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

Ilya Sergey[*,a,1], Dave Clarke[a]

[a] *DistriNet & IBBT, Dept. Computer Science, Katholieke Universiteit Leuven*
Celestijnenlaan 200a, bus 2402, B-3001 Leuven-Heverlee, Belgium

Abstract

We describe a derivational approach to proving the equivalence of different representations of a type system. Different ways of representing type assignments are convenient for particular applications such as reasoning or implementation, but some kind of correspondence between them should be proven. In this paper we address two such semantics for type checking: one, due to Kuan et al., in the form of a term rewriting system and the other in the form of a traditional set of derivation rules. By employing a set of techniques investigated by Danvy et al., we mechanically derive the correspondence between a reduction-based semantics for type-checking and a traditional one in the form of derivation rules, implemented as a recursive descent. The correspondence is established through a series of semantics-preserving functional program transformations.

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

1. Introduction

A well-designed type system makes a tradeoff between the expressiveness of its definition and the effectiveness of its implementation. Traditionally, type systems are described as collections of logical inference rules that are convenient to reason about. Computationally, such a model, implemented straightforwardly in a functional programming language, corresponds to a recursive descent over the inductively-defined language syntax. However, other more algorithmic representations allow one to reason about computational aspects of a type inference procedure. As an example of such a system, we consider a reduction semantics for type checking, proposed initially by Kuan et al. [14]. 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 such a term-rewriting system requires that one explicitly shows that the underlying type inference algorithm is equivalent to the traditional system described as a set of derivation rules. For this purpose, appropriate soundness and completeness theorems need to be proven [12].

In this paper a correspondence between a traditional type system and a corresponding reduction-based semantics for type inference is provided by the construction and inter-derivation of their computational counterparts. Thus no soundness and completeness theorems need to be proven: they are instead corollaries of the correctness of inter-derivation and of the initial specification. Starting from the implementation of a reduction-based semantics, we employ a set of techniques, investigated by Danvy et al. [1, 4, 6, 7, 8, 9], to eventually obtain a traditional recursive descent for type-checking via a series of semantics-preserving functional-program transformations. The transformations we use are off-the-shelf [5] and we invite an interested reader to take a look on the overview of the available techniques with references to the corresponding correctness proofs [4].

[*] Corresponding author
Email address: ilya.sergey@cs.kuleuven.be (Ilya Sergey)

[1] This work was carried out in September 2010 while the first author was visiting the Department of Computer Science at Aarhus University.

$$
\begin{array}{llll}
\text{SLC\_H} & e & ::= & n \mid x \mid \lambda x : \tau.e \mid e\ e \mid \tau \to e \mid \mathsf{num} \\
\text{CTX} & T & ::= & T\ e \mid \tau\ T \mid \tau \to T \mid [\,] \\
\text{TYPE} & \tau & ::= & \mathsf{num} \mid \tau \to \tau \\
 & n & ::= & \textit{number}
\end{array}
$$

Hybrid language and type-checking contexts

$$
\begin{array}{rcll}
T[n] & \mapsto_t & T[\mathsf{num}] & \text{[tc-const]} \\
T[\lambda x : \tau.e] & \mapsto_t & T[\tau \to \{\tau/x\}\ e] & \text{[tc-lam]} \\
T[(\tau_1 \to \tau_2)\ \tau_1] & \mapsto_t & T[\tau_2] & \text{[tc-}\tau\beta\text{]}
\end{array}
$$

Type-checking reduction rules

Figure 1: Reduction semantics of $\lambda_{\mathcal{H}}$

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

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \text{[t-app]} \qquad \Gamma \vdash \textit{number} : \mathsf{num}\ \ \text{[t-num]}
$$

Figure 2: Type system for the simply typed lambda calculus

1.1. Starting point: a hybrid language for type checking

We consider a reduction system for type checking the simply typed lambda calculus. The system was originally proposed by Kuan et al. [14] and is presented as a case study in the scope of PLT Redex framework [13]. The approach scales to Curry-Hindley type inference and Hindley-Milner let-polymorphism. The techniques presented in the current paper can be adjusted for these cases by adding unification variables, so for the sake of brevity we examine only the simplest model. The hybrid language $\lambda_{\mathcal{H}}$ and its semantics are described in Figure 1. The reduction system introduces a type-checking context T that induces a left-most, inner-most order of reduction. Variable occurrences are replaced by their types at the moment a λ-abstraction is reduced according to rule [tc-lam]. Rule [tc-lam] also introduces the arrow type constructor. Finally, rule [tc-$\tau\beta$] syntactically matches the function parameter type against an argument type.

The classical way to represent type checking is via a collection of logical derivation rules. Such rules for the simply typed lambda calculus are given in Figure 2. According to Kuan et al., a complete type reduction sequence is one that reduces to a type, which corresponds to a well-typed term. The following theorem states that a complete type reduction sequence corresponds to a complete type derivation proof tree for a well-typed term in the host language and vice versa.

Theorem 1.1. [14] (Soundness and Completeness for $\mapsto_t$) *For any e and τ, $\emptyset \vdash e : \tau$ iff $e \mapsto_t^* \tau$*

The question we address in this paper is whether a natural correspondence between these semantics exists which avoids the need for the soundness and completeness theorems. The answer to this question is positive and below we show how to derive a traditional type-checker mechanically from the given rewriting system. The contribution of the paper is a demonstation of how the application of well-studied program derivations to type checkers is helpful to show an equivalence between them.

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 describes an implementation of the hybrid language and its reduction semantics in Standard ML. Section 4 describes a set of program transformations corresponding to the transition from the reduction-based semantics for type inference to a traditional recursive descent. Section 5 provides a brief survey of related work. Section 6 concludes.

2. Method overview

The overview of the program metamorphoses is shown in Figure 3. We start by providing the implementation of a hybrid language for the simply typed lambda calculus, a notion of closures in it and

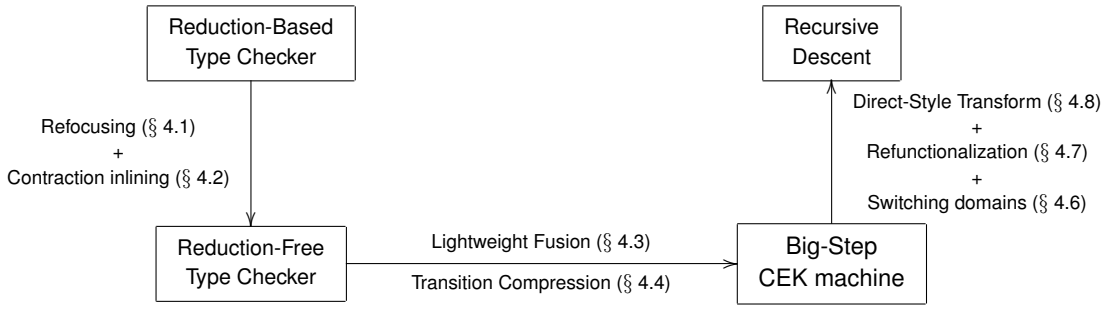

Figure 3: Inter-derivation

a corresponding reduction semantics via contraction as a starting point for further transformations (Section 3). The reduction-based normalization function is transformed to a family of reduction-free normalization functions, i.e., 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 4.1). After inlining the contraction function (Section 4.2), we transform this small-step abstract machine into a big-step one applying a technique known as "lightweight fusion by fixed-point promotion" [7] (Section 4.3). This machine exhibits a number of corridor transitions, which we compress (Section 4.4). We then flatten its configurations and rename its transition functions to something more intuitive (Section 4.5). We also switch domains of evaluator functions to factor out artifacts of the hybrid language (Section 4.6). The resulting abstract machine is in defunctionalized form, so we refunctionalize it (Section 4.7). The result is in continuation-passing style, so we transform it into direct style (Section 4.8). The final result is a traditional compositional type-checker.

Standard ML (SML) [15] is used as a metalanguage. SML is a statically-typed, call-by-value language with computational effects. In Section 4.8 we rely on the library of undelimited continuations to model top-level exceptions. For the sake of brevity, we omit most program artifacts (sometimes only giving their signature), keeping only essential parts to demonstrate the corresponding program transformation.[2] At each transformation stage the trailing index of all involved functions is incremented.

3. A Reduction-Based Type Checker

This section provides the initial implementation of $\lambda_{\mathcal{H}}$ in SML, which will be used for further transformations in Section 4.

3.1. Reduction-based hybrid term normalization

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.

In the specification of $\lambda_{\mathcal{H}}$, the contraction of lambda expressions (rule [tc-lam]) is specified using a meta-level notion of capture-avoiding substitutions. However, most implementations do not use actual substitutions and keep an *explicit representation* of what should be substituted on demand, leaving the term untouched [10, pages 100–105]. To model *explicit substitutions*, we chose the applicative order version of Curien's calculus, which uses closures, i.e, terms together with their lexical environment [3][3]. The environments map variables to values (i.e., types in this case) while reducing an expression, which corresponds do the capture-avoiding substitution strategy [5, Section 6]. The chosen calculus allows us to come eventually in Section 4 to a well-known representation of a type-checking algorithm with an environment Γ, which predictably serves the same purpose.

[2]The accompanying code and the technical report with detailed listings are available from `http://people.cs.kuleuven.be/ilya.sergey/type-reduction/`

[3]The cited paper also relates values in the language of closures with values in λ-calculus (see Section 2.5).

3.2. Abstract syntax of $\lambda_{\mathcal{H}}$: closures and values

The abstract syntax for $\lambda_{\mathcal{H}}$, which is presented in Figure 1, is described in SML below. It includes integer literals, identifiers, lambda-abstractions, applications as well as "hybrid" elements such as numeric types and arrows $\tau \rightarrow e$. 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.

```
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

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
```

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 [18], we leave environments parametrized by the type parameter `'a`, which is instantiated with `typ` in this case.

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

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.

```
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
```

We also specify the corresponding embeddings values to closures and terms to hybrid terms (the definitions are omitted):

```
val type_to_closure : typ -> closure
val term_to_hterm : term -> hterm
```

3.3. Notion of contraction

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.

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

A potential redex may trigger a contraction or it may get stuck. These outcomes are captured by the following datatype:

```
datatype contract_or_error = CONTRACTUM of closure
                           | ERROR of string
```

The string content of ERROR is an error message.

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.

```
(*  contract: potential_redex -> contract_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"
```

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.

3.4. Reduction strategy

Reduction contexts are defined as follows:

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

A context is a closure with a hole, represented inside-out in a zipper-like fashion. 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. These possibilities are captured by the following datatype:

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

A decomposition function recursively searches for the left-most inner-most redex in a closure. Examples of some specific decomposition functions may be found in recent work of Danvy [5]. 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. The signatures of these functions are:

```
val decompose_closure : closure * hctx -> type_or_decomposition
val decompose_context : hctx * typ -> type_or_decomposition
val decompose : closure -> type_or_decomposition
val recompose : hctx * closure -> closure
```

3.5. Reduction-based normalization

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` from Section 3.3 and then be recomposed. If during contraction an error occurs, it must be reported:

```
datatype result = RESULT of typ
                | WRONG of string

(*  iterate: type_or_decomposition -> result -> 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)
```

At this point we should take into account the fact 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.

```
(*  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
```

4. From Reduction-Based to Compositional Type Checker

In this section we follow a systematic approach to the construction of a reduction-free normalization function out of a reduction-based normalization function [5].

4.1. 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 3.

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

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.

4.2. Inlining the contraction function

We inline the function `contract` (Section 3.3) 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`.

4.3. 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 [7]:

- `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`.

4.4. 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.

4.5. 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. Therefore, closures are no longer involved in computations, and the former hybrid contexts now look as follows:

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

datatype result = RESULT of value
                | WRONG of string

val eval5: hterm * bindings * context -> result

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

4.6. 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:

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

This might come as a surprise, but the resulting abstract machine is the well-known environment-based CEK machine [11].

4.7. Refunctionalization

The abstract machine obtained in the previous section is in fact in defunctionalized form [9]: 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* [8]. The resulting refunctionalized program is a compositional evaluation function in continuation-passing style.

4.8. 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.

```
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
```

The resulting program is a traditional evaluator for type checking, such as the one described by Pierce [17, pages 113-116]. The only one difference is that our implementation uses undelimited continuations via `callcc` to propagate encountered type errors whereas a classical implementation would just perform some additional check in each clause of the `eval` function or use the exceptions. This last transition completes the chain of transformations.

5. Related work and applications

The functional correspondence between different semantics artifacts has been applied to various tasks. Ager et al. [1] 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. More recently, Anton and Thiemann [2] 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.

Reduction semantics for type inference provides a powerful framework to implement type debuggers and improve the quality of error messages. Currently, the majority of techniques used for this task rely on program slicing [19]. The explicit notion of evaluation context for type inference can provide better information for type reification based on the expected type of an expression, as it is done, for instance, in the Scala programming language [16].

6. 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.

Acknowledgements

We would like to express our sincerest gratitude to Olivier Danvy for his encouragement and insightful comments on the paper, and for suggesting the problem in the first place. We are grateful to José Proença for proof-reading a draft version. We also thank the anonymous reviewers for the comments and especially for the suggestions on the cleanup and reorganization of the accompanying code.

References

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

[2] K. Anton, P. Thiemann, Deriving type systems and implementations for coroutines, in: APLAS '10, pp. 63–79.

[3] M. Biernacka, O. Danvy, A concrete framework for environment machines, Transactions on Computational Logic 9 (2007) 6+.

[4] O. Danvy, Defunctionalized interpreters for programming languages, in: ICFP '08, pp. 131–142.

[5] O. Danvy, From Reduction-Based to Reduction-Free Normalization, in: P. Koopman, R. Plasmeijer, D. Swierstra (Eds.), Advanced Functional Programming, Sixth International School, Nijmegen, The Netherlands, 2008, pp. 66–164. Lecture notes including 70+ exercises.

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

[7] O. Danvy, K. Millikin, On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion, Information Processing Letters 106 (2008) 100–109.

[8] O. Danvy, K. Millikin, Refunctionalization at Work, Science of Computer Programming 74 (2009) 534–549.

[9] O. Danvy, L.R. Nielsen, Defunctionalization at work, in: PPDP '01, pp. 162–174. Extended version available as the technical report BRICS RS-01-23.

[10] M. Felleisen, R.B. Findler, M. Flatt, Semantics Engineering with PLT Redex, 1st edition, The MIT Press, August 2009.

[11] M. Felleisen, D.P. Friedman, Control operators, the SECD machine, and the λ-calculus, in: M. Wirsing (Ed.), Formal Description of Programming Concepts III, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, pp. 193–217.

[12] G. Kuan, A rewriting semantics for type inference, Technical Report TR-2007-01, University of Chicago, 2007.

[13] G. Kuan, Type Checking and Inference via Reductions, in: M. Felleisen, R.B. Findler, M. Flatt (Eds.), Semantics Engineering with PLT Redex, 1st edition, The MIT Press, August 2009, pp. 403–428.

[14] G. Kuan, D. MacQueen, R.B. Findler, A rewriting semantics for type inference, in: ESOP '07, pp. 426–440.

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

[16] M. Odersky, The Scala Language Specification, version 2.9, 2011. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[17] B.C. Pierce, Types and programming languages, MIT Press, Cambridge, MA, USA, 2002.

[18] I. Sergey, D. Clarke, From type checking by recursive descent to type checking with an abstract machine, in: LDTA '11, pp. 9–15.

[19] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (1995) 121–189.