Incorporating First-Order Unification into Functional Language via First-Class Environments

Shin-ya Nishizaki

Department of Computer Science, Tokyo Institute of Technology
2-12-1-W8-69, O-okayama, Meguro-ku, Tokyo, 152-8552, Japan
`nisizaki@cs.titech.ac.jp`

Abstract. Unification is a useful process by which one attempts to find a substitute satisfying a given set of equations. Among several kinds of unification algorithms, the unification for equations between first-order terms is known to be decidable and to satisfy the completeness. A unification mechanism plays an important role in logic programming languages, such as Prolog. In this paper, we propose an approach to incorporating a unification mechanism into a functional programming language via first-class environments. The first-class environment is a reflective feature in a programming language, which enables us to reify environments, to handle them as first-class values such as integers and Boolean values, and to reflect the reified environment as an environment at a meta-level. By identifying resulting substitutions of unification problems as first-class environments, we can introduce unification into functional programming languages. In this paper, we first give the syntax of a simple functional language with unifications. Second, we give its operational semantics in the style of Kahn's natural semantics. Finally, we introduce some related works and show the future direction of our works.

Keywords: functional programming language, first-order unification, first-class environment, unification.

1 Introduction

1.1 First-Class Environment and Environment Calculus

In program, variables are bound to certain values and refereed in expressions. The correspondence between the variables and their values at some point in time is called an *environment*. In the semantics of programming languages, this is usually formalized by a partial function whose domain is a finite set of variables and whose codomain is a set of denotable values.

In a programming language Scheme [13], we can use two kinds of runtime objects – continuations and environments – as first-class citizens; that is, it is possible to pass such values as parameters and to return them as results. The availability of first-class continuations and environments increases the expressiveness of the programming language. In some versions of Scheme, the following primitives enable environments to be treated as first-class citizens:

- **the-environment** is a zero-ary procedure returning a representation of the current environment in which the expression itself is evaluated;
- **eval** is a binary procedure mapping the representation of an expression and the representation of an environment into the value of this expression in this environment.

An environment does not appear explicitly in a functional program's computation expressed as reduction sequences. An environment is usually represented as a list of pairs of variables and their bound denotation, which forms an implicit computational structure of the lambda-calculus.

The substitution is used as a meta-level mechanism to describe the beta-reduction of the lambda-calculus, but it is not an object-level mechanism of the lambda-calculus, since it is not an explicit operation in the lambda-calculus. The idea of using explicit substitutions [1,5–7] is an interesting approach to make substitutions work at object-level in the lambda-calculus, and explicit substitutions are formalized as object-level substitutions using an environment in the $\lambda\sigma$-calculus.

Although explicit substitutions allow us to treat an environment at object-level in the $\lambda\sigma$-calculus, there is still a crucial difference between the object-level environments of the $\lambda\sigma$-calculus and the first-class environments of Scheme. In the $\lambda\sigma$-calculus, it is not possible to pass substitutions as parameters. For instance, the following term is not permissible in the $\lambda\sigma$-calculus. $\lambda sub.(x[sub])$, where an explicit substitution is passed to the argument *sub*. The point to be stressed is that, in the $\lambda\sigma$-calculus the syntactic class of explicit substitutions is precisely distinguished from its terms. If we introduce first-class environments into the $\lambda\sigma$-calculus, we should allow $\lambda env.(x[env])$ as a permissible term in such an extended lambda-calculus. Roughly speaking, the lambda-calculus with first-class environments is an extended lambda-calculus that allows environments as additional permissible terms.

1.2 Embedding Unification into a Functional Programming Language via a First-Class Environment

Unification [2,15] is processing by which one attempts to solve the satisfiability problem given as a set of equations. The goal of unification is to find a substitution which makes each equation hold by applying to both sides. There are various kinds of unification depending on syntactic structures of terms. Unification is widely used in automated reasoning, logic programming and programming language type system implementation. In this paper, we focus on the *first-order unification* [2,15], which solves unification problems for first-order terms. Variables in first-order terms are not assumed to have functional values but individuals, similar to first-order predicate logic.

2 The Lambda Calculus with Unifications, λunify

In this section, we introduce the syntax of a theoretical programming language, λunify, which is an untyped lambda calculus into which we have incorporated first-order unification.

A set **Var** of *variables* and a set **FunSym** of *constructors* (or sometimes *function symbols*) are given in advance of the following definition of the λunify's syntax. As the first-order predicate logic and the equational logic [2], to each function symbol *f*, a non-negative integer, called *arity*, is assigned. This is written as arity(*f*). The symbols *x,y,z* are typically used for variables and *f,g,h* for function symbols.

Definition 1 (Expression of λunify). The *expressions* of λunify are inductively defined by the following grammar:

$$\begin{aligned} e ::= {} & x \mid \lambda x.e \mid (e_1\ e_2) \mid id \mid (e_1/x)\cdot e_2 \mid (e_1 \circ e_2) \mid f(e_1,\dots,e_n) \\ & \mid \{e_1 = e_1',\dots,e_m = e_m'\}\ \textbf{orelse}\ e \end{aligned}$$

The first three kinds of expression are called a variable, a *lambda-abstraction*, and a *function application*, respectively, and these are assumed to have similar meanings to the traditional lambda calculus[4]. The next three expressions are called the *identity environment*, an *environment extension*, and an *environment composition* respectively, and are the same as the environment lambda-calculi's [9,10]. The last two kinds of expression are called a *construnctor term* and a *unificand* respectively. The constructor terms are similar to the first-order terms in the predicate logic. A unificand

$$\{e_1 = e_1',\dots,e_m = e_m'\}\ \textbf{orelse}\ e$$

has the following intuitive meaning:

- Try the first-order unification of a set of equations { $e_1 = e_1$', …, $e_m = e_m$' }
  - If the unification succeeds, the unifier is regarded as a value of a first-class environment.
  - Otherwise, the expression *e* is evaluated and its value is returned.

This intuitive meaning will be formalized as the operational semantics presented in the later section. We sometimes use an abbreviation

$$\overline{\{e_m = e_m'\}}\ \textbf{orelse}\ e$$

for

$$\{e_1 = e_1',\dots,e_m = e_m'\}\ \textbf{orelse}\ e$$

3 Operational Semantics of λunify

In this paper, the operational semantics of the calculus λunify is given in the style of the *natural semantics* proposed by G. Kahn[8].

In the original natural semantics, the semantic relation takes two input arguments: the first argument is an expression to be assigned a meaning, and the second an environment that gives a meaning to each free variable occurring in the expression.

$$\langle \mathit{Expression}, \mathit{Environment} \rangle \Downarrow \mathit{Denotation}$$

Both the input arguments of the semantics relation of λunify are expressions. More precisely, the second argument is an expression denoting an environment. Though the lambda calculus cannot represent environments as expressions, our calculus can handle first-class environments and represent environments as expressions.

Definition 2 (Values). The set **Value** of *values* is defined inductively by the following grammar. Meta-variables $v, v', v_1, \ldots, v_n$ stand for values.

$$v ::= x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid (u\ v) \mid (\lambda x.e) \circ v \mid \mathit{id} \mid (v/x) \cdot v'$$

Metavariable *w* and *u* stand for elements of subsets of **Value**, which are defined inductively by the following grammar:

$$\begin{aligned} w &::= x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid (u\ v) \\ u &::= x \mid x \circ w \mid f(v_1, \ldots, v_n) \mid (u\ v) \\ &\quad \mid \mathit{id} \mid (v/x) \cdot v' \end{aligned}$$

Definition 3 (Semantic Relation). The ternary relation $\langle e, v \rangle \Downarrow v'$ among a term *e* and values v, v' is defined inductively by the following rules.

$$\frac{}{\langle x, (v/x) \cdot v' \rangle \Downarrow v}\ \textbf{VarHit} \qquad \frac{\langle x, v \rangle \Downarrow v''}{\langle x, (v/y) \cdot v' \rangle \Downarrow v''}\ \textbf{VarSkip}$$

$$\frac{}{\langle x, \mathit{id} \rangle \Downarrow x}\ \textbf{VarId} \qquad \frac{v \neq \mathit{id} \quad v \neq (v_1/x) \cdot v_2}{\langle x, v \rangle \Downarrow x \circ v}\ \textbf{VarPending}$$

$$\frac{\langle e_i, v \rangle \Downarrow v_i\ (i = 1, \ldots, n)}{\langle f(e_1, \ldots, e_n), v \rangle \Downarrow f(v_1, \ldots, v_n)}\ \textbf{Constr} \qquad \frac{}{\langle \lambda x.e, v \rangle \Downarrow (\lambda x.e) \circ v}\ \textbf{Lam}$$

$$\frac{\langle e_1, v \rangle \Downarrow (\lambda x.e_1') \circ v_1 \quad \langle e_2, v \rangle \Downarrow v_2 \quad \langle e_1', (v_2/x) \cdot v_1 \rangle \Downarrow v}{\langle (e_1\ e_2), v \rangle \Downarrow v'}\ \textbf{Beta}$$

$$\frac{\langle e_1, v \rangle \Downarrow v_1 \quad v_1 \neq (\lambda x.e_1') \circ v_1'' \quad \langle e_2, v \rangle \Downarrow v_2}{\langle (e_1\ e_2), v \rangle \Downarrow (v_1\ v_2)}\ \textbf{AppPending}$$

$$\frac{}{\langle \mathit{id}, v \rangle \Downarrow v}\ \textbf{Id}$$

$$\frac{\langle e_1, v\rangle \Downarrow v_1 \quad \langle e_2, v\rangle \Downarrow v_2}{\langle (e_1/x)\cdot e_2, v\rangle \Downarrow (v_1/x)\cdot v_2}\ \textbf{Extn} \qquad \frac{\langle e_2, v\rangle \Downarrow v_2 \quad \langle e_1, v_2\rangle \Downarrow v_1}{\langle e_1 \circ e_2, v\rangle \Downarrow v_1}\ \textbf{Comp}$$

$$\frac{\begin{cases}\langle e_i, v\rangle \Downarrow v_i\\ \langle e_i', v\rangle \Downarrow v_i'\\ \textbf{Unify}(\{\overline{v_n = v_n'}\}) = [\overline{x_m \mapsto v_m''}]\end{cases}}{\langle\{\overline{e_n = e_n'}\}\ \textbf{orelse}\ e, v\rangle \Downarrow (v_1''/x_1)\cdots(v_m''/x_m)\cdot v}\ \textbf{UnifySuccess}$$

$$\frac{\begin{cases}\langle e_i, v\rangle \Downarrow v_i\\ \langle e_i', v\rangle \Downarrow v_i'\\ \textbf{Unify}(\{\overline{v_n = v_n'}\}) = \textbf{failure}\\ \langle e, v\rangle \Downarrow v'\end{cases}}{\langle\{\overline{e_n = e_n'}\}\ \textbf{orelse}\ e, v\rangle \Downarrow v'}\ \textbf{UnifyFailure}$$

Definition 4 (Unification Procedure). *Unification procedure* **Unify** is defined by the following equations, which takes a finite set of expressions as an argument and returns either a substitution (or a *unifier*) or a failure signal **failure**.

$$\begin{aligned}
\textbf{Unify}(\{\ \}) &= [\ ],\\
\textbf{Unify}(\{v = v, \overline{v_n = v_n'}\}) &= \textbf{Unify}(\{\overline{v_n = v_n'}\}),\\
\textbf{Unify}(\{x = y, \overline{v_n = v_n'}\}) &= [x \mapsto \sigma(y)] \cup \sigma\\
&\quad \textbf{where}\ \sigma = \textbf{Unify}(\{\overline{v_n[x \mapsto y] = v_n'[x \mapsto y],}\}\\
\textbf{Unify}(\{x = v, \overline{v_n = v_n'}\}) &= \textbf{if}\ x\ \mathit{occurs\ in}\ v\ \textbf{then raise failure else}\\
&\qquad [x \mapsto \sigma(x)] \cup \sigma\\
&\qquad \textbf{where}\ \sigma = \textbf{Unify}(\{\overline{v_n[x \mapsto v] = v_n'[x \mapsto v]}\},\\
\textbf{Unify}(\{v = x, \overline{v_n = v_n'}\}) &= \textbf{Unify}(\{x = v, \overline{v_n = v_n'}\}),\\
\textbf{Unify}(\{f(\overline{v_n^1}) = f(\overline{v_n^2}), \overline{v_n = v_n'}\}) &= \textbf{Unify}(\{\overline{v_n^1 = v_n^2}, \overline{v_n = v_n'}\}),\\
\textbf{Unify}(\{\overline{v_n = v_n'}\}) &= \textbf{raise failure}.
\end{aligned}$$

4 Example of λunify

In order to give fruitful examples, we extend the language λunify by adding several basic constructs such as conditionals and the recursive operator. In the lambda calculus, it is known that such constructs are encoded. For example, the recursive fixed-point operator can be represented as

$$Y_{cbv} = \lambda f.(\lambda x.\lambda y.(f(xx)y)(\lambda x.\lambda y.(f(xx)y))$$

in the call-by-value lambda calculus. In this paper, we introduce it as a primitive construct with the following rule.

$$\frac{\langle (M(Y_{cbv}\ M)), v\rangle \Downarrow v'}{\langle (Y_{cbv}\ M), v\rangle \Downarrow v'}$$

We also introduce the conditional branch and the comparison operator as primitive operators similar to the recursive operator.

$$\frac{\langle e_1, v\rangle \Downarrow \mathbf{true} \quad \langle e_2, v\rangle \Downarrow v_2}{\langle \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, v\rangle \Downarrow v_2} \quad \frac{\langle e_1, v\rangle \Downarrow \mathbf{false} \quad \langle e_3, v\rangle \Downarrow v_3}{\langle \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, v\rangle \Downarrow v_3}$$

$$\frac{\langle e_1, v\rangle \Downarrow v' \quad \langle e_2, v\rangle \Downarrow v'}{\langle e_1 = e_2, v\rangle \Downarrow \mathbf{true}} \quad \frac{\langle e_1, v\rangle \Downarrow v_1' \quad \langle e_2, v\rangle \Downarrow v_2' \quad v_1' \neq v_2'}{\langle e_1 = e_2, v\rangle \Downarrow \mathbf{false}}$$

For describing richer examples in this section, we introduce the constant symbols **nil** and **0**, which are function symbols **succ** and **cons** of arity 1 and 2, respectively.

By using the unification mechanism of λunif, we can describe destructing of the data structures. For example, a function that returns the length of a list given as an argument is represented as follows.

$$\begin{aligned} Y_{cbv}(\lambda len.\lambda l.\ (&\mathbf{if}\ l = \mathbf{nil}\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ (\mathbf{succ}(len\ l_1)) \\ &\circ (\{l = \mathbf{cons}(a_1, l_1)\}\ \mathbf{orelse}\ id)) \end{aligned}$$

The following is a detailed explanation of the term 'λunify'.

- This term gives a recursive definition of the list-length function, using the fixed-point operator Y_{cbv}.
- Before solving the unificand { l = **cons**(a_1,l_1) }, the variable l is assumed to be bound to a list. After the unification, the variables a_1 and l_1 are bound to the head and the tail of the list, respectively.
- The conditional expression (**if** l = **nil then** 0 **else** (**succ** (*len,* l_1)) is evaluated under the environment obtained by evaluating the unificand.

We give another example of a function which searches for an item in the list; if found, it returns the item's position; otherwise, it returns (the length of the list)+1.

$$\begin{aligned} Y_{cbv}(\lambda f.\lambda l.\ (&\mathbf{if}\ a_1 = \mathbf{0}\ \mathbf{then}\ \mathbf{1}\ \mathbf{else}\ (\mathbf{succ}(f\ l_1)) \\ &\circ (\{l = \mathbf{cons}(a_1, l_1)\}\ \mathbf{orelse}\ (\mathbf{0}/a_1) \cdot id)) \end{aligned}$$

5 Concluding Remarks

In this paper, we proposed a functional programming language with a unification mechanism. We incorporated the unification by using first-class environments. We first gave the syntax of the language, and second, we gave its operational semantics in the style of Kahn's natural semantics. We finally introduced some related works and showed the future direction of our works.

Discussions. There are several studies in which the unification is embedded into programming languages. In the paradigm of functional programming, one such study is *Qute* by Sato and Sakurai[12]. In their language, the beta-reduction and the unification computation is tightly combined. The unification is processed as needed by the beta-reduction. The characteristic feature of Qute is parallel execution of the beta-reduction and the unification processing. However, handling of the variable scope is more complicated than that of λunify.

One of the future research directions of λunify is parallel execution of beta-reduction and unification processing, keeping the simple scoping feature of λunify.
In logic programming languages such as Prolog[14], the unification is the most fundamental mechanism of handling data. However, the meaning of variables in the logic programming languages is different to the other kinds of programming language. On the other hand, we succeeded in introducing unification into **λenv** keeping the standard meaning of the variables.

In this work, we focused on the first-order unification. The other kinds of unification such as higher-order unification[3], which enables us to unify the lambda terms. If we incorporate the higher-order unification into λunify, it enables us to describe programs which handle data with variable-bindings more easily. We would apply λunify to proof checking software such as the theorem prover Isabelle [11].

Acknowledgement. This work was supported by Grants-in-Aid for Scientific Research (C) (24500009). I would like to express my gratitude to Takayuki Higuchi for his collaboration and fruitful discussions in the early stage of this work.

References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. Journal of Functional Programming 1(4), 375–416 (1991)
2. Baarer, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1999)
3. Baader, F., Snyder, W.: Unification theory. In: Robinson, J., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 447–533. Elsevier Science Publishers (2001)
4. Barendregt, H.P.: The Lambda Calculus. Elsevier (1984)
5. Curien, P.L.: An abstract framework for environment machines. Theor. Comput. Sci. 82, 389–402 (1991)
6. Curien, P.L., Hardin, T., Lévy, J.-J.: Confluence properties of weak and strong calculi of explicit substitutions. J. ACM 43(2), 363–397 (1996)
7. Dowek, G., Hardin, T., Kirchner, C.: Higher-order unification via explicit substitutions, extended abstract. In: Proceedings of the Symposium on Logic in Computer Science, pp. 22–39. Springer (1987)
8. Kahn, G.: Natural Semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
9. Nishizaki, S.: Simply typed lambda calculus with first-class environments. Publications of Reseach Institute for Mathematical Sciences Kyoto University 30(6), 1055–1121 (1995)
10. Nishizaki, S.: Polymorphic environment calculus and its type inference algorithm. Higher-Order and Symbolic Computation 13(3), 239–278 (2000)