User-Defined Operators Including Name Binding for New Language Constructs

Kazuhiro Ichikawa and Shigeru Chiba

The University of Tokyo

Typed-Lambda Calculus

$$\Gamma, x : T_1 \vdash e : T_2$$

Here, $x : T_1$ expresses that

variable x of type T_1 is available in the expression e

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : T_2}$$

x is available in e_2

Setters and Getters

getter operator

setter operator

$$\Gamma, \boxed{x : T_1}, \boxed{x = \_ \; : T_1 \rightarrow Unit} \vdash e : T_2$$

Here, $x = \_ \; : T_1 \rightarrow Unit$ expresses that assignment syntax to x is available in the expression e

assignment is also typeable

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1, x = \_ \; : T_1 \rightarrow Unit \vdash e_2 : T_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : T_2}$$

assignment syntax $x = \_$ is available in e_2

Turnstile Types

A type for dealing with name binding

$$\Gamma \vdash e : T \Rightarrow e : \Gamma \vdash T$$

- regarding $\Gamma \vdash T$ as a type
- read as a type T under the context Γ
- A type can specify available variables within a scope and the availability is type-checked!

$$\frac{e_1 : \boxed{\Gamma \vdash T_1} \qquad e_2 : \boxed{(\Gamma, x : T_1, x = \_ : T_1 \to Unit) \vdash T_2}}{let\ x = e_1\ in\ e_2 : \boxed{\Gamma \vdash T_2}}$$

turnstile type

Reduction

$$\frac{\Gamma, S \vdash T}{\Gamma \vdash S \to T}$$

$$e : (\Gamma, s : T_1) \vdash T_2 \Longrightarrow \lambda s.\, e : \Gamma \vdash T_1 \to T_2$$

Turnstile type expression e behaves

as a function $T_1 \to T_2$ under the context Γ

operator s's type

$$e_2 : \boxed{(\Gamma, x : T_1, x = \_ : T_1 \to Unit) \vdash T_2}$$

$$\Longrightarrow \lambda(x = \_).\, e_2 : \boxed{(\Gamma, x : T_1) \vdash (T_1 \to Unit) \to T_2}$$

$$\Longrightarrow \lambda x.\, \lambda(x = \_).\, e_2 : \boxed{\Gamma \vdash T_1 \to (T_1 \to Unit) \to T_2}$$

ProteaJ2

Programming language implementing our idea

- Java-based language
- syntax extensions based on **user-defined operators**
- supporting **turnstile types** for expressing local syntax
- providing **DSL classes** for modularizing operators
- providing **generic names** for recognizing names
- compiler is available from github.com/csg-tokyo/proteaj2

➢ Users can define new language constructs
involving a name binding in a type-safe fashion!

User-Defined Operators

Functions with their own syntax

- extension of operator overloading
- overloaded by return type and parameter types
- proposed in our previous paper

return type

getter operator

setter operator

```
int "x" () { return value; }
void "x" "=" _ (int v) { value = v; }
```

name part

operand

operand type

Let-x Expressions

Syntax of variable x is declared as instance operators of a **DSL class**

operators
constructor
field

```
dsl VarX <T> {
  T "x" () { return value; }
  void "x" "=" _ (T v) { value = v; }
  VarX (T v) { value = v; }
  private T value;
}
```

$$\frac{e_1 : \Gamma \vdash T_1 \qquad e_2 : (\Gamma, \overbrace{x : T_1, x = \_ : T_1 \to Unit}^{\texttt{VarX<}T_1\texttt{>}}) \vdash T_2}{let\ x = e_1\ in\ e_2 : \Gamma \vdash T_2}$$

Let-x Expressions

Syntax of let expressions is declared as an operator

- the type of e2 is a turnstile type
- Γ is omitted in the operator definition because it is verbose

```
<T1, T2> T2
  "let" "x" "=" _ "in" _ (T1 e1, VarX<T1> |- T2 e2)
{ … }
```

$$\frac{e_1 : \Gamma \vdash T_1 \qquad e_2 : \Gamma, VarX{<}T_1{>} \vdash T_2}{let\ x = e_1\ in\ e_2 : \Gamma \vdash T_2}$$

Let-x Expressions

Turnstile type value behaves as a function

- value of VarX<T1> ⊢ T2 is changed to
  a function VarX<T1> → T2 in operator body

```
<T1, T2> T2
  "let" "x" "=" _ "in" _ (T1 e1, VarX<T1> |- T2 e2)
{
  return e2.apply(new VarX<T1>(e1));
}
```

Let Expression

not Let-x

To recognize arbitrary names given by users, names can be parameterized in ProteaJ2

- `X:Ident` expresses a **generic name X** and the name is an expression of type `Ident`

```
<T1, T2, X:Ident> T2
  "let" X "=" _ "in" _ (T1 e1, Var<T1, X> |- T2 e2)
{
  return e2.apply(new Var<T1, X>(e1));
}
```

Let Expression

Generic name can be used as a part of syntax

- X takes an expression of `Ident`
- Compiler checks the AST is identical to the argument of X

```
dsl Var <T, X:Ident> {
  T X () { return value; }
  void X "=" _ (T v) { value = v; }
  Var (T v) { value = v; }
  private T value;
}
```

Let Expression

Ident is not a meta type or primitive type

- Ident is a normal class
- Syntax of Ident can be defined as operators

```
literal Ident _ _ (Letter letter, Ident rest)
 { return new Ident(letter, rest); }
literal Ident _ (Letter letter)
 { return new Ident(letter); }
literal Letter "a" () { return Letter.a; }
...
literal Letter "z" () { return Letter.z; }
```

literal indicates that the syntax does not recognize whitespace characters as a separator

Case Study: Lambda Expressions

Users can define the syntax of lambda expressions

- Type parameters can be used in operator syntax

```
<T1, T2, X:Ident> Function<T1, T2>
  X ":" T1 "->" _ (Var<T1, X> |- T2 f)
{
  return new Function<T1, T2> {
    T2 apply (T1 value) {
      return f.apply(new Var<T1, X>(value));
  };
}
```

```
a: String -> b: String -> a + b
```

Case Study: Anaphoric If Statements

Anaphoric-if is well-known unhygienic macros but it can be safely implemented in ProteaJ2

```
<T> void "a-if" "(" _ ")" _ (T value, It<T> |- Void f)
{
  if (value != null) { f.apply(new It<T>(value)); }
}
dsl It <T> {
  T "it" () { return value; }
  It (T v) { value = v; }
  private T value;
}
```

```
a-if (get_nullable()) {
  System.out.println(it);
}
```

Case Study: Try-with Statements

```
<R extends AutoCloseable, X:Ident> void
 "try" "(" R X "=" _ ")" "{" _ "}"
 (R resource, Val<R, X> |- Void f) throws Exception
{
  try { f.apply(new Val<R, X>(resource)); }
  finally { if (resource != null) resource.close(); }
}
dsl Val <T, X:Ident> {
  T X () { return value; }
  Val (T v) { value = v; }
  private T value;
}
```

```
try (FileReader reader =
        new FileReader(file))
{
  parse(reader)
}
```

Case Study: For Expressions

```
<T, R, X:Ident> List<R>
 "for" "(" T X "=" _ ";" _ ";" _ ")" "yield" "{" _ "}"
 (T ini, Var<T, X> |- Boolean cond,
  Var<T, X> |- Void update, Var<T, X> |- R body)
{
  List<R> list = new ArrayList<R>();
  Var<T, X> var = new Var<T, X>(ini);
  while (cond.apply(var)) {
    list.add(body.apply(var));
    update.apply(var);
  }
  return list;
}
```

```
for (Integer x = 0;
     x < 10; x = x + 2)
yield { x * x }
// List(0, 4, 16, 36, 64)
```

Parsing

Top-down context-sensitive parsing

- environment Γ is needed for parsing
- environment Γ propagates from root to leaf of AST
- our implementation is based on packrat parser combinators

ProteaJ2 compiler uses expected types for parsing

- compiler only uses operators returning expected type

Related Work

Function literals with receiver in Kotlin

- they specify their receiver at call site
- members of the receiver are available in the literal
- instance_eval in Ruby is an untyped variant of them

➢ they can be regarded as turnstile types that only support limited syntax

receiver

```
fun html (ini: HTML.() -> Unit): HTML {
  val html = new HTML()
  html.ini()
  return html
}
```

can be called as a method of HTML

```
html {
  head {
    …
  }
}
```

method of HTML

Related Work

Implicit parameters in Scala

- if argument is not specified at call site, it is implicitly given
- argument is found among implicit values of current scope

➢ implicit values work similar to Γ (left-hand side of $\vdash$)
but users cannot flexibly control Γ

Conclusion

We proposed **turnstile types**

- $\Gamma \vdash e : T \Rightarrow e : \Gamma \vdash T$
- Γ contains typed syntax like $x = \_ : T_1 \rightarrow Unit$
- A type can specify available syntax in a restricted scope

We have developed ProteaJ2 supporting our proposal

- syntax extensions based on user-defined operators
- providing DSL classes for modularizing operators
- providing generic names for arbitrary name binding
- compiler is available from github.com/csg-tokyo/proteaj2