A Multi-Stage Language with Intensional Analysis

Marcos Viera    Alberto Pardo
Instituto de Computación
Universidad de la República
Montevideo - Uruguay
{mviera,pardo}@fing.edu.uy

Abstract

This paper presents the definition of a language with reflection primitives. The language is a homogeneous multi-stage language that provides the capacity of code analysis by the inclusion of a pattern matching mechanism that permits inspection of the structure of quoted expressions and their destruction into component subparts. Quoted expressions include an explicit annotation of their context which is used for dynamic inference of type, where a dynamic typing discipline based on Hinze and Cheney’s approach is used for typing quoted expressions.

This paper follows the approach of Sheard and Pasalic about the use of the meta-language \( \Omega \) as a tool for language design. In this sense, it is shown how to represent the syntax, the static as well as the dynamic semantics of the proposed language in terms of \( \Omega \) mega constructs.

Categories and Subject Descriptors    D.3.1 [Programming Languages]: Formal Definitions and Theory – Theory; D.3.4 [Programming Languages]: Processors – Code Generation

General Terms    Design, Languages, Theory

Keywords    Reflection, Multi-stage Programming, Intensional Analysis, Dynamics

1. Introduction

With the evolution of computer systems and their growing complexity it has become more and more important to take into account the way to improve their flexibility. In order to provide systems with the ability to evolve during its own execution, programming languages should support reflection, understanding it as the ability to “reason about itself”.

In this paper, we propose a multi-stage language with intensional analysis, understanding intensional analysis as the ability of a homogeneous meta-system to observe the structure of its object-programs. This is carried out by a pattern matching mechanism that is used to inspect the structure of quoted expressions and destruct them into their component subparts.

In most multi-stage languages the type of quoted expressions is \( \langle \tau \rangle \) (or \( \text{cod} \), meaning code of \( \tau \), for \( \tau \) the type of the expression being quoted). This typing statically ensures that dynamically generated programs are type-safe, but excludes some functions that destruct or traverse the structure of expressions. Other approaches [20, 8, 22] assign the same type \( \text{cod} \) to all quoted expressions, performing their type checking at run-time. Such languages make a tradeoff between static and dynamic typing. We follow these ideas using the techniques proposed by Cheney and Hinze [5] and Baars and Swierstra [2] for encoding dynamic typing. So, our language somehow relaxes static safety in favour of retaining flexibility.

Our type for quoted expressions is of the form \( \text{cod}^\ast \), being \( \Gamma \) a type context reflecting, like Naneyevsky [11, 12, 13] names, the free variables of the expression. When an expression is quoted, its type context needs to be explicitly annotated as it is necessary for dynamic type inference.

We follow the approach of Sheard and Pasalic [19, 18, 21, 14] about the use of \( \Omega \) mega as a tool for language design. Languages are encoded as object-program representation that enforces the semantic invariants of scoping and typing rules. The type system of \( \Omega \) mega then guarantees that all meta-programs respect these additional object-language properties. In the following subsection we briefly describe some of the \( \Omega \) mega features we use, for further information see the mentioned works.
1.1 Omega

Ωmega is based on Haskell, although it is strict and doesn’t have a class system. Some of its most important features are the so-called Generalized Algebraic Data Types and an extensible kind systems, which make it possible to state and enforce interesting properties of programs using the type system.

Generalized Algebraic Data Types (GADTs) are a generalization of Algebraic Data Types (ADTs). GADTs remove the restriction for parameterized ADTs which states that the range of every constructor must be a polymorphic instance of the type constructor being defined. This is possible by introducing an alternative syntax for data types declarations, where the type being defined is given an explicit kind, and every constructor is given an explicit type. For example, the type constructor Term has kind *0 > > *0, taking types to types, and represents a typed object-language:

```
data Term :: *0 ~> *0 where
  Const :: a -> Term a
  Pair :: Term a -> Term b -> Term (a,b)
  App :: Term (a -> b) -> Term a -> Term b
  
  The only restriction on constructors’ type is that their range must be a fully applied instance of the type being defined. For example, the range of the constructor Pair is a non-polymorphic instance of Term. Observe that the type argument of Term is used to stand for the object level type of the represented term.

In the same way types classify values, types are classified by kinds. Kinds are implicit in functional languages like Haskell, and can only be either the base kind (k0), which classifies types, or higher order kinds (κ1 ~> κ2), which classify type constructors. In Ωmega, new kinds can be introduced by a kind declaration, which is analogous to a data declaration. Instead of introducing value constructors, a kind declaration introduces type constructors that produce types classified by that kind.

Sheard [18] proposes using Ωmega to explore the design of new languages by encoding language semantics as meta-programs. The language is defined as a GADT. Each GADT represents a judgment, and its constructors encode the typing rules. Type parameters may have an arbitrary structure, because of definition of new kinds, and correspond to static semantics properties. These properties are checked and maintained by Ωmega’s type system. Ωmega’s type system also guarantees that meta-level programs maintain object level type-safety. A big step semantics can be defined as an interpreter or evaluation function, or a small step semantics can be defined in terms of substitutions over the term language. The typing of this function maintains object level type-safety.

1.2 Structure of the paper

The paper is organized as follows. In Section 2 we introduce a language with reflection primitives and present its static semantics. Section 3 shows how the static semantics of the language is encoded in Ωmega. Section 4 describes a big step semantics of the language in the form of an Ωmega function. We discuss related work in Section 5. Finally, Section 6 draws some conclusions.

2. Language

The aim of this paper is the proposal of a language with linguistic reflection primitives that permit us to perform type-safe intensional code analysis. In this section we define the syntax and static semantics of that language.

2.1 Basic Calculus

The core of the language is a Church-style [1] simply typed λ-calculus, with the following syntax:

```
Types τ ∈ T ::= int | bool | τ → τ | τ × τ
Contexts Γ ∈ G ::= | Γ, τ
Ctxt. Stacks P ∈ GS ::= | (P, Γ)
Variables v ∈ V ::= z | s v
Terms e ∈ E ::= b | i | (e1, e2) | fst e | snd e | λx. e | fix e | v^x | e1 e2 | if e1 then e2 else e3
```

A type τ can be a base type int or bool, a function type τ → τ or a binary product τ × τ. De Brujin indices [6] are used to encode variables bindings, so variables are natural numbers and type contexts are sequences of types.

The typing judgment is of the form P ; Γ ⊢ e : τ, and reads “expression e has type τ in local context Γ under stack P”. The presence of context stacks in the typing rules of Figure 1 is the only difference from standard λ-calculus, but can be ignored until we explain their use in the multi-stage extension.

```
Figure 1. Basic λ-calculus typing
```

In the typing rules for variables, the Base rule projects the 0-th index type and the Weak rule for s n projects recursively the (n + 1)-th type.

```
Figure 2. Typing rules for variables
```

2.2 Multi-stage Extension

We include some staging annotations as part of the language to build and combine pieces of code, partitioning the execution of programs into computational stages.
Annotations include brackets, escape, run and explicit substitution. We don’t include Cross-Stage Persistence in our language. Like in ref. [8, 9], this decision is based on the observations of Taha [23] that intensional analysis requires reductions not to be allowed in higher levels, which leads to a loss of confluence if cross-stage persistence is included.

The typing rules for the staged terms (Figure 3) are inspired by the “sliding band” of type contexts proposed by Sheard [18], except for the “future” stack of contexts which is unnecessary without cross-stage persistence. The “past” stack contains the contexts of the past stages that could be accessed when an escape is applied to the current context.

2.2.1 Dynamic Typing and Explicit Contexts

The type for quoted expressions is cod tf, where Γf is a type context reflecting the free variables of the expression. When an expression is quoted, a context including the free variables of the expression must be passed explicitly. Observe that Γf doesn’t need to be minimal. That is, if Γf represents all free variables in the bracketed expression Γf must fulfill the relation Γf = (Γf, Γf), meaning that all free variables in the expression have to be in Γf, but some others (Γf) could be added.

Unlike most multi-stage languages this type doesn’t include the type of the expression, so the escape annotation judgment could type wrong formed expressions. For example the expression

\[(\lambda (\mathbb{cod}^{(.)}) (\text{int}) (\mathbb{cod}^{(.)} (\mathbb{int}))))\]

is well typed, because the requirement that the bound code must be an integer expression cannot be checked statically. The type checking of this kind of expressions is deferred until run-time, and ill-typed quoted expressions evaluates to the well-typed value [1 Fail 1].

The run annotation is similar to the one proposed in [22], where a run-time type checking and unification is done to decide if code expression is executed. In the Run rule, the type of the executed quoted expression e1 must be the type of e2, called the exception expression. If its type is not the expected one or type checking fails then e2 is evaluated. The Run rule assures that only closed code can be evaluated by allowing only expressions with type cod tf, that is, without free variables.

\[
\frac{\text{Br}}{(P, \Gamma); \text{cod} tf \vdash e : \tau} \quad \frac{\text{Esc}}{(P, \Gamma); \text{cod} tf \vdash e : \tau} \quad \frac{\text{Run}}{\text{cod} tf \vdash \text{run} e1 \vdash e2 : \tau}
\]

\[
\frac{\text{Subst}}{\text{cod} tf \vdash e[\Theta] : \text{cod} tf}
\]

Figure 3. Multi-stage extension typing

2.2.2 Explicit Substitution

An explicit substitution operator over quoted expressions is included in order to provide a simple way of capturing free variables. We use the notation for substitutions of λν [3], adding an explicit annotation of the new type in the case of shifting.

The typing judgment for substitutions is of the form Γ ⇒ Θ ⇒ Γf. It relates a type context and a substitution with a “resulting” type context. Therefore, a substitution Θ over an expression typed in local context Γ results in an expression typed in local context Γf.

The typing rules are shown in Figure 4.

\[
\frac{P; \Gamma \vdash e : \tau}{\Gamma, \tau \vdash e / f \Rightarrow \Gamma} \quad \frac{\text{Shift}}{\Gamma \vdash \Theta \Rightarrow \Gammaf \quad \text{Lift}}
\]

Figure 4. Explicit substitution typing

Given an expression e of type τ in a local context Γ under any past stack P, a slash (e/) replaces the first variable by e and decrements the indexes of the remaining variables by one. Shift (↑) increments the indexes of all variables by one and appends the type τ at index 0. Applying lift (↑γ), the 0-index type τ remains unchanged and the substitution Θ is applied to the rest of the context. For example, the expression

\[(\#0)\text{int}\cdot\text{bool} \Rightarrow (\#1)\text{int}\cdot\text{bool}]\]

would reduce to a code, with type cod tf, corresponding to the expression: (\#0)\text{int}\cdot\text{bool} \Rightarrow (9, \#1)\text{bool}).

2.3 Intensional Analysis Extension

In order to provide intensional code analysis we extend the calculus with an alternation primitive, similar to the one proposed in [8], where variables are bound by a pattern matching mechanism.

Terms

\[e \in \mathbb{E} ::= \ldots | \lambda \nu e1 \cdot e2\]

Patterns

\[p \in \mathbb{P} ::= i \cdot b | (p1, p2) | \text{lit} | [\mathbb{pc}]\]

Code Patterns

\[\mathbb{pc} \in \mathbb{PC} ::= \text{\$}(\text{lit}) | \text{\$}(\mathbb{fail}) | \text{\$}(\text{pc}1, \text{pc}2) | \text{\$}\text{\text{lit}} | \text{\text{and}} \mathbb{pc} | \text{\text{ill}} \mathbb{pc} | \text{\text{fix}} \mathbb{pc} | \text{\text{v}} \mathbb{pc}1 \cdot \text{pc}2 | \text{\text{pc}1 \cdot \text{pc}2 | \text{\text{pc}1 \text{pc}2}}\]

The semantics of patterns is inspired by the pattern matching mechanism defined by Pasalic and Linger [15]. In that work, a pattern judgment Γ ⇒ p : τ ⇒ Γf involves an “input” type context Γ, a pattern p, which should match a value of type τ, and a resulting type context Γf. This context extends Γ with the types of the pattern variables. Based on the fact that the only change possible to an “input” context is its extension with the free variables of p, and in order to simplify the dynamic semantics of substitutions over alternations (see section 4.2), we had omitted the “input” context in the pattern judgment. So the judgment is of the form Γ ⇒ p : τ ⇒ Γ, meaning that a pattern p (matching a value of type τ) has the free variables contained in Γ.

The Alt rule for an alternation of type \(\nu \alpha \cdot \nu \beta \Rightarrow \nu \gamma \) relates a pattern p, which should match a value of type \(\nu \alpha \cdot \nu \beta \), and an alternative expression e2 of type \(\nu \alpha \Rightarrow \nu \beta \). If p matches a value of type \(\nu \alpha \), then e2 is evaluated in local context Γ, Γα, otherwise e2 is evaluated in local context Γ and applied to the matched value.

The simplest pattern is the pattern any (\_ ) which matches any value of type τ and leaves the context unchanged. Another basic
pattern is the (nameless) variable binding pattern $(\cdot^\dagger)$, which differs from the previous one in the type annotation and the extension of the context binding the value matched. More than one variable in a pattern could be bound. The PPair rule shows how variables are related to the indexes in the resulting context. Given a pair pattern $(p_1, p_2)$, where $p_1$ and $p_2$ are related to $\Gamma'$ and $\Gamma''$ respectively, its free variables are $\Gamma'$, $\Gamma''$. So the variables of the furthest to the right subpattern $(p_2)$ would be those of smaller indices in the context. This can be taken as a general rule for patterns with multiple variables.

\[
\begin{align*}
\Gamma \vdash p : \tau_1 \Rightarrow \Gamma' \\
P_1; \Gamma, \Gamma' \vdash e_1 : \tau_2 \\
P_2; \Gamma \vdash \lambda p. e_1|_2 : \tau_1 \Rightarrow \tau_2
\end{align*}
\]

\[\text{Alt}\]

Figure 5. Alternation typing

Code analysis is carried out with the help of code patterns. Their typing rules are shown in Figure 7. The judgment $\Gamma^f \vdash \mathit{pc} \Rightarrow \Gamma^r$ expresses that a pattern $\mathit{pc}$, which should match a quoted expression with type $\mathit{cod}^{\mathit{pc}}$, has the variables contained in $\Gamma^r$.

Most code patterns consist in destructing the expression and applying code patterns to the subexpressions. The any $(\_)$ code pattern matches any code, while the fail (fail) code pattern matches only failed code. Both patterns leave the context unchanged.

The syntax of variable binding $(\$ (\cdot \dagger))$ and literal binding $(\$ (\text{lit}))$ code patterns suggest their semantics in the sense that they only match with expressions that when unquoted have type $\tau$. Having fulfilled this constraint the former matches any value while the latter matches only quoted literal expressions. Both patterns extend the context with the code value matched.

The variable constant behaves like literal constants (i and b). It matches with code which quoted expression is exactly the variable $\mathit{v}^e$, unchanging the context. The any bracket code pattern matches any brackets quoted expression with free variables $\mathit{G}^{\mathit{f}}$. Given a quoted explicit substitution, $e[\Theta^e]$, the substitution code pattern requires $\Theta$ to be equal to $\mathit{Theta}^f$ and matches the code pattern $\mathit{pc}$ with $e$.

An example of code analysis is the following:

\[
\lambda [\text{if } #0^{\mathit{bool}} \text{ then } $(\cdot | \mathit{int})$ \text{ else } $(\cdot | \mathit{int})] \Rightarrow \lambda \mathit{cod}^{\mathit{bool}}. [10] \{. \}
\]

This expression takes a code value with type $\mathit{cod}^{\mathit{bool}}$ and returns one with type $\mathit{cod}^{\mathit{bool}}$. If the code passed is a quotation of an “if-then-else” expression, with condition $\#0^{\mathit{bool}}$, a code with the “then” subexpression is returned, with a True literal in each occurrence of the variable $\#0^{\mathit{bool}}$. Otherwise, the returned value is a code of the literal 0.

3. Static Semantics as an Omega GADT

In this section we will encode the typing judgments of section 2 as Omega GADTs. A value of each datatype then represents a derivation of the encoded judgment. This ensures that the properties of the static semantics are checked and maintained by the meta-language type system.

The expression judgment $P; \Gamma \vdash e : \tau$ is represented by the multiple indexed type $(\\mathit{Exp} \ \mathit{p} \ \mathit{n} \ \mathit{t})$. The “past” stack $P$ is tracked by the first index, a nested product type, which contains types of
kind Row *0 1 representing type contexts. The next index is a Row *0 tracking the current context type Γ. Finally, t tracks the term type τ. The Omega encoding of the rules showed in Figures 1, 3 and 4 is the following:

data Exp :: *0 -> Row *0 -> *0 -> *0 where
  ELBool :: Bool -> Exp p n Bool
  ELInt :: Int -> Exp p n Int
  EPair :: Exp p n t -> Exp p n s
  -> Exp p n (t,s)
  EPFst :: Exp p n (t,s,t) -> Exp p n t
  EPSnd :: Exp p n (t,s,t) -> Exp p n s
  EAbs :: Rep s -> Exp p (RCons s n) t
  -> Exp p n (s->t)
  EFix :: Exp p (RCons t n) t
  -> Exp p n t
  EApp :: Exp p n (s->t) -> Exp p n s
  -> Exp p n t
  EVar :: Var n t -> Rep t
  -> Exp p n t
  ECond :: Exp p n Bool
  -> Exp p n t
  -> Exp p n t
  -> Exp p n t
  EBr :: Exp (p,Env env) c t -> RepEnv c
t
  -> Exp p n (Cod c)
  ERun :: Exp p n (Cod nil) -> Exp p n t
  -> Exp p n t
  EEsc :: Exp p b (Cod n) -> Rep t
  -> Exp (p, Env b) n t
  ESubst :: Exp p n (Cod f) -> Subst f fc
t
  -> Exp p n (Cod fc)
  EAlt :: Pat s c -> Exp p (eapp c n) t
  -> Exp p n (s->t)

Each constructor has the structure of a formal judgment. For example, EApp takes two arguments Exp p n (s->t) and Exp p n s. These arguments correspond to the judgments P; Γ ⊢ e₁ : τ₂ → τ₁ and P; Γ ⊢ e₂ : τ₂, respectively. If these can be supplied, the constructor results in the type Exp p n t, encoding P; Γ ⊢ e₁ e₂ : τ₂.

In EAbs, EVar and EEsc a type must be annotated. This is done by an argument of type Rep t, the parametric type representation defined both by Cheney and Hinze [5] and Baars and Swierstra [2] for dynamic typing:

data Rep :: *0 -> *0 where
  Int :: Rep Int
  Bool :: Rep Bool
  Arr :: Rep a -> Rep b -> Rep(a -> b)
  Prod :: Rep a -> Rep b -> Rep(a,b)
  Cod :: RepEnv n -> (Rep (Cod n))

These type annotations are used to carry out the run-time type checking in the same way dynamic typing is handled in the works mentioned previously.

The EVar constructor includes the Var n t sub-judgment, where VZ and VS encode the rules Base and Weak of Figure 2.

data Var :: Row *0 -> *0 -> *0 where
  VZ :: Var (RCons t env)
  VS :: Var env t -> Var (RCons s env) t

Observe that a context extension Γ', τ is represented by the Row constructor (RCons t env).

The stacks of contexts are nested pairs. A type Env, which is indexed by a Row *0, is used to push a context. This is done because the pair constructor takes only types of kind *0.

data Env :: Row *0 -> *0 where
  EnvNil :: Env
  EnvCons :: t -> Env r -> Env (RCons t r)

Multi-stage annotations involves expressions with type cod*.

The encoding of this type in Omega has the following definition:

data Cod :: Row *0 -> *0 where
  Q :: (forall p. Exp p n t) -> RepEnv n
  -> Cod n
  F :: RepEnv n -> Cod n

Because of dynamic typing, it could happen that an expression evaluates to a bad formed code value. For this reason, the type Cod has two constructors: one for well formed quoted expressions and another for failed ones. A well formed code is an expression at level 0, typed in a given environment. A term at level 0 has no escapes at level 0. This is captured by requiring that the past contexts stack is universally quantified. Both in the case of well formed code like for failed code, a representation of the context is passed as an argument. This representation has type RepEnv.

data RepEnv :: Row *0 -> *0 where
  RepEnvNil :: RepEnv
  RepEnvCons :: Rep t -> RepEnv r
  -> RepEnv (RCons t r)

This type classifies lists of Rep t and is indexed by a Row *0. Type RepEnv is also used in the constructor EBr to represent the free variables of the expression.

The substitutions judgment is encoded by the datatype Subst. Like in the Q constructor for Cod, the expression passed to the SSInh constructor must carry an universally quantified past contexts stack.

data Subst :: Row *0 -> *0 where
  SSft :: Rep t -> Subst n (RCons t n)
  SLeft :: Subst n c
  -> Subst (RCons t n) (RCons t c)
  SSInh :: (forall p. Exp p n t)
  -> Subst (RCons t n) n

So, the encoding for the explicit substitution example of section 2.2.2 is:

(ESubst
  (EBr
    (EApp
      (EVar VZ (Arr (Prod Int Bool)) Bool))
    (EPair
      (EVar (VS VZ) Int)
      (EVar (VS VS VZ) Bool)))
    (REnvCons (Arr (Prod Int Bool))
      (REnvCons Int (REnvCons Bool RepEnvNil))))
  (SLeft (SSInh (ELInt 9))))

In the EAlt constructor we use the type function eapp to encode a list append constraint (Γ', Γ''). It can be proven by doing induction on the first argument that this function terminates.

eapp :: Row *0 -> Row *0 -> Row *0
  {eapp RSSNil ys} = ys
  {eapp (RCons x xs) ys} = RCons x {eapp xs ys}
  {eapp {eapp xs ys} zs} = {eapp xs {eapp ys zs}}
The pattern judgment $\vdash p : \tau \Rightarrow \Gamma$ is encoded by the datatype $(\text{Pat } t n)$.

```haskell
data Pat :: *0 ~> Row *0 ~> *0 where
  PLInt :: Int -> Pat Int RNil
  PLBool :: Bool -> Pat Bool RNil
  PPair :: Pat t1 c1 -> Pat t2 c2
  -> Pat (t1,t2) {eapp c2 c1}
  PVar :: Rep t -> Pat t (RCons t RNil)
  PAny :: Pat t RNil
  PCod :: PatCod f c -> Pat (Cod f) c
```

The constructor function $\text{PCod}$ includes a sub-judgment for code patterns. The definition of the type $\text{PatCod}$, representing the code patterns judgment, is the following:

```haskell
data PatCod :: Row *0 ~> Row *0 ~> *0 where
  PCPVar :: Rep t
  -> PatCod f (RCons (Cod f) RNil)
  PCPAny :: PatCod f RNil
  ...
  PCPair :: PatCod f c1
  -> PatCod f c2
  -> PatCod f {eapp c2 c1}
  ...
  PCAbs :: Rep s -> PatCod (RCons s f) c
  -> PatCod f c
  ...
  PCCond :: PatCod f c1
  -> PatCod f c2
  -> PatCod f c3
  -> PatCod f {eapp3 c3 c2 c1}
  ...
  PCBr :: RepEnv fp
  -> PatCod f (RCons (Cod f) RNil)
  PCRun :: PatCod f c1 -> PatCod f c2
  -> PatCod f {eapp c2 c1}
  PCSubst :: PatCod fc c -> Subst f fc
  -> PatCod fc c
```

4. Dynamic Semantics as an Ω mega evaluator

Dynamic semantics for the language is given by a big-step semantics written as an evaluation function. The semantics shows that the evaluation of well typed terms doesn’t go wrong.

The evaluation function has type $\text{Exp } p n t \rightarrow \text{Env } n \rightarrow t$. Given any well typed expression $\text{Exp } p n t$ and an environment with shape $n$, $\text{eval}$ returns a value with type $t$.

```haskell
eval :: Exp p n t -> Env n -> t
eval (ELInt i) env = i
...
eval (EPair e1 e2) env = (eval e1 env, eval e2 env)
...
eval (EAbs t e) env = \ v -> eval e (EnvCons v env)
eval (EApp f x) env = (eval f env)(eval x env)
eval (EVar v t) env = evalVar v t env
...
eval (EFix e) env = lazy \((\ v ->
  (eval e (EnvCons v env)))
  (eval (EFix e) env))
eval (EBr e renv) env
  = case (bd (CountBrZ env) e) of
    (x,True) -> Q x renv
    (x,False) -> F renv
eval (ERun e1 e2) env = case (eval e1 env) of
  Q e REnvNil ->
  case eqType (getType e)
  (getEmail e2) of
  Just Eq -> eval e EnvNil
  Nothing -> eval e2 env
  _ -> eval e2 env
eval (ESubst e s) env
  = case (eval e env) of
  Q eb rb ->
  case (evalSub s eb rb) of
  (en, rn) -> Q en ren
  F rb -> F (evalSubR s rb)
eval (EAlt p e1 e2) env = \ v -> case (evalPat p v env) of
  Just env2 -> eval e1 env2
  Nothing -> (eval e2 env) v
```

This function is total excepting for the EEsc case, which is not evaluated. In an expression at level 0 will not be an escape, so the evaluation function must be defined to take expressions at level 0. This could be enforced defining an evaluation function that can only be applied to terms polymorphic in their past.

```haskell
eval0 :: (forall p. Exp p n t) -> Env n -> t
eval0 exp env = eval exp env
```

To avoid infinite loops, the Ωmega construct for explicit laziness (lazy) is used in the evaluation of EFix.

4.1 Dynamic Type Checking and Building Code

The type checking is implemented by the unification function $\text{eqType}$, which takes two type representations, tests them for structural equality, and possibly returns a proof of their equivalence. Its signature is:

```haskell
eqType :: Rep a -> Rep b -> Maybe(Equal a b)
```

During the evaluation of $\text{ERun}$, after a verification that the code is well formed, an unification between the types of the quoted expression $e$ and the exception expression $e2$ is made. If the unification succeeds, there’s a witness that the type of $e$ is the same as $e2$. So, the expression $e$ is evaluated in the empty environment (static type-checking assures that $e$ is closed). If the unification fails, the expression $e2$ is evaluated in the environment $env$. The types of $e$ and $e2$ are obtained by the type inference function $\text{getType}$, which is based on the typing rules:

```haskell
getType:: Exp p n t -> Rep t
getype (ELInt i) = Int
getype (ELBool b) = Bool
getype (EPair e1 e2) = Prod (getType e1) (getType e2)
getype (EPFst e) = case (getType e) of
  Prod r1 r2 -> r1
  ...
getype (EAbs t e) = Arr t (getType e)
getype (EApp e1 e2) = case (getType e1) of
  Arr r1 r2 -> r1
  ...
geetype (EVar v t) = t
geetype (EBr e renv) = Cod renv
getype (ESubst e s) = case (getType e) of
  Cod renv ->
```

```haskell
Cod (evalSubR s renv)
```
The core of the expression substitution is the one defined by Sheard and Pasalic [21], extended by passing the representation of the source environment.

evalSubE :: Subst g gp -> Exp p g t -> RepEnv g -> Exp p gp t

evalSubE s (ELInt i) r = ELInt i

evalSubE s (EPair e1 e2) r = EPair (evalSubE s e1 r) (evalSubE s e2 r)

evalSubE s (ESubst e sp) r = ESubst (evalSubE s r) sp

evalSubE s (EAlt p e1 e2) r = case (getType e2) of (Arr t1 t2) -> EAlt p (evalSubE (evalPatS p s) e1 (evalPatR p r t1)) (evalSubE s e2 r)

value (Just env), being env the extended environment (with type Env eapp eout e1n), and e1 is evaluated in this environment. If matching fails, a Nothing value is returned, and e2 is evaluated in the current context.

For example, if i is passed when evaluating the pattern (PLInt i), the same environment passed as argument is returned. On the other hand, evaluating (PVar t) never fails, just returning the current environment extended with the value passed.

```
evalPat :: (Pat t eout) -> t -> Env ein
          -> Maybe (Env {eapp eout ein})
```

```
evalPat (PLInt i) v env = if (i==v)
                        then (Just env)
                        else Nothing
```

```
evalPat (PPair p1 p2) (v1, v2) env = case (evalPat p1 v1 env) of
                                      Just env1 -> evalPat p2 v2 env1
                                      Nothing -> Nothing
```

```
evalPat (PVar t) v env = Just (EnvCons v env)
evalPat (PCod p) v env = evalCPat p v env
```

In the case of (PPair p1 p2), the pattern p1 is evaluated extending the current environment and then p2 is evaluated extending the environment returned by p1.

The code patterns are evaluated by the function evalCPat.

```
evalCPat :: (PatCod f eout) -> (Cod f) -> Env ein
           -> Maybe (Env {eapp eout ein})
```

```
evalCPat (PCPVar t) e env = case e of
                            Q v renv ->
                            case (eqType (getType v) t) of
                               Just Eq -> Just (EnvCons e env)
                               Nothing -> Nothing
                            _ -> Nothing
```

```
evalCPat PCPAny e env = Just env
evalCPat PCPair e env = case e of
                         F renv -> Just env
                         Q eq renv -> Nothing
```

```
evalCPat (PCLit t) e env = case e of
      Q (ELInt i) renv ->
        case (eqType (Int t)) of
           Just Eq -> Just (EnvCons e env)
           Nothing -> Nothing
      Q (ELBool b) renv ->
        case (eqType (Bool t)) of
           Just Eq -> Just (EnvCons e env)
           Nothing -> Nothing
```

```
evalCPat (PCAbs tx pb) e env = case e of
                             Q (EAbs tvx vb) renv ->
                               case (eqType tvx (Cod renv)) of
                                 Just Eq -> evalCPat pb
                                 Nothing -> Nothing
```

```
evalCPat (PCVar v t) e env = case e of
                      Q (EVar vv vt) renv ->
                        case (eqType t vt) of
                           Just Eq -> if (eqVar v vv)
                                           then Just env
                                           else Nothing
```

```
evalCPat (PCBr r) e env = case e of
                       Q (EBr ve renv) renv ->
                         case (eqType (Cod r) (Cod renv)) of
                            Just Eq -> Just env
                            Nothing -> Nothing
```

```
evalCPat (PCSubst p s) e env = case e of
                       Q (ESubst ve vs) renv ->
                         if (eqSubst s vs)
                           then evalCPat p
                                       (eval (EBr ve renv) env)
                           else Nothing
```

Consider the case of (PCAbs tx pb). If the value passed is a code (Q (EAbs tvx vb) renv) and tx represents the same type than tvx, the code pattern pb is evaluated to match with a quotation of vb with context renv extended by tvx ((REnvCons tvx renv)).

### 4.4 Soundness

The soundness of a type system with respect to the semantics means that, if a term is well-typed, then its evaluation either returns a value of same type or gives rise to an infinite reduction sequence. In other words, well-typed terms never go wrong. To prove soundness, subject reduction and progress must be proved. The former property means that reduction preserves typing while the latter means that programs which are well-typed are either values or can be further reduced (evaluation never gets stuck).

According to the type of the evaluation function, `Exp p n t -> Env n -> t`, the evaluation of any expression e satisfying the type judgment $P; \Gamma \vdash e : \tau$ yields, if it terminates, a value of type $\tau$. This means that subject reduction is automatically ensured by $\Omega$mega’s type system.
Concerning progress, observe that every well-typed term of the language always matches one of the clauses of \( \text{eval} \). Therefore, if the term is not a value, there is a reduction rule that is applicable to it.

5. Related Work

Our language is based on multi-stage languages like MetaML [23, 24, 27, 10, 26] and MetaOCaml [25, 4], with the incorporation of features presented in languages like Template Haskell [20], \( \text{reFL}^{\text{ext}} \) [8, 9] and \( \nu \) [11, 12, 13] with the aim of supporting intensional analysis in a flexible way.

Typing MetaML and MetaOCaml have static type checking, associating a type code of \( \tau \) to the quotation of an expression of type \( \tau \). On the other hand, languages like Template Haskell, \( \text{reFL}^{\text{ext}} \) and the one proposed in [22] associate a universal type code to all quotations. As a consequence, these languages need to perform a dynamic type-checking for generated code, excepting for Template Haskell which performs compile-time code generation. Our language follows the approach of [22]. We perform dynamic type-checking for generated code, avoiding run-time errors by the inclusion of an exception expression in the \( \text{run} \) construct.

In our language quoted open expressions are represented by annotating the type code with a type context, containing the types of the free variables. These variables can be captured by an explicit substitution mechanism provided by the language. This approach is similar to that of \( \nu \), which uses names to represent free variables in quoted expressions.

Intensional Analysis Neither MetaML nor MetaOCaml are proposed as code analyzers, they focus on code generation and its optimization. Taha [23, 24] argued that by introducing \( \beta \) reduction at higher levels and code inspection the property of coherence is violated. Therefore there exists many optimizations that can only be applied to code at stage 0. Moreover, cross-stage persistence, one of the most distinguishing features of these languages, can not be present as well.

In Template Haskell code is represented by an algebraic data type, allowing its inspection. In contrast, our language uses a high-level pattern matching interface to intensional analysis, in the line of \( \nu \) and \( \text{reFL}^{\text{ext}} \). \( \nu \) pattern matching is only defined over the simply typed \( \lambda \)-calculus fragment of the language. Our pattern matching mechanism is similar to the one proposed in \( \text{reFL}^{\text{ext}} \).

\( \Omega \) for language design The use of \( \Omega \) for developing the semantics of our language is inspired in the encoding of MetaML done by Sheard in [19].

6. Conclusions

In this paper we presented an homogeneous functional multi-stage language with support for intensional analysis. A pattern matching mechanism was defined as a high-level interface to perform code inspection. The type of quoted expressions reflects the free variables of the expression but not its type, which is inferred at runtime. Although ill-typed quoted expressions can be generated at run-time only well-typed generated code can be evaluated by \( \text{run} \). An explicit substitution operator over quoted expressions was included too.

The proposed language may seem impractical due to its type annotations. However, like in [8] and [22], a type annotation algorithm from implicitly typed terms to annotated terms could be defined to avoid this. This algorithm would be essentially an extension of the Hindley-Milner type inference algorithm.

Static and Dynamic Semantics were represented in \( \Omega \) by encoding the typing judgments as GADTs and defining a big-step semantics written as an evaluation function, respectively. Since the evaluation function has a case defined for any well-typed term, the \( \Omega \) implementation of the semantics showed that the evaluation of well-typed terms doesn’t go wrong.

References


