just by redefining the meaning of appropriate control stack elements.

5.2 Related work

The functional correspondence between different semantics artifacts has been recently applied to various tasks. Ager et al. [2] investigate a correspondence between semantics described in terms of monadic evaluators and languages with computational effects. They show that a calculus for tail-recursive stack inspection corresponds to a lifted state monad. This correspondence allows one to combine it with other monads and obtain abstract machines with both tail-recursive stack inspection and other computational effects. The similar technique applied to the standard call-by-need reduction for the λ-calculus yields a reduction-free stateless abstract machine and a heapless natural semantics for call-by-need evaluation [10]. Danvy and Zerny [12] present a purely syntactic theory of graph reduction for the canonical combinators S, K, and I, where graph vertices are represented with evaluation contexts and let expressions. This syntactic theory is expressed as a reduction semantics. Through the series of functional transformations, the authors derive a store-based abstract machine whose architecture coincides with that of Turner's original reduction machine.

Reduction semantics for type checking, proposed initially by Kuan et al. [17], is another operational view on type inference algorithms. Defined as a set of term-reduction rules, such a term-rewriting system gives an operational view on the semantics of type checking, which is useful for debugging complex type systems, since the developer can trace each step of the type computation. Dealing with this term-rewriting system requires one to show explicitly that an underlying type inference algorithm is equivalent to the traditional system described as a set of derivation rules. For this purpose, appropriate completeness and soundness theorems need to be proven. As it was shown recently by Sergey and Clarke [25], a correspondence between a traditional type system and a corresponding reduction-based semantics for type inference can be provided by the construction and inter-derivation of their computational counterparts in the spirit of the current work and recently described systematic functional transformations [6].

Anton and Thiemann [3] took reduction semantics for different implementations of coroutines from the literature and obtained equivalent definitional interpreters by applying the same sequence of transformations we used. The obtained operational semantics is transformed further into a denotational implementation that provides a necessary basis to construct a sound type system.

5.3 Future work

In the further research we are going to address scalability issues of the described approach. In the presence of various pluggable type systems one may want to augment the typing rules or add new ones, so the resulting abstract machine will change as well. The natural question is how to reflect these changes incrementally without going again through the whole chain of described transformations. We also leave a comparison of different implementations of a type checker with respect to performance to the future work.

The relation between attribute grammar approaches and the described transformations is another interesting topic of discussion. Since attributes are functions from AST nodes to attribute values, type checking can be represented as a computation of such attribute values. However, the described approach deals with an *eager* semantics of type checking whereas practical attribute grammars perform the value computation *lazily*. The possible way to unify these two approaches is to derive a call-by-need semantics for type checking.

6. CONCLUSION

In program analysis, one always has to distinguish between an analysis as its formal definition and a program that implements it. This paper proposes a methodology to bridge this gap by using the inter-derivational method due Reynolds [23] and Danvy et al. [6]. As an example, two implementation of a traditional type checking algorithm are considered: one in the form of recursive descent and another in the form of Landin's SECD machine. The correspondence between these two models is provided by the construction and inter-derivation of their computational counterparts. Through a series of behaviour-preserving program transformations we have shown that both models are computationally equivalent. Starting from one particular traversal strategy, a family of algorithms is derived. All of them implement this traversal strategy, but exhibit different computational properties. The result is a step towards reusing different computational models for compositional program analyses, such that the equivalence of the models is correct by construction.

Acknowledgements

We wish to acknowledge the valuable comments of Olivier Danvy on early drafts of the paper. We also thank the anonymous reviewers for their constructive feedback.

7. REFERENCES

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, PPDP '03, pages 8–19, New York, NY, USA, 2003. ACM.

[2] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

[3] K. Anton and P. Thiemann. Deriving Type Systems and Implementations for Coroutines. In *APLAS'10: Proceedings of the 8th Asian Symposium on Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, Shanghai, China, 2010.

[4] O. Danvy. Three Steps for the CPS Transformation. Technical Report CIS-92-2, Kansas State University, 1992.

[5] O. Danvy. An Analytical Approach to Program as Data Objects. DSc Thesis, Department of Computer Science, Aarhus University, October 2006.

[6] O. Danvy. From Reduction-Based to Reduction-Free Normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.

[7] O. Danvy and A. Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[8] O. Danvy and A. Filinski. Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[9] O. Danvy and K. Millikin. A Rational Deconstruction of Landin's SECD Machine with the J Operator. *Logical Methods in Computer Science*, 4(4), November 2008.

[10] O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized Interpreters for Call-by-Need Evaluation. In M. Blume and G. Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in LNCS, pages 240–256, Sendai, Japan, Apr. 2010. Springer.

[11] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 162–174, New York, NY, USA, 2001. ACM.

[12] O. Danvy and I. Zerny. Three syntactic theories for combinatory graph reduction. In M. Alpuente, editor, *LOPSTR'10 - 20th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 10-14 of *RISC-Linz Report Series*, pages 3–32, 2010.

[13] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, August 1986.

[14] C. Hankin and D. Le Métayer. Deriving Algorithms from Type Inference Systems: Application to Strictness Analysis. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 202–212, New York, NY, USA, 1994. ACM.

[15] J. Hannan and D. Miller. From Operational Semantics to Abstract Machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[16] T. P. Jensen. Strictness Analysis in Logical Form. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 352–366, London, UK, 1991. Springer-Verlag.

[17] G. Kuan, D. MacQueen, and R. B. Findler. A Rewriting Semantics for Type Inference. In *ESOP'07: Proceedings of the 16th European conference on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 426–440, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[20] R. Milner, M. Tofte, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[21] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, corrected edition, October 1999.

[22] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[23] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [24].

[24] J. C. Reynolds. Definitional interpreters revisited. 11(4):355–361, 1998.

[25] I. Sergey and D. Clarke. A Correspondence between Type Checking via Reduction and Type Checking via Evaluation. Katholieke Universiteit Leuven, December 2010.

[26] C. Skalka, S. Smith, and D. Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[27] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, November 1994.

[28] D. Zanardini. Higher-order abstract non-interference. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 417–432. Springer Berlin / Heidelberg, 2005.