Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 229 (5) (2011) 3–17

www.elsevier.com/locate/entcs

A Partial Type Checking Algorithm for Type : Type

Andreas Abel[1,3]

Institut für Informatik, Ludwigs-Maximilians-Universität München, Oettingenstraße 67, D-80538 München, Germany

Thorsten Altenkirch[1,2,4]

School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, United Kingdom

Abstract

We analyze a partial type checking algorithm for the inconsistent domain-free pure type system Type:Type ($\lambda*$). We show that the algorithm is sound and partially complete using a coinductive specification of algorithmic equality. This entails that the algorithm will only diverge due to the presence of diverging computations, in particular it will terminate for all typeable terms.

Keywords: Dependent Types, Pure Type Systems, Type Checking, Type:Type

1 Introduction

In this paper, we analyze and implement a partial type checking algorithm for the inconsistent theory Type:Type ($\lambda*$) similar to the one presented in [6]. This is an instance of a domain-free pure type system [4] and it seems possible to extend it to any functional pure type system (PTS). The motivation for this work is to implement type checkers for dependently typed programming languages which support general recursion such as Augustsson's Cayenne [3]. We use Type:Type as a test case for a language with dependent types avoiding the syntactic complexity of a full programming language.

[1] Research supported by FP6 IST coordination action *TYPES* (510996).

[2] Research supported by EPSRC grant *Observational Equality For Dependently Typed Programming* (EP/C512022/1)

[3] Email: andreas.abel@ifi.lmu.de

[4] Email: txa@cs.nott.ac.uk

doi:10.1016/j.entcs.2011.02.013

Our main contribution is that we show soundness and partial completeness. By partial completeness we mean that if the algorithm diverges, it will do only because the program or its type, or their combination, contains some loop; divergence because of an error in the algorithm is excluded. Hence, for a given PTS it is sufficient to establish termination to show that the algorithm is complete and does indeed decide the typing relation. We believe that this is a promising approach, because it means we can establish basic syntactic properties of the typing algorithm independently of termination.

In particular, we give algorithmic typing rules $\Gamma \vdash t \Leftarrow A$, read *in context* Γ, *term* t *checks against type* A, in two versions: $\Gamma \vdash^{\mu} t \Leftarrow A$, using inductive equality, and $\Gamma \vdash^{\nu} t \Leftarrow A$, using coinductive equality. The inductive version of the algorithm is shown sound, whereas the coinductive version is proven complete.

We present the algorithm for Type:Type *(type is a type)* with type equality by untyped β-conversion $=_\beta$. Our proofs crucially rely on the injectivity of the function type constructor $\Pi x : A.\, B$ which is a consequence of confluence of β-reduction in our case.

The type checking algorithm computes weak head normal forms (whnf) of types. This is sufficient, because β-reduction is standardizing. Standardization can be subsumed by the slogan *if a term β-reduces to a whnf, then weak head reduction reaches a whnf of the same shape.* For instance, if $t \longrightarrow_\beta^* \lambda xu$, then $t \longrightarrow_{\mathsf{w}}^* \lambda xu'$ with $u' \longrightarrow_\beta^* u$. With confluence, this becomes: if $t =_\beta^* \lambda xu$, then $t \longrightarrow_{\mathsf{w}}^* \lambda xu'$ with $u' =_\beta^* u$.

Related work

The algorithm presented here is basically a modern reimplementation of Coquand's algorithm [6], see also [7], but the study of partial completeness using coinduction is new. The fact that we consider only β-equality simplifies the treatment — a syntactic study of $\beta\eta$-equality along the lines of [9,8] is left for future work. The recent work by the first author [1] is also directed at $\beta\eta$-equality but relies on normalization.

Overview

We start by presenting Type:Type and verifying some basic properties. Next we specify the type checking algorithm in relational form and show soundness of the inductive type checking relation. The completeness of the coinductive relation is then established using coinduction. Finally we present an implementation of the algorithm in Haskell and discuss further extensions of the present work.

2 Type:Type

The Curry-style λ^* is a domain free pure type system [5] with just one sort Type, axiom $\mathsf{Type} : \mathsf{Type}$ and rule $(\mathsf{Type}, \mathsf{Type}, \mathsf{Type})$.

Syntax

As usual for pure type systems, there is only one grammatical class $\mathsf{Expression}$ for terms t, u, types A, B, C, and sorts s. Metavariable x ranges over a countably infinite set of variable identifiers.

$$
\begin{array}{lll}
\mathsf{Exp} \ni t, u, A, B, C, s ::= x \mid \lambda x t \mid t\, u \mid \Pi x{:}A.\, B \mid \mathsf{Type} & & \text{expressions} \\
\mathsf{Ne} \ni n ::= x \mid n\, u & & \text{neutral terms} \\
\mathsf{Cxt} \ni \Gamma ::= \diamond \mid \Gamma, x{:}A & & \text{typing contexts}
\end{array}
$$

We identify expressions up to α-conversion. A context Γ is just a list of pairs $x{:}A$, but it is also considered a finite map from variables to types. Hence, no variable may be assigned two types in a context.

Capture-avoiding substitution of u for x in t is written $t[u/x]$. One-step β-reduction is denoted by $\longrightarrow_\beta$, its reflexive-transitive closure by $\longrightarrow_\beta^*$ and its reflexive-transitive-symmetric closure by $=_\beta$. By confluence, $t =_\beta t'$ if and only if there is some u with $t \longrightarrow_\beta^* u$ and $t' \longrightarrow_\beta^* u$. Weak head reduction is given by the rule

$$(\lambda x t)\, u\, u_1 \dots u_n \longrightarrow_{\mathsf{w}} t[u/x]\, u_1 \dots u_n$$

for $n \geqslant 0$. Its reflexive-transitive closure is written $\longrightarrow_{\mathsf{w}}^*$. (Typeable) whnfs are neutral terms n, abstractions $\lambda x t$, function types $\Pi x{:}A.\, B$, and the constant Type. In the following we employ a vector notation and write $t\, u_1 \dots u_n$ simply as $t\, \boldsymbol{u}$.

Proposition 2.1 (Standardization of β-reduction [11])

(i) *If* $t \longrightarrow_\beta^* x\, \boldsymbol{u}'$ *then* $t \longrightarrow_{\mathsf{w}}^* x\, \boldsymbol{u}$ *and* $\boldsymbol{u} \longrightarrow_\beta^* \boldsymbol{u}'$.

(ii) *If* $t \longrightarrow_\beta^* \lambda x u'$ *then* $t \longrightarrow_{\mathsf{w}}^* \lambda x u$ *and* $u \longrightarrow_\beta^* u'$.

(iii) *If* $C \longrightarrow_\beta^* \Pi x{:}A'.\, B'$ *then* $C \longrightarrow_{\mathsf{w}}^* \Pi x{:}A.\, B$ *and* $A \longrightarrow_\beta^* A'$ *and* $B \longrightarrow_\beta^* B'$.

(iv) *If* $C \longrightarrow_\beta^* \mathsf{Type}$ *then* $C \longrightarrow_{\mathsf{w}}^* \mathsf{Type}$.

Using confluence, $\longrightarrow_\beta^*$ can be replaced by $=_\beta$ in the above statements. In particular, we can derive the following corollary from confluence:

Corollary 2.2 (Injectivity of Π) *If* $\Pi x : A.\, B =_\beta \Pi x : A'.\, B'$ *then* $A =_\beta A'$ *and* $B =_\beta B'$.

Inference rules of $\lambda*$

The terms t of type A are given by the judgement $\Gamma \vdash t : A$ which is mutually defined with the judgement $\Gamma \vdash \mathsf{ok}$ for well-formed contexts. If J is a judgement, we write $\mathcal{D} :: J$ to express that $\mathcal{D}$ is a derivation of J.

Well-formed contexts $\Gamma \vdash \mathsf{ok}$

$$\textsc{cxt-empty}\ \frac{}{\diamond \vdash \mathsf{ok}} \qquad \textsc{cxt-ext}\ \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x{:}A \vdash \mathsf{ok}}$$

Typing $\Gamma \vdash t : A$

$$\text{TYPE-F}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \text{FUN-F}\ \frac{\Gamma, x\!:\!A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x\!:\!A.\, B : \mathsf{Type}}$$

$$\text{HYP}\ \frac{\Gamma \vdash \mathsf{ok} \qquad (x\!:\!A) \in \Gamma}{\Gamma \vdash x : A} \qquad \text{FUN-I}\ \frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x t : \Pi x\!:\!A.\, B}$$

$$\text{FUN-E}\ \frac{\Gamma \vdash t : \Pi x\!:\!A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]} \qquad \text{CONV}\ \frac{\Gamma \vdash t : A \qquad A =_\beta B}{\Gamma \vdash t : B}$$

The judgement $\Gamma \vdash t : A$ implies $\Gamma \vdash \mathsf{ok}$, which is easy to check.

The following inversion lemma is independent of injectivity.

Lemma 2.3 (Inversion of Typing) (i) *If* $\mathcal{D} :: \Gamma \vdash \mathsf{Type} : C$ *then* $C =_\beta \mathsf{Type}$.

(ii) *If* $\mathcal{D} :: \Gamma \vdash \Pi x\!:\!A.\, B : C$ *then* $C =_\beta \mathsf{Type}$ *and* $\Gamma, x\!:\!A \vdash B : \mathsf{Type}$.

(iii) *If* $\mathcal{D} :: \Gamma \vdash x : C$ *then* $C =_\beta \Gamma(x)$.

(iv) *If* $\mathcal{D} :: \Gamma \vdash \lambda x t : C$ *then* $C =_\beta \Pi x\!:\!A.\, B$ *and* $\Gamma, x\!:\!A \vdash t : B$.

(v) *If* $\mathcal{D} :: \Gamma \vdash t\, u : C$ *then* $\Gamma \vdash t : \Pi x\!:\!A.\, B$ *with* $\Gamma \vdash u : A$ *and* $C =_\beta B[u/x]$.

Proof. By induction on $\mathcal{D}$. □

Typing enjoys the usual properties of weakening, substitution, and subject reduction for β. The proofs are standard.

3 A Type-Checking Algorithm

The most elementary format of a strongly typed functional program is a list of non-recursive declarations of the form $x : A = t$, meaning identifier x of type A is defined as term t. In a list of declarations, later declarations may rely on the type and definition of previously declared identifiers. It is reasonable to assume that both t and A are free of β-redexes, however, during type-checking redexes will occur in types.

We use a bidirectional representation of algorithmic type checking, using $\Gamma \vdash t \rightrightarrows A$ to denote that the type A of t can be inferred and $\Gamma \vdash t \leftleftarrows A$ that t can be checked to have type A.

A program is type checked by first ensuring that A is a well-formed type, written $\Gamma \vdash A \rightrightarrows \mathsf{Type}$, then checking that t is of type A, written $\Gamma \vdash t \leftleftarrows A$, adding the declaration $x : A = t$ to the global environment and proceeding with the next declaration.

Type inference

$\Gamma \vdash t \rightrightarrows A$. (Input: Γ well-formed, t neutral and β-normal. Output: A with $\Gamma \vdash t : A$.)

$$\textsc{inf-var}\ \frac{}{\Gamma \vdash x \rightrightarrows \Gamma(x)}$$

$$\textsc{inf-fun-e}\ \frac{\Gamma \vdash t \rightrightarrows C \qquad C \longrightarrow_{\mathsf{w}}^{*} \Pi x\!:\!A.\,B \qquad \Gamma \vdash u \leftleftarrows A}{\Gamma \vdash t\,u \rightrightarrows B[u/x]}$$

$$\textsc{inf-type}\ \frac{}{\Gamma \vdash \mathsf{Type} \rightrightarrows \mathsf{Type}}$$

$$\textsc{inf-fun-f}\ \frac{\Gamma \vdash A \rightrightarrows s \qquad s \longrightarrow_{\mathsf{w}}^{*} \mathsf{Type} \qquad \Gamma,\ x\!:\!A \vdash B \rightrightarrows s' \qquad s' \longrightarrow_{\mathsf{w}}^{*} \mathsf{Type}}{\Gamma \vdash \Pi x\!:\!A.\,B \rightrightarrows \mathsf{Type}}$$

Type inference diverges for applications $t\,u$ when the inferred type of t has no whnf. We don't specify here that the result of type inference has to be a whnf, even though we will use whnfs in the implementation. Indeed, any inferred type will have to be reduced to a whnf when it is used anyway.

Type checking

$\Gamma \vdash t \leftleftarrows A$. (Input: Γ, A with $\Gamma \vdash A : \mathsf{Type}$, t β-normal. Output: none.)

$$\textsc{chk-inf}\ \frac{\Gamma \vdash t \rightrightarrows A \qquad \vdash A \sim A'}{\Gamma \vdash t \leftleftarrows A'}\ t \text{ not a } \lambda$$

$$\textsc{chk-fun-i}\ \frac{C \longrightarrow_{\mathsf{w}}^{*} \Pi x\!:\!A.\,B \qquad \Gamma, x\!:\!A \vdash t \leftleftarrows B}{\Gamma \vdash \lambda x t \leftleftarrows C}$$

Rule CHK-RED is applied when we want to check an abstraction against a type which is not yet in whnf. Checking against a type which has no whnf diverges.

Algorithmic equality

$\vdash A \sim A'$. If the type of a term t is declared as A' but inferred as A (rule CHK-INF), we need to ensure that A and A' are β-equal. The following rules specify an algorithm which alternates weak head normalization (AQ-RED-L and AQ-RED-R) and structural comparison (the other rules).

$$\textsc{aq-red-l}\ \frac{t_1 \longrightarrow_{\mathsf{w}} t_1' \qquad \vdash t_1' \sim t_2}{\vdash t_1 \sim t_2} \qquad \textsc{aq-red-r}\ \frac{t_2 \longrightarrow_{\mathsf{w}} t_2' \qquad \vdash t_1 \sim t_2'}{\vdash t_1 \sim t_2}$$

$$\textsc{aq-var}\ \frac{}{\vdash x \sim x} \qquad \textsc{aq-app}\ \frac{\vdash n \sim n' \qquad \vdash u \sim u'}{\vdash n\,u \sim n'\,u'} \qquad \textsc{aq-}\lambda\ \frac{\vdash t \sim t'}{\vdash \lambda x t \sim \lambda x t'}$$

$$\textsc{aq-type}\ \frac{}{\vdash \mathsf{Type} \sim \mathsf{Type}} \qquad \textsc{aq-fun}\ \frac{\vdash A \sim A' \qquad \vdash B \sim B'}{\vdash \Pi x\!:\!A.\,B \sim \Pi x\!:\!A'.\,B'}$$

4 Soundness

A terminating run of the type checker corresponds to a *finite* derivation in the system of algorithmic rules presented above. Hence, when we want to reason that the algorithm is sound, i. e., that it only accepts well-typed terms, we need to consider *inductive* algorithmic equality $\vdash^{\mu} t \sim t'$ and algorithmic typing $\Gamma \vdash^{\mu} t \leftleftarrows/\rightrightarrows A$ which refers to *inductive* equality.

Lemma 4.1 (Soundness of algorithmic equality) *$\mathcal{D} :: \vdash^{\mu} t \sim t'$ implies $t =_{\beta} t'$.*

Proof. Trivially by induction on $\mathcal{D}$.

□

Theorem 4.2 (Soundness of bidirectional type checking) (i) *If $\mathcal{D} :: \Gamma \vdash^{\mu} t \rightrightarrows A$ and $\Gamma \vdash \mathsf{ok}$ then $\Gamma \vdash t : A$.*

(ii) *If $\mathcal{D} :: \Gamma \vdash^{\mu} t \leftleftarrows C$ and $\Gamma \vdash C : \mathsf{Type}$, then $\Gamma \vdash t : C$.*

Proof. Simultaneously by induction on $\mathcal{D}$. Likewise trivial.

□

5 Completeness

Since type-checking of $\lambda *$ is undecidable, an appropriate completeness result for our algorithm would be: if β-normal t is of type A, checking t against A does *not fail finitely*. I. e., the algorithm might diverge or succeed, but not report an error. We make this formal by considering the *coinductive* version of algorithmic equality $\vdash^{\nu} t \sim t'$, i. e., we allow infinite derivations, and a version of algorithmic typing $\Gamma \vdash^{\nu} t \leftleftarrows/\rightrightarrows A$ which refers to coinductive equality. In the following we prove, using the technique of coinduction [10], that finite derivations of typing and equality in the declarative system (of Section 2) map to possibly infinite derivations in the algorithmic system (of Section 3).

First we show that if two terms t_1 and t_2 are β-equal, then $\mathcal{D} :: \vdash^{\nu} t_1 \sim t_2$. In case $t_1 \equiv \Omega := (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$, the derivation $\mathcal{D}$ is simply an infinite repetition of AQ-RED-L. Note that the same derivation shows $\vdash^{\nu} \Omega \sim t$ for an *arbitrary* term t, hence, the contraposition of the following lemma cannot hold:

Lemma 5.1 (Completeness of algorithmic equality) *If $t_1 =_{\beta} t_2$ then $\vdash^{\nu} t_1 \sim t_2$.*

Proof. By coinduction. We consider the following cases:

- Case $t_1 \longrightarrow_{\mathsf{w}} t_1'$. Then $\vdash^{\nu} t_1 \sim t_2$ follows by rule AQ-RED-L using coinductive hypothesis $\vdash^{\nu} t_1' \sim t_2$.
- Case $t_2 \longrightarrow_{\mathsf{w}} t_2'$. Analogously.

In the remaining cases, t_1 and t_2 are whnfs.

- Case $t_1 \equiv \mathsf{Type} =_\beta t_2$. By confluence, $t_2 \longrightarrow_\beta^* \mathsf{Type}$. Since t_2 is a whnf, $t_2 \equiv \mathsf{Type}$. The goal follows by AQ-TYPE.
- Case $t_1 \equiv \Pi x\!:\!A_1.\,B_1 =_\beta t_2$. By confluence, $t_2 \equiv \Pi x\!:\!A_2.\,B_2$ with $A_1 =_\beta A_2$ and $B_1 =_\beta B_2$. The goal follows by AQ-FUN with coinductive hypotheses $\vdash^\nu A_1 \sim A_2$ and $\vdash^\nu B_1 \sim B_2$.

The other cases are proven analogously.

□

Next we show that for a well-typed and checkable (i. e., β-normal) term t there is an algorithmic typing derivation with possibly infinite derivations of algorithmic equality.

Theorem 5.2 (Completeness of type checking) *Let t β-normal and $\Gamma \vdash t : C$.*

(i) *If t is neutral then $\Gamma \vdash^\nu t \rightrightarrows A$ and $A =_\beta C$.*

(ii) *In any case, $\Gamma \vdash^\nu t \leftleftarrows C$.*

Proof. Simultaneously by induction on t.

- Case $t \equiv x$. By inversion $C =_\beta \Gamma(x)$. We have $\Gamma \vdash^\nu x \rightrightarrows \Gamma(x)$ by INF-VAR. The second goal follows since by Lemma 5.1 $\vdash^\nu \Gamma(x) \sim C$.
- Case $t \equiv n\,u$. By inversion, $\Gamma \vdash n : \Pi x\!:\!A.\,B$ with $\Gamma \vdash u : A$ and $C =_\beta B[u/x]$. By induction hypothesis, $\Gamma \vdash^\nu n \rightrightarrows D$ with $D =_\beta \Pi x : A.\,B$. By confluence and standardization, $D \longrightarrow_{\mathsf{w}}^* \Pi x : A'.\,B'$ with $A =_\beta A'$ and $B =_\beta B'$. Since by the conversion rule, $\Gamma \vdash u : A'$ we have by second induction hypothesis $\Gamma \vdash^\nu u \leftleftarrows A'$, hence, by INF-FUN-E we can conclude $\Gamma \vdash^\nu n\,u \rightrightarrows B'[u/x]$ with $B'[u/x] =_\beta B[u/x] =_\beta C$. This implies the second goal $\Gamma \vdash^\nu t \leftleftarrows C$.
- Case $t \equiv \mathsf{Type}$. By inversion $C =_\beta \mathsf{Type}$. We conclude by INF-TYPE.
- Case $t \equiv \Pi x\!:\!A.\,B$. By inversion, $C =_\beta \mathsf{Type}$ and $\Gamma, x\!:\!A \vdash B : \mathsf{Type}$ which implies $\Gamma \vdash A : \mathsf{Type}$. By the first induction hypothesis we have $\Gamma \vdash^\nu A \rightrightarrows s$ with $s =_\beta \mathsf{Type}$. By second induction hypothesis, $\Gamma, x\!:\!A \vdash^\nu B : s'$ with $s' =_\beta \mathsf{Type}$. Since by confluence and standardization $s \longrightarrow_{\mathsf{w}}^* \mathsf{Type}$ and $s' \longrightarrow_{\mathsf{w}}^* \mathsf{Type}$, we conclude by INF-FUN-F.
- Case $t \equiv \lambda x t'$. By inversion, $C =_\beta \Pi x : A.\,B$ and $\Gamma, x : A \vdash t' : B$. Since $C \longrightarrow_{\mathsf{w}}^* \Pi x\!:\!A'.\,B'$ with $A =_\beta A'$ and $B =_\beta B'$, we have $\Gamma, x\!:\!A' \vdash t' : B'$. By induction hypothesis $\Gamma, x\!:\!A' \vdash t' \leftleftarrows B'$ and we conclude by CHK-FUN-I.

□

Completeness leads to the following important corollary which shows that the only reason that the algorithm will reject a typeable term is non-termination:

Corollary 5.3 *Let t β-normal and $\Gamma \vdash t : C$ but $\Gamma \not\vdash^\mu t \leftleftarrows C$. Then a subterm of t has an inferred or ascribed type which is not strongly normalizing.*

Proof. From 5.2 we know that $\mathcal{D} :: \Gamma \vdash^\nu t \leftleftarrows C$. Since $\vdash$ and $\vdash^\nu$ differ only in the equality check, there must be types A and A' with an infinite derivation of

$\vdash^\nu A \sim A'$ contained in $\mathcal{D}$. This derivation must contain infinitely many applications of AQ-RED-L or AQ-RED-R, thus, A or A' is not strongly normalizing. $\square$

6 Haskell Implementation

In the following, we present a Haskell implementation of our type checking algorithm for $\lambda *$. We choose an efficient implementation of substitution and weak head reduction through closures. In the end, it is very similar to Coquand's algorithm [6], however, we distinguish closures and weak head normal forms through different data types, making some invariants explicit this way. Also, we explicitly use monads, and this in an abstract way that makes the implementation extensible, e. g., to universe inference.

We use monads for handling of errors and lookup in the typing context, which is implemented by finite maps.

```
module TypeType where
import Control.Monad.Error
import Control.Monad.Reader
import Data.Map (Map)
import qualified Data.Map as Map
```

Syntax

as parsed from a file is represented by abstract syntax trees of (Haskell) type *Exp*. Variables are referred to by *Name*. We maintain the invariant that function types appear only in the form *Pi a* (*Abs x b*).

```
type Name = String
data Exp
  = Var Name
  | Abs Name Exp
  | App Exp Exp
  | Pi Exp Exp
  | Type
    deriving Show
arr a b = Pi a (Abs "_" b)
```

Values and environments

Evaluation is lazy, so values are closures *Clos t rho*, pairs of an expression t and an environment *rho*. When type checking the body of an *Abs*traction, the free variable is mapped a unique *Id*, called a *generic value Gen* by Coquand [6]. Thus, the environment component *rho* may map variable names either to generic values or to closures in turn. The (Haskell) type e of environments is passed as a parameter to *Val*, since we do not want to commit to a particular representation of environments here.

```
type Id = Int
data Show e ⇒ Val e
  = Gen Id
  | Clos Exp e
    deriving Show
type Ty e = Val e
```

The weak head normal form (whnf) of a closure might either be an introduction, *WType*, *WPi*, or *WAbs*, or an elimination of a generic value, *WNe*, i.e., an identifier applied to several closures. Evaluation does not step under binders, thus, the whnf of a function closure *Clos* (*Abs x t*) *rho* is simply *WAbs x t rho*.

```
data Show e ⇒ Whnf e
  = WNe Id [Val e]   -- reversed list of arguments
  | WAbs Name Exp e
  | WPi (Val e) (Val e)
  | WType
    deriving Show
type WTy e = Whnf e
```

Environments, which map names to values, are left abstract. We specify them via the type class *Env*, providing operations for construction (*emptyEnv* and *extEnv*, extension) and query (*lookupEnv*).

```
class Show e ⇒ Env e where
  emptyEnv :: e
  extEnv    :: Name → Val e → e → e
  lookupEnv :: e → Name → Val e
```

Evaluation and application

whnf computes the weak head normal form of a value, by removing the weak head β-redexes. There are two cases of values: generic values *Gen*, which are already weak head normal, and closures, which we normalize using the auxiliary function *whnf′*.

```
whnf :: Env e ⇒ Val e → Whnf e
whnf (Gen i)        = WNe i []
whnf (Clos t rho) = whnf′ t rho
```

whnf′ computes the whnf of an expression in an environment *rho*. The value of variables *Var x* is looked up in the environment. The result might be a closure which has to be evaluated recursively. Or, it might be a generic value, in case x has become free by stepping under its binder. Applications are the source of redexes, which are resolved lazily (cbn), using function *app*. Expressions of the other shapes, *Abs*, *Pi*, and *Type*, are already whnfs.

$$
\begin{array}{lll}
\mathit{whnf}' :: \mathit{Env}\ e \Rightarrow \mathit{Exp} \rightarrow e \rightarrow \mathit{Whnf}\ e \\
\mathit{whnf}'\ (\mathit{Var}\ x) & \mathit{rho} = \mathit{whnf}\ (\mathit{lookupEnv}\ \mathit{rho}\ x) \\
\mathit{whnf}'\ (\mathit{App}\ t\ u) & \mathit{rho} = \mathit{app}\ (\mathit{whnf}'\ t\ \mathit{rho})\ (\mathit{Clos}\ u\ \mathit{rho}) \\
\mathit{whnf}'\ (\mathit{Abs}\ x\ t) & \mathit{rho} = \mathit{WAbs}\ x\ t\ \mathit{rho} \\
\mathit{whnf}'\ (\mathit{Pi}\ a\ b) & \mathit{rho} = \mathit{WPi}\ (\mathit{Clos}\ a\ \mathit{rho})\ (\mathit{Clos}\ b\ \mathit{rho}) \\
\mathit{whnf}'\ \mathit{Type} & \mathit{rho} = \mathit{WType}
\end{array}
$$

app applies a whnf to a closure, reducing the result to a whnf. The function part can only be neutral or an abstraction, other cases are impossible since ill-typed.

$$
\begin{array}{ll}
\mathit{app} :: \mathit{Env}\ e \Rightarrow \mathit{Whnf}\ e \rightarrow \mathit{Val}\ e \rightarrow \mathit{Whnf}\ e \\
\mathit{app}\ (\mathit{WNe}\ i\ \mathit{vs}) & v = \mathit{WNe}\ i\ (v : \mathit{vs}) \\
\mathit{app}\ (\mathit{WAbs}\ x\ t\ \mathit{rho})\ & v = \mathit{whnf}'\ t\ (\mathit{extEnv}\ x\ v\ \mathit{rho})
\end{array}
$$

A context for type checking

We hide the context in a monad of class *MonadCxt*. The context provides both a type and a value for each name. *bind* extends the context with both type and value. *new* extends it with the given type, creating a new generic value. *new′* creates just a generic value, in situations where its type does not matter.

The type of a name can be queried by *typeOf*, and expression can be *close*d in the context which acts like an environment in this case (this is the only way we need to refer to the values of names).

$$
\begin{array}{l}
\mathbf{class}\ (\mathit{Env}\ e, \mathit{Monad}\ m) \Rightarrow \mathit{MonadCxt}\ e\ m \mid m \rightarrow e\ \mathbf{where} \\
\quad \mathit{bind} \quad :: \mathit{Name} \rightarrow \mathit{Ty}\ e \rightarrow \mathit{Val}\ e \rightarrow m\ a \rightarrow m\ a \\
\quad \mathit{new} \quad :: \mathit{Name} \rightarrow \mathit{Ty}\ e \rightarrow (\mathit{Val}\ e \rightarrow m\ a) \rightarrow m\ a \\
\quad \mathit{new}' \quad :: \mathit{Name} \rightarrow (\mathit{Val}\ e \rightarrow m\ a) \rightarrow m\ a \\
\quad \mathit{new}'\ x = \mathit{new}\ x\ \mathit{dontCare} \\
\quad \mathit{typeOf} :: \mathit{Name} \rightarrow m\ (\mathit{Ty}\ e) \\
\quad \mathit{close} \quad :: \mathit{Exp} \rightarrow m\ (\mathit{Val}\ e) \\
\mathit{dontCare} = \mathit{error}\ \texttt{"Internal error: no type assigned to variable"}
\end{array}
$$

Bidirectional type checking

infer t infers the type of expression t, returning it in whnf. Inferable are all expressions shapes except abstractions.

For a variable, the type is looked up in the context and then weak head normalized. This does not introduce unnecessary divergence, since an inferred type needs always to be converted to weak head normal form, either to check whether it is a function type (see case *App*), or to compare it to another type (see *eq* below). Note however, that types in the context are not in weak head normal form. Normalizing them before adding them to the context would indeed introduce unnecessary divergence, e.g., for unused variables of diverging type.

$$
\begin{array}{l}
\mathit{infer} :: \mathit{MonadCxt}\ e\ m \Rightarrow \mathit{Exp} \to m\ (\mathit{WTy}\ e)\\
\mathit{infer}\ (\mathit{Var}\ x) \quad = \mathit{typeOf}\ x \ggg= \mathit{return} \circ \mathit{whnf}\\
\mathit{infer}\ (\mathit{App}\ t\ u) = \mathbf{do}\ w \leftarrow \mathit{infer}\ t\\
\qquad\qquad\qquad\qquad\quad \mathbf{case}\ w\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad \mathit{WPi}\ v\ f \to \mathbf{do}\ \mathit{check}\ u\ v\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad u' \leftarrow \mathit{close}\ u\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{whnf}\ f\ `\mathit{app}`\ u')\\
\qquad\qquad\qquad\qquad\qquad \_ \to \mathit{fail}\ (\texttt{"expected "} +\!\!+\ \mathit{show}\ t\ +\!\!+\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{" to be of function type"})\\
\mathit{infer}\ \mathit{Type} \qquad\ = \mathit{return}\ \mathit{WType}\\
\mathit{infer}\ (\mathit{Pi}\ a\ b) \quad = \mathbf{do}\ \mathit{check}'\ a\ \mathit{WType}\\
\qquad\qquad\qquad\qquad\quad v \leftarrow \mathit{close}\ (a\ `\mathit{arr}`\ \mathit{Type})\\
\qquad\qquad\qquad\qquad\quad \mathit{check}\ b\ v\\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ \mathit{WType}
\end{array}
$$

check t v checks expression t against type value v by converting the type to weak head normal form and calling *check*′. *check*′ treats only abstractions *Abs* x t, which must be of function type *Pi* v f, and their body t must type check in the context extended by x whose type is v and whose value is set to a new generic value i. The type of non-abstractions t is inferred as w' and compared to the ascribed type w.

$$
\begin{array}{l}
\mathit{check} :: \mathit{MonadCxt}\ e\ m \Rightarrow \mathit{Exp} \to \mathit{Ty}\ e \to m\ ()\\
\mathit{check}\ t\ v = \mathit{check}'\ t\ (\mathit{whnf}\ v)\\[4pt]
\mathit{check}' :: \mathit{MonadCxt}\ e\ m \Rightarrow \mathit{Exp} \to \mathit{WTy}\ e \to m\ ()\\
\mathit{check}'\ (\mathit{Abs}\ x\ t)\ (\mathit{WPi}\ v\ f) = \mathit{new}\ x\ v\ (\lambda i \to \mathit{check}'\ t\ (\mathit{whnf}\ f\ `\mathit{app}`\ i))\\
\mathit{check}'\ (\mathit{Abs}\ x\ t)\ w \qquad\quad\ = \mathit{fail}\ (\texttt{"expected "} +\!\!+\ \mathit{show}\ w\ +\!\!+\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{" to be a function type"})\\
\mathit{check}'\ t \qquad\qquad w \qquad\quad\ = \mathbf{do}\ w' \leftarrow \mathit{infer}\ t\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{eq}\ w'\ w
\end{array}
$$

Equality checking of values. We define three mutually recursive functions, each returning a monadic boolean m (). *eq* operates on whnfs, *eq*′ on arbitrary closures, *eqs* compares lists of closures of the same length. Two function closures *WAbs* are tested for equality by applying them to a new generic value i.

$$
\begin{array}{l}
\mathit{eq} :: \mathit{MonadCxt}\ e\ m \Rightarrow \mathit{Whnf}\ e \to \mathit{Whnf}\ e \to m\ ()\\
\mathit{eq}\ \mathit{WType} \qquad\quad \mathit{WType} \qquad\qquad = \mathit{return}\ ()\\
\mathit{eq}\ (\mathit{WPi}\ a\ b) \quad\ (\mathit{WPi}\ a'\ b') \qquad = \mathit{eq}'\ a\ a' \gg \mathit{eq}'\ b\ b'\\
\mathit{eq}\ v@(\mathit{WAbs}\{\,\})\ v'@(\mathit{WAbs}\ x\ \_\ \_) = \mathit{new}'\ x\ (\lambda i \to \mathit{eq}\ (v\ `\mathit{app}`\ i)\ (v'\ `\mathit{app}`\ i))
\end{array}
$$

$eq\ (WNe\ i\ vs)\quad (WNe\ i'\ vs')\mid i \equiv i' = eqs\ vs\ vs'$
$eq\ w\ w' = fail\ (\texttt{"equality check fails for "} +\!\!+\ show\ w\ +\!\!+$
$\qquad\qquad\texttt{" and "} +\!\!+\ show\ w')$

$eq' :: MonadCxt\ e\ m \Rightarrow Val\ e \to Val\ e \to m\ ()$
$eq'\ v\ v' = eq\ (whnf\ v)\ (whnf\ v')$

$eqs :: MonadCxt\ e\ m \Rightarrow [Val\ e] \to [Val\ e] \to m\ ()$
$eqs\ [\,]\qquad [\,]\qquad = return\ ()$
$eqs\ (v : vs)\ (v' : vs') = eq'\ v\ v' \gg eqs\ vs\ vs'$
$eqs\ vs\ vs' = fail\ (\texttt{"equality check fails: "} +\!\!+$
$\qquad\qquad\texttt{"argument vectors of different lengths"})$

Declarations

Input to the type checker are declarations of the form $x : A = t$ meaning name x has type A and definition t. The type checker will first ensure that A is a well-formed type, evaluate it (lazily), then check t against the value of A, and finally bind x to type value of A and the value of t in the current environment. Then it will go on to the next declaration.

data $Decl = Decl\{name :: Name, ty :: Exp, value :: Exp\}$ **deriving** $Show$

$checkDecl :: MonadCxt\ e\ m \Rightarrow Decl \to m\ (Ty\ e, Val\ e)$
$checkDecl\ (Decl\ x\ a\ t) =$ **do**
$\quad check'\ a\ WType$
$\quad v \leftarrow close\ a$
$\quad check\ t\ v$
$\quad w \leftarrow close\ t$
$\quad return\ (v, w)$

type $Decls = [Decl]$

$checkDecls :: MonadCxt\ e\ m \Rightarrow Decls \to m\ ()$
$checkDecls\ [\,] = return\ ()$
$checkDecls\ (d : ds) =$ **do**
$\quad (a, v) \leftarrow checkDecl\ d$
$\quad bind\ (name\ d)\ a\ v\ (checkDecls\ ds)$

An implementation of contexts

We implement contexts as finite maps from names to their type and value. They also handle the generation of fresh identifiers. To this end, the next unused generic value is store in field *nextFree*. *cxtLookup* just retrieves the type of a name, *cxtExt* just binds a type to a name, and *cxtBind* binds both type and value to a name.

data $Cxt = Cxt\{nextFree :: Int$
$\qquad\qquad , cxt :: Map\ Name\ (Ty\ Cxt, Val\ Cxt)\}$
$\quad$ **deriving** $Show$

$cxtLookup :: Monad\ m \Rightarrow Cxt \rightarrow Name \rightarrow m\ (Ty\ Cxt)$
$cxtLookup\ gamma\ x = \mathbf{case}\ Map.lookup\ x\ (cxt\ gamma)\ \mathbf{of}$
$\quad Just\ (a, v) \rightarrow return\ a$
$\quad Nothing \rightarrow fail\ (\texttt{"identifier not in scope: "} +\!\!+\ x)$

$cxtEmpty :: Cxt$
$cxtEmpty = Cxt\ 0\ Map.empty$

$cxtExt :: Name \rightarrow Ty\ Cxt \rightarrow Cxt \rightarrow Cxt$
$cxtExt\ x\ a\ (Cxt\ n\ gamma) = Cxt\ (n + 1)\ (Map.insert\ x\ (a, Gen\ n)\ gamma)$

$cxtBind :: Name \rightarrow Ty\ Cxt \rightarrow Val\ Cxt \rightarrow Cxt \rightarrow Cxt$
$cxtBind\ x\ a\ v\ gamma = gamma\{cxt = Map.insert\ x\ (a, v)\ (cxt\ gamma)\}$

Contexts can be seen as environments, since they provide a value for each name.

$\mathbf{instance}\ Env\ Cxt\ \mathbf{where}$
$\quad emptyEnv = cxtEmpty$
$\quad extEnv\ x\ v\ rho = rho\{cxt = Map.insert\ x\ (dontCare, v)\ (cxt\ rho)\}$
$\quad lookupEnv\ rho\ x \mid Just\ (a, v) \leftarrow Map.lookup\ x\ (cxt\ rho) = v$

Implementation of the type checking monad

During type checking, we need to query the context and we need to raise errors. The type checking monad wraps a reader monad *ReaderT Cxt* (see module *Control.Monad.Reader*) around an error monad *Either String*. The implementation of the *MonadCxt* operations access the context through the *MonadReader* operation *ask* and modify it through *local*. The Reader Monad here is only used to hide the *plumbing* used in a standard implementation of static binding. In particular shadowing of variables is implemented by replacing the previous definition.

$\mathbf{type}\ TC = ReaderT\ Cxt\ (Either\ String)$

$\mathbf{instance}\ MonadCxt\ Cxt\ TC\ \mathbf{where}$
$\quad typeOf\ x \quad = \mathbf{do}\ gamma \leftarrow ask$
$\qquad\qquad\qquad cxtLookup\ gamma\ x$
$\quad close\ t \quad = \mathbf{do}\ rho \leftarrow ask$
$\qquad\qquad\qquad return\ (Clos\ t\ rho)$
$\quad new\ x\ a\ f \quad = \mathbf{do}\ gamma \leftarrow ask$
$\qquad\qquad\qquad local\ (cxtExt\ x\ a)\ (f\ (Gen\ (nextFree\ gamma)))$
$\quad bind\ x\ a\ v\ c = local\ (cxtBind\ x\ a\ v)\ c$

The implementation of the main type checking loop uses the reader monad to type check a sequence of declarations.

$checkFile :: Decls \rightarrow IO\ ()$
$checkFile\ ds = \mathbf{case}\ (checkDecls\ ds\ \text{`}runReaderT\text{`}\ cxtEmpty)\ \mathbf{of}$
$\quad Right\ () \rightarrow putStrLn\ \texttt{"Type checking succeeded"}$
$\quad Left\ \ s \rightarrow putStrLn\ (\texttt{"Type checking error: "} +\!\!+\ s)$

7 Conclusion

We have presented a correct partial type checking algorithm for $\lambda *$ which has non-normalizing types. It should be possible to extend the algorithm for functional PTS by annotating types with sorts—however, there is a known issue with the abstraction rule which needs to be investigated (see [12]).

We have shown that the algorithm will only fail because of the presence of diverging terms during type checking (Corollary 5.3). This does not mean that the algorithm could not be improved, e.g., it could check for syntactic equality before normalizing terms. However, in practice we are interested in type checking in a normalizing fragment of the theory anyway. Indeed, for a given PTS we only have to show normalization to be able to conclude that our algorithm decides the typing relation. Thus, apart from being applicable for non-terminating type systems our paper also suggests a new way of showing decidability of terminating type theories: as in this paper, one can prove partial correctness of type checking, and then show normalization separately which entails decidability of type checking.

The proof presented here should be also extensible to languages with explicit recursion and additional features to model dependent data types, e.g., we plan to apply it to ΠΣ, a core language for dependently typed programming [2].

Another line of research would be to extend our approach to $\lambda *$ with $\beta\eta$-equality using a type-sensitive implementation of the equality checker. The problem is that the separation of equality checking and type checking does not work anymore—however, we conjecture that such an algorithm would still be sound and partially complete.

Acknowledgement

The Haskell code has been typeset by `lhs2TeX` (Andres Löh and Ralf Hinze).

References

[1] Abel, A., T. Coquand and P. Dybjer, *Verifying a semantic βη-conversion test for Martin-Löf type theory*, in: P. Audebaud and C. Paulin-Mohring, eds., "Proc. of 9th Int. Conf. on Mathematics of Program Construction, MPC 2008 (Marseille, July 2008)," Lecture Notes in Computer Science **5133**, Springer, 2008, pp. 29–56.

[2] Altenkirch, T. and N. Oury, ΠΣ*: A core language for dependently typed programming*, draft, 2008, available at http://www.cs.nott.ac.uk/~txa/publ/.

[3] Augustsson, L., *Cayenne - a language with dependent types*, in: "Proc. of 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '98 (Baltimore, MD, Sept. 1998)," ACM Press, 1999, pp. 239–250.

[4] Barthe, G. and T. Coquand, *Remarks on the equational theory of non-normalizing pure type systems*, J. of Funct. Program. **16**(2) (2006), pp. 137–155.

[5] Barthe, G. and M. H. Sørensen, *Domain-free pure type systems*, J. of Funct. Program. **10**(5) (2000), pp. 417–452.

[6] Coquand, T., *An algorithm for type-checking dependent types*, Sci. of Comput. Program. **26**(1–3) (1996), pp. 167–177.

[7] Coquand, T. and M. Takeyama, *An implementation of Type : Type*, in: P. Callaghan, Z. Luo, J. McKinna and R. Pollack, eds., "Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 2000 (Durham, Dec. 2000)," Lecture Notes in Computer Science **2277**, Springer, 2002, pp. 53–62.

[8] Goguen, H., "A Typed Operational Semantics for Type Theory," PhD thesis, University of Edinburgh, 1994.

[9] Goguen, H., *Justifying algorithms for $\beta\eta$ conversion*, in: V. Sassone, ed., "Proc. of 8th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2005 (Edinburgh, Apr. 2005)," Lecture Notes in Computer Science **3441**, Springer, 2005, pp. 410–424.

[10] Gordon, A., *A tutorial on co-induction and functional programming*, in: K. Hammond, D. N. Turner and P. M. Sansom, eds., "Proc. of 1994 Glasgow Wksh. on Functional Programming (Ayr, Sept. 1994)", Workshops in Computing, Springer, 1995, pp. 78–95.

[11] Plotkin, G., *Call-by-name, call-by-value, and the λ-calculus*, Theor. Comput. Sci. **1** (1975), pp. 125–159.

[12] Pollack, R., "The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions," PhD thesis, University of Edinburgh, 1994.