Type Inference for Bimorphic Recursion

Makoto Tatsuta
National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
tatsuta@nii.ac.jp

Ferruccio Damiani
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, I-10149 Torino, Italy
damiani@di.unito.it

This paper proposes bimorphic recursion, which is restricted polymorphic recursion such that every recursive call in the body of a function definition has the same type. Bimorphic recursion allows us to assign two different types to a recursively defined function: one is for its recursive calls and the other is for its calls outside its definition. Bimorphic recursion in this paper can be nested. This paper shows bimorphic recursion has principal types and decidable type inference. Hence bimorphic recursion gives us flexible typing for recursion with decidable type inference. This paper also shows that its typability becomes undecidable because of nesting of recursions when one removes the instantiation property from the bimorphic recursion.

1 Introduction

The Hindley-Milner system, which is called the ML type system [4] and the core of the type systems of functional programming languages like SML, OCaml, and Haskell, is only able to infer types for monomorphic recursion, that is, recursive function definitions where all the occurrences of recursive calls have the same simple type of the function definition. The problem of inferring types for polymorphic recursion, that is, recursive function definitions where different occurrences of recursive calls have different simple types that specialize the polymorphic type of the function definition [14, 17], has been studied both by people working on type systems [9, 13, 2, 11, 12, 19, 5, 18] and by people working on abstract interpretation [15, 16, 6, 7].

Type inference for polymorphic recursion was shown to be undecidable [8, 13]. For this reason, those programming languages do not use polymorphic types for recursive definitions. Haskell and OCaml allow polymorphic recursion only when we provide type annotation. On the other hand, a restricted form of polymorphic recursion could be useful for programming. It is important theoretically as well as practically to find some restriction such that it is enough flexible and its type inference is decidable.

Henglein [8] suggested that we have decidable type inference in some restricted polymorphic recursion. In this system, only one recursive call in the body of the function definition is allowed. In addition, recursive definitions must not be nested. We call this recursion the single polymorphic recursion. Type inference of the single polymorphic recursion is reduced to semi-unification problems with a single inequation, which are known to be decidable.

Our contribution is proposing bimorphic recursion and proving that it has decidable type inference. Bimorphic recursion is an extension of the single polymorphic recursion. In bimorphic recursion, each recursive call in the body of the function definition must have the same type. Recursive definitions can be nested. "Bimorphic" means that a recursively defined function can have two different types: one is for recursive calls in the body of the definition, and another is for its calls of the function outside the body of the definition. This paper shows that the system with bimorphic recursion has principal types and decidable type inference.

The idea for the type inference algorithm is based on an observation that for a given recursive definition by bimorphic recursion, we can infer its principal type by typing only the body of its definition.

Giovanna D'Agostino, Salvatore La Torre (Eds.):
Proceedings of the Second International Symposium on
"Games, Automata, Logics and Formal Verification" (GandALF 2011)
EPTCS 54, 2011, pp. 102–115, doi:10.4204/EPTCS.54.8

We do not have to think of types of calls of the function outside the body of its definition. By this idea, our algorithm first handles the innermost recursive definition and then goes to the second innermost recursive definition and so on. According to the type inference algorithm in [8, 9], our bimorphic recursion produces semi-unification problems, which are undecidable in general. Our idea enables us to reduce these semi-unification problems to semi-unification problems with a single inequation, which are decidable [8]. This algorithm can also work with polymorphic types in the let constructor. The system with bimorphic recursion and the polymorphic let constructor has principal types and decidable type inference.

Because of nesting of recursions, this idea is so subtle that it becomes unavailable even by a small change of a typing system. An example is the system with extended monomorphic recursions where every recursive call has the same type that is an instantiation of the type of the function definition. Since every recursive call has the same type as the type of the function definition in monomorphic recursion, the system is an extension of monomorphic recursion and can type more expressions. This system is also obtained from our bimorphic recursion type system by removing instantiation in the rule for recursion. The system types less expressions than our bimorphic recursion type system. Nonetheless the same idea does not work for its type inference, since the type of the function depends also on the types of its calls outside the body of its definition. Indeed this paper will show that its type inference is undecidable. This will be proved by reducing semi-unification problems to type inference. This reduction is obtained by refining the reduction of semi-unification problems to the type inference of polymorphic recursion given in [8].

About ten years ago, building on results by Cousot [3], Gori and Levi [6, 7] have developed a type abstract interpreter that is able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion. As pointed out in [1], the problem of establishing whether Gori-Levi typability is decidable is open. Since our system is inspired by [1], our bimorphic recursion type system can help to solve it.

Section 2 defines bimorphic recursion. Section 3 gives its type inference algorithm and shows bimorphic recursion has principal types and decidable type inference. Section 4 discusses its extension to the polymorphic let constructor. Section 5 studies bimorphic recursion without instantiation and shows that its type inference is undecidable. Section 6 concludes.

2 The system BR

We will define the type system BR of bimorphic recursion. We assume variables $x, y, z, \ldots$, and constants $c, \ldots$. We have expressions e defined by

$e ::= x|c|\lambda x.e|ee|\mathrm{rec}\{x = e\}$.

These consist of λ-terms with constants and rec. The expression $\mathrm{rec}\{x = e\}$ means the recursively defined function x by the equation $x = e$ where e may contain recursive calls of x. We will write $e[x := e_1]$ for the expression obtained from e by capture-avoiding substitution of e_1 for x.

We assume type variables $\alpha, \beta, \ldots$. We have types u, v, w defined by

$u, v, w ::= \alpha|\mathrm{bool}|\mathrm{int}|u \to u|u \times u|u$ list.

These consist of type variables, and the types of booleans, integers, functions, cartesian products, and lists.

We will write $\mathrm{FV}(e)$ for the set of free variables in e and $\mathrm{FTV}(u)$ for the set of free type variables in u.

A substitution s is defined as a function from type variables to types such that $\{\alpha|s(\alpha) \neq \alpha\}$ is finite. We will write $\mathrm{Dom}(s) = \{\alpha|s(\alpha) \neq \alpha\}$. We extend s to types by defining $s(u)$ by s(bool)=bool,

$$\frac{}{U,x:u\vdash x:u}\ (var) \qquad \frac{}{U\vdash c:s(u)}\ (con) \quad (\mathrm{type}(c)=u)$$

$$\frac{U,x:u\vdash e:v}{U\vdash \lambda x.e:u\to v}\ (\to I) \qquad \frac{U\vdash e_1:v\to u \quad U\vdash e_2:v}{U\vdash e_1e_2:u}\ (\to E)$$

$$\frac{U,x:s_1(u)\vdash e:u}{U\vdash \mathrm{rec}\{x=e\}:s_2(u)}\ (rec) \quad (\mathrm{Dom}(s_1),\mathrm{Dom}(s_2)\subseteq \mathrm{FTV}(u)-\mathrm{FTV}(U))$$

Figure 1: System BR

s(int)=int, $s(u\to v)=s(u)\to s(v)$, $s(u\times v)=s(u)\times s(v)$, and $s(u\ \mathrm{list})=s(u)\ \mathrm{list}$. We will use s,r for substitutions. We will write $u[\alpha:=v]$ for the type obtained from u by replacing α by v.

A type environment U is the set $\{x_1:u_1,\ldots,x_n:u_n\}$ where $u_i=u_j$ for $x_i=x_j$. We will write $\mathrm{FTV}(U)=\mathrm{FTV}(u_1)\cup\ldots\cup\mathrm{FTV}(u_n)$ and $\mathrm{Dom}(U)=\{x_1,\ldots,x_n\}$. We will use U for type environments.

A judgment is of the form $U\vdash e:u$. We will write $x_1:u_1,\ldots,x_n:u_n\vdash e:u$ when U is $\{x_1:u_1,\ldots,x_n:u_n\}$. We will write $U,x:u\vdash e:v$ for $U\cup\{x:u\}\vdash e:v$.

We assume each constant c is given its type denoted by $\mathrm{type}(c)$.

The system has the inference rules given in Figure 1.

These rules form an extension of the simply typed λ-calculus with the rules (con) and (rec). By the rule (con), the constant c has the type $s(u)$ which is an instantiation $s(u)$ of the given type u. By the rule (rec), for the recursively defined function x with its definition $x=e$, we can use some general type u to type the function. First we have to show the body e of its definition has this type u by assuming each recursive call of x in e has the unique type $s_1(u)$ which is obtained from u by instantiation with a substitution s_1. Then we can say the defined function x has the type $s_2(u)$ which is another instantiation of the type u. The side condition guarantees that s_1 and s_2 change only type variables that do not occur in U.

An expression e is defined to be typable if $\vdash e:u$ is provable for some type u. A type u is defined to be a principal type for a term e when (1) $\vdash e:u$ is provable, and (2) if $\vdash e:u'$ is provable, then there is some substitution s such that $s(u)=u'$.

Theorem 2.1 *There is a type inference algorithm for the type system BR. That is, there is an algorithm such that for a given term it returns its principal type if the term is typable, and it fails if the term is not typable.*

We will prove this theorem in the next section. This theorem can be extended to a system with polymorphic let in Section 4.

We will write $u\to v\to w$ for $u\to(v\to w)$. We assume the constants pair, fst, snd, nil, cons, hd, tl, null, 0,1,2, and ifc with $\mathrm{type}(\mathrm{pair})=\alpha_1\to\alpha_2\to(\alpha_1\times\alpha_2)$, $\mathrm{type}(\mathrm{fst})=(\alpha_1\times\alpha_2)\to\alpha_1$, $\mathrm{type}(\mathrm{snd})=(\alpha_1\times\alpha_2)\to\alpha_2$, $\mathrm{type}(\mathrm{nil})=\alpha\ \mathrm{list}$, $\mathrm{type}(\mathrm{cons})=\alpha\to\alpha\ \mathrm{list}\to\alpha\ \mathrm{list}$, $\mathrm{type}(\mathrm{hd})=\alpha\ \mathrm{list}\to\alpha$, $\mathrm{type}(\mathrm{tl})=\alpha\ \mathrm{list}\to\alpha\ \mathrm{list}$, $\mathrm{type}(\mathrm{null})=\alpha\ \mathrm{list}\to\mathrm{bool}$, $\mathrm{type}(0)=\mathrm{type}(1)=\mathrm{type}(2)=\mathrm{int}$, and $\mathrm{type}(\mathrm{ifc})=\mathrm{bool}\to\alpha\to\alpha\to\alpha$. They are the pair, the first projection, the second projection, the empty list, the list construction, the head function for lists, the tail function for lists, the null function for the empty list, three integers, and the if-then-else statement respectively. We will use the following

abbreviations.

$$[e_1.e_2] = \text{cons } e_1 e_2,$$
$$[\,] = \text{nil},$$
$$[e_1, e_2, \ldots, e_n] = [e_1.[e_2. \ldots [e_n.[\,]] \ldots]],$$
$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 = \text{ifc } e_1 e_2 e_3,$$

We will explain bimorphic recursion by examples.

Example 2.2 This example is a list doubling function by using a dispatcher.

$$\text{DB} = \lambda x.(\text{DB2 } (\lambda y.y))x,$$
$$\text{DB2} = \text{rec}\{ f_2 = \lambda zw.\text{if } (\text{null } w) \text{ then } z[\,]$$
$$\text{else } f_2(\lambda xy.z(yx))(\text{tl } w)(\lambda x.[(\text{hd } w).[(\text{hd } w).x]])\}.$$

According to informal meaning, we have DB $[0,1,2] = [0,0,1,1,2,2]$ and (DB2 d)$l = d$(DB l) where d is a dispatcher that takes the value x and several continuations $f_1, \ldots, f_n$ as its arguments and returns $f_n(\ldots(f_1 x)\ldots)$. The exact meaning is given by the following Haskell program.

```
db x = (db2 (\y -> y)) x
db2 :: ([b] -> a) -> [b] -> a
db2 z w = if (null w) then z []
    else db2 (\x y -> z (y x)) (tail w) (\x -> (head w):(head w):x)
```

The term DB2 is not typable in ML since monomorphic recursion is not sufficient for typing it. This is typable in BR in the following way. Let

$$e_2 = \lambda zw.\text{if } (\text{null } w) \text{ then } z[\,] \text{ else } f_2(\lambda xy.z(yx))(\text{tl } w)(\lambda x.[(\text{hd } w).[(\text{hd } w).x]]).$$

We have

$$f_2 : (\beta \text{ list} \to (\beta \text{ list} \to \beta \text{ list}) \to \alpha) \to \beta \text{ list} \to (\beta \text{ list} \to \beta \text{ list}) \to \alpha$$
$$\vdash e_2 : (\beta \text{ list} \to \alpha) \to \beta \text{ list} \to \alpha.$$

Hence by (rec) with $s_1(\alpha) = (\beta \text{ list} \to \beta \text{ list}) \to \alpha$ and $s_2(\alpha) = \beta$ list, we have

$$\vdash \text{DB2} : (\beta \text{ list} \to \beta \text{ list}) \to \beta \text{ list} \to \beta \text{ list}$$

and then we also have

$$\vdash \text{DB} : \beta \text{ list} \to \beta \text{ list}.$$

Example 2.3 The following is an example for nesting of bimorphic recursions. This is obtained from DB2 in the previous example by adding some constant dummy task and writing them by mutual recursion. Let

$$e_0 = [0,1,2],$$
$$e_3 = \lambda zw.\text{if } (\text{null } w) \text{ then } (\lambda x.z[\,])(f_4(\lambda x.x)e_0)$$
$$\text{else } f_3(\lambda xy.z(yx))(\text{tl } w)(\lambda x.[(\text{hd } w).[(\text{hd } w).x]]).$$

We want to define functions by the following mutual recursion.

$$\text{DB3} = e_3[f_3 := \text{DB3}, f_4 := \text{DB4}],$$
$$\text{DB4} = \text{DB3}.$$

The functions DB3 and DB4 behave in the same way as DB2 except that the additional task $f_4(\lambda x.x)e_0$ calculates the doubled list of the fixed list e_0, and its resulting value is thrown away. We can actually define these functions by using nests of bimorphic recursions as follows:

$$\begin{array}{l}\mathrm{DB4} = \mathrm{rec}\{f_4 = \mathrm{rec}\{f_3 = e_3\}\},\\ \mathrm{DB3} = \mathrm{rec}\{f_3 = e_3[f_4 := \mathrm{DB4}]\}.\end{array}$$

Since the body of each recursion has only one recursive call, they are bimorphic recursion. This kind of patterns cannot simulate full polymorphic recursion because of the variable side condition.

Note that the following DB3′ does not work, since the recursive call f_3 occurs twice with different types in the body and it is not bimorphic recursion.

$$\mathrm{DB3}' = \mathrm{rec}\{f_3 = e_3[f_4 := f_3]\}.$$

We also note that the following DB3″ does not work, since the recursive call f_3 occurs twice with different types in the body and it is not bimorphic recursion.

$$\mathrm{DB3}'' = \mathrm{rec}\{f_3 = e_3[f_4 := \mathrm{rec}\{f_4 = f_3\}]\}.$$

We can type DB3 and DB4 in our system in the following way. We have

$$\begin{array}{l}f_4 : (\mathrm{int\ list} \to \mathrm{int\ list}) \to \mathrm{int\ list} \to \mathrm{int\ list},\\ f_3 : (\beta\ \mathrm{list} \to (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to \alpha) \to \beta\ \mathrm{list} \to (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to \alpha\\ \quad \vdash e_3 : (\beta\ \mathrm{list} \to \alpha) \to \beta\ \mathrm{list} \to \alpha.\end{array}$$

By (rec) rule, we have

$$\begin{array}{l}f_4 : (\mathrm{int\ list} \to \mathrm{int\ list}) \to \mathrm{int\ list} \to \mathrm{int\ list}\\ \quad \vdash \mathrm{rec}\{f_3 = e_3\} : (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to (\beta\ \mathrm{list} \to \beta\ \mathrm{list}).\end{array}$$

By (rec), we have

$$\vdash \mathrm{DB4} : (\mathrm{int\ list} \to \mathrm{int\ list}) \to \mathrm{int\ list} \to \mathrm{int\ list}.$$

We also have

$$\begin{array}{l}f_3 : (\beta\ \mathrm{list} \to (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to \alpha) \to \beta\ \mathrm{list} \to (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to \alpha\\ \quad \vdash e_3[f_4 := \mathrm{DB4}] : (\beta\ \mathrm{list} \to \alpha) \to \beta\ \mathrm{list} \to \alpha\end{array}$$

and by (rec) we have

$$\vdash \mathrm{DB3} : (\beta\ \mathrm{list} \to \beta\ \mathrm{list}) \to \beta\ \mathrm{list} \to \beta\ \mathrm{list}.$$

3 Type Inference Algorithm

This section gives our type inference algorithm for bimorphic recursion, and proves its correctness.

A principal typing is defined as a judgment $U \vdash e : u$ when (1) $U \vdash e : u$ is provable, and (2) if $U' \vdash e : u'$ is provable, then there is some substitution s such that $s(U) = U'$ and $s(u) = u'$.

Given types u, v, we write $u = v$ to mean u is v. We write $u \leq v$ to mean there is some substitution s such that $s(u) = v$. A unification problem is defined as a set of equations of the form $u = v$. We say a substitution s is a unifier of the unification problem $\{u_1 = u_1', \ldots, u_n = u_n'\}$ when $s(u_1) = s(u_1'), \ldots, s(u_n) = s(u_n')$ hold. A semi-unification problem is defined as a set of equations of the

forms $u = v$ and inequations of the form $u \leq v$. We say a substitution s is a semiunifier of the semiunification problem $\{u_1 = u'_1, \ldots, u_n = u'_n, v_1 \leq v'_1, \ldots, v_m \leq v'_m\}$ when $s(u_1) = s(u'_1), \ldots, s(u_n) = s(u'_n), s(v_1) \leq s(v'_1), \ldots, s(v_m) \leq s(v'_m)$ hold. A typing problem is defined as the judgment $x_1 : u_1, \ldots, x_n : u_n \vdash e : u$. We say a substitution s is a solution of the typing problem $x_1 : u_1, \ldots, x_n : u_n \vdash e : u$ when $x_1 : s(u_1), \ldots, x_n : s(u_n) \vdash e : s(u)$ is provable.

We will write $s(U)$ for $\{x_1 : s(u_1), \ldots, x_n : s(u_n)\}$ when U is $\{x_1 : u_1, \ldots, x_n : u_n\}$. We will use vector notation $\vec{u}$ for a sequence $u_1, \ldots, u_n$. We will sometimes denote the set $\{u_1, \ldots, u_n\}$ by $\vec{u}$.

When we take Henglein's algorithm [8, 9] for our system, we have the following algorithm E' that produces a semi-unification problem from a given judgment such that $E'(U \vdash e : u) = E_0$ and s is a most general unifier of E_0 if and only if $s(U) \vdash e : s(u)$ is a principal typing.

$$
\begin{array}{l}
E'(U, x : u \vdash x : v) = \{u = v\}, \\
E'(U \vdash x : u) = \{\mathrm{bool} = \mathrm{int}\} \text{ where } x \notin \mathrm{Dom}(U), \\
E'(U \vdash c : u) = \{u = v\} \text{ where } \mathrm{type}(c) = v, \\
E'(U \vdash \lambda x.e_1 : u) = E'(U, x : \alpha \vdash e_1 : \beta) \cup \{\alpha \to \beta = u\} \text{ where} \\
\quad \alpha, \beta \text{ fresh type variables}, \\
E'(U \vdash e_1 e_2 : u) = E'(U \vdash e_1 : \alpha \to u) \cup E'(U \vdash e_2 : \alpha) \text{ where} \\
\quad \alpha \text{ a fresh type variable}, \\
E'(U \vdash \mathrm{rec}\{x = e_1\} : u) = E_1 \cup \{\alpha \times \vec{u} \leq \beta \times \vec{u}, \alpha \times \vec{u} \leq u \times \vec{u}\} \\
\quad \text{where } \alpha, \beta \text{ fresh type variables}, \\
\quad U = \{x_1 : u_1, \ldots, x_n : u_n\}, \\
\quad \vec{u} = u_1 \times \ldots \times u_n, \\
\quad E'(U, x : \beta \vdash e_1 : \alpha) = E_1.
\end{array}
$$

Note that the type inference for the rec construct produces inequations.

This does not give a decidable type inference for our system, since it may produce a semi-unification problem with two or more inequations. For example, when we apply it to the function DB4 in Example 2.3, we have four inequations in $E'(\vdash \mathrm{DB4} : \alpha)$. By some property of the semi-unification problem, we can eliminate two inequations. Then the semi-unification problem becomes

$$\{\alpha_1 \leq \beta_1, \alpha_2 \times \beta_1 \leq \beta_2 \times \beta_1\} \cup E_0$$

where E_0 is some set of equations. In general, semi-unification problems with two inequations are undecidable [8]. Our key idea is that for bimorphic recursion, we can always divide the semi-unification problem $\{\alpha_1 \leq \beta_1, \alpha_2 \times \beta_1 \leq \beta_2 \times \beta_1\} \cup E_0$ into two problems $\{\alpha_1 \leq \beta_1\} \cup E_1$ and $\{\alpha_2 \times \beta_1 \leq \beta_2 \times \beta_1\} \cup E_2$ where E_1 and E_2 are some sets of equations such that in order to solve $\{\alpha_1 \leq \beta_1, \alpha_2 \times \beta_1 \leq \beta_2 \times \beta_1\} \cup E_0$ it is sufficient to first solve $\{\alpha_2 \times \beta_1 \leq \beta_2 \times \beta_1\} \cup E_2$ and then solve $\{\alpha_1 \leq \beta_1\} \cup E_1$. This idea reduces those semi-unification problems into semi-unification problems with a single inequation and gives an algorithm of solving them, since semi-unification problems with a single inequation are decidable [8].

This idea is based on the observation that bimorphic recursion can be typed locally. We explain this observation. We first tried to find an algorithm like E' which behaves as follows: in the same way as type inference algorithms for simply typed lambda calculus, when a term e, a type u, and a type environment U are given, the algorithm chases the proof of $U \vdash e : u$ upward from the conclusion, and produces a set of equations between types such that the existence of its unifier s is equivalent to the provability of $s(U) \vdash e : s(u)$. Then we had difficulty for the (rec) rule.

The idea is that we follow the above algorithm but we handle the (rec) rule in a separate way. First we choose an uppermost (rec) rule in the proof:

$$\dfrac{\begin{array}{c}\vdots\ \pi_1\\ U,x:s_1(u)\vdash e:u\end{array}}{\begin{array}{c}U\vdash \mathrm{rec}\{x=e\}:s_2(u)\\ \vdots\ \pi_2\end{array}}\ (rec)$$

Let V be $\mathrm{FTV}(u)-\mathrm{FTV}(U)$. We will use π_3 to denote the subproof with the (rec) rule and π_1. The subproof π_2 cannot access V because s_2 hides V and U does not have any information of V. Hence type inference for π_3 can be done separately from π_2. Since π_3 has only one (rec) rule, the type inference for π_3 is reduced to a semi-unification problem with a single inequation. Hence type inference for π_3 is possible since there is an algorithm solving a semi-unification problem with a single inequation. By this, we will have a most general semiunifier s and a principal type v of $\mathrm{rec}\{x=e\}$. Then our type inference is reduced to type inference of the proof π_4:

$$\dfrac{\begin{array}{c}\\ s(U)\vdash \mathrm{rec}\{x=e\}:s_1(v)\\ \vdots\ s(\pi_2)\end{array}}{}\ (axiom)$$

for some s_1 where $s(\pi_2)$ denotes the proof obtained from π_2 by replacing every judgment $U_1\vdash e_1:u_1$ by $s(U_1)\vdash e_1:s(u_1)$ and the rule $(axiom)$ denotes a temporary axiom. Since this reduction eliminates one (rec) rule, by repeating this reduction, we can reduce our type inference problem to type inference problem for some term without the (rec) rule. Hence we can complete type inference by solving it with the type inference algorithm for the simply typed lambda calculus.

We will write $\mathrm{FTV}(a_1,\ldots,a_n)$ for $\mathrm{FTV}(a_1)\cup\ldots\cup\mathrm{FTV}(a_n)$ when a_i is a type or a type environment. We define $\mathrm{FTV}(\{u_1=u_1',\ldots,u_n=u_n'\})$ as $\mathrm{FTV}(u_1,u_1',\ldots,u_n,u_n')$.

For a substitution s, a type variable α, and a type u, the substitution $s[\alpha:=u]$ is defined by $s[\alpha:=u](\alpha)=u$, and $s[\alpha:=u](\beta)=s(\beta)$ if $\beta\neq\alpha$.

For a substitution s and a set V of type variables, the substitution $s|_V$ is defined by $s|_V(\alpha)=s(\alpha)$ if $\alpha\in V$ and $s|_V(\alpha)=\alpha$ if $\alpha\notin V$.

For substitutions s_1,s_2, the substitution s_1s_2 is defined by $s_1s_2(\alpha)=s_1(s_2(\alpha))$.

For substitutions s_1,s_2 and a set V of type variables, $s_1=_V s_2$ is defined to hold if $s_1(\alpha)=s_2(\alpha)$ for $\alpha\in V$. We will say $s_1=s_2$ for V when $s_1=_V s_2$.

We will write 1 for the identity substitution, that is, $1(\alpha)=\alpha$.

For a semiunification problem S, we say s is a most general semiunifier of S when s is a semiunifier of S and for every semiunifier r of S there is a substitution r' such that $r's=r$.

We will write mgu for the algorithm that returns a most general semiunifier for a semiunification problem with a single inequation, that is, $\mathrm{mgu}(\{u_1=u_1',\ldots,u_n=u_n',v\leq v'\})=s$ if s is a most general semiunifier of the semiunification problem $\{u_1=u_1',\ldots,u_n=u_n',v\leq v'\}$, and $\mathrm{mgu}(\{u_1=u_1',\ldots,u_n=u_n',v\leq v'\})=$ fail if no semiunifier exists for the semiunification problem $\{u_1=u_1',\ldots,u_n=u_n',v\leq v'\}$. We assume mgu uses fresh variables.

Definition 3.1 (Type Inference Algorithm) In Figure 2, we define an algorithm E that takes a typing problem as its input and returns a pair of a unification problem and a substitution as its outputs. That is, $E(U\vdash e:u)=(E_0,s_0)$ where E_0 is a unification problem. The algorithm E assumes fresh variables. Fresh variables are maintained globally and $E(U\vdash e:u)$ may return an answer with different fresh variables depending on its global context.

$$
\begin{array}{l}
E(U,x:u\vdash x:v)=(\{u=v\},1),\\
E(U\vdash x:u)=(\{\mathrm{bool}=\mathrm{int}\},1)\text{ where }x\notin \mathrm{Dom}(U),\\
E(U\vdash c:u)=(\{u=v\},1)\text{ where }\mathrm{type}(c)=v,\\
E(U\vdash \lambda x.e_1:u)=(E_1\cup\{s_1(\alpha\to\beta)=s_1(u)\},s_1)\text{ where}\\
\quad \alpha,\beta\text{ fresh type variables},\\
\quad E(U,x:\alpha\vdash e_1:\beta)=(E_1,s_1),\\
E(U\vdash e_1e_2:u)=(s_2(E_1)\cup E_2,s_2s_1)\text{ where}\\
\quad \alpha\text{ a fresh type variable},\\
\quad E(U\vdash e_1:\alpha\to u)=(E_1,s_1),\\
\quad E(s_1(U)\vdash e_2:s_1(\alpha))=(E_2,s_2),\\
E(U\vdash \mathrm{rec}\{x=e_1\}:u)=(\{s_2s_1(u)=s_2s_1(\alpha)\},s_2s_1)\text{ where}\\
\quad \alpha,\beta\text{ fresh type variables},\\
\quad E(U,x:\beta\vdash e_1:\alpha)=(E_1,s_1),\\
\quad U=\{x_1:u_1,\ldots,x_n:u_n\},\\
\quad \vec{u}=u_1\times\ldots\times u_n,\\
\quad s_2=\mathrm{mgu}(E_1\cup\{s_1(\alpha\times\vec{u})\le s_1(\beta\times\vec{u})\}),\\
E(U\vdash \mathrm{rec}\{x=e_1\}:u)=(\{\mathrm{bool}=\mathrm{int}\},1)\text{ where}\\
\quad \alpha,\beta\text{ fresh type variables},\\
\quad E(U,x:\beta\vdash e_1:\alpha)=(E_1,s_1),\\
\quad U=\{x_1:u_1,\ldots,x_n:u_n\},\\
\quad \vec{u}=u_1\times\ldots\times u_n,\\
\quad \mathrm{mgu}(E_1\cup\{s_1(\alpha\times\vec{u})\le s_1(\beta\times\vec{u})\})=\mathrm{fail}.
\end{array}
$$

Figure 2: Type Inference Algorithm

When $E(U\vdash e:u)=(E_0,s_0)$, the typing problem $U\vdash e:u$ is reduced to the unification problem E_0. The substitution s_0 gives a partial solution of the typing problem, that is, with a unifier s of E_0, the typing problem $U\vdash e:u$ has a solution ss_0.

Proposition 3.2 *If $U\vdash e:u$ is provable, then $s(U)\vdash e:s(u)$ is provable for any s.*

Proof. By induction on the proof. We consider cases according to the last rule.
Case (var). Suppose $U,x:u\vdash x:u$. We have $s(U),x:s(u)\vdash x:s(u)$.
Case (con). Suppose $U\vdash c:s_1(u)$. We have $s(U)\vdash c:ss_1(u)$.
Case $(\to I)$. Suppose

$$\dfrac{\begin{array}{c}\vdots\\ U,x:u\vdash e:v\end{array}}{U\vdash \lambda x.e:u\to v}$$

By IH, we have $s(U),x:s(u)\vdash e:s(v)$. Hence we have the claim.
Case $(\to E)$ is similar to Case $(\to I)$.
Case (rec). Suppose

$$\dfrac{\begin{array}{c}\vdots\\ U,x:s_1(u)\vdash e:u\end{array}}{U\vdash \mathrm{rec}\{x=e\}:s_2(u)}$$

Let $\vec{\alpha}$ be $\mathrm{FTV}(u) - \mathrm{FTV}(U)$. We can assume $\vec{\alpha}$ are fresh. Hence we have $\mathrm{Dom}(s) \cap \vec{\alpha} = s(U) \cap \vec{\alpha} = \phi$.

Let $s'_1 = (ss_1)|_{\vec{\alpha}}$ and $s'_2 = (ss_2)|_{\vec{\alpha}}$. We have $ss_1(u) = s'_1 s(u)$ since $ss_1(\alpha_i) = s'_1(\alpha_i) = s'_1 s(\alpha_i)$ and $ss_1(\beta) = s(\beta) = s'_1 s(\beta)$ for $\beta \in \mathrm{FTV}(U)$. Similarly we have $ss_2(u) = s'_2 s(u)$.

By IH, we have $s(U), x : ss_1(u) \vdash e : s(u)$. Hence we have $s(U), x : s'_1 s(u) \vdash e : s(u)$. We have $\mathrm{Dom}(s'_1) \subseteq \mathrm{FTV}(s(u)) - \mathrm{FTV}(s(U))$ since $\mathrm{Dom}(s'_1) \subseteq \vec{\alpha}$ holds and $\alpha_i \in \mathrm{FTV}(s(u))$ and $\alpha_i \notin \mathrm{FTV}(s(U))$ show $\vec{\alpha} \subseteq \mathrm{FTV}(s(u)) - \mathrm{FTV}(s(U))$. Similarly we have $\mathrm{Dom}(s'_2) \subseteq \mathrm{FTV}(s(u)) - \mathrm{FTV}(s(U))$. By the rule (rec), we have $s(U) \vdash \mathrm{rec}\{x = e\} : s'_2 s(u)$. Hence we have the claim. □

We define $s(\{\alpha_1, \ldots, \alpha_n\})$ as $\{s(\alpha_1), \ldots, s(\alpha_n)\}$.

Theorem 3.3 *If the typing problem $U \vdash e : u$ has a solution s, V is a finite set of type variables, $FTV(U) \cup FTV(u) \subseteq V$, and $E(U \vdash e : u) = (E_0, s_0)$, then there is a unifier s'_0 of E_0 such that $s'_0 s_0 =_V s$.*

Proof. By induction on e. We consider cases according to e.

Case $e = x$. We can suppose U is $U_1, x : v$. We have $s(v) = s(u)$ and $(E_0, s_0) = (\{v = u\}, 1)$. We can take $s'_0 = s$.

Case $e = c$. Let $\mathrm{type}(c) = v$. We have $s_1(v) = s(u)$ for some s_1. We can assume $\mathrm{FTV}(v)$ is fresh. Then we have $\mathrm{FTV}(v) \cap V = \phi$. We have $(E_0, s_0) = (\{u = v\}, 1)$. We can define s'_0 by $s'_0(\alpha) = s(\alpha)$ for $\alpha \in V$ and $s'_0(\alpha) = s_1(\alpha)$ for $\alpha \in \mathrm{FTV}(v)$. s'_0 is a unifier of E_0 since $s'_0(v) = s_1(v)$ and $s'_0(u) = s(u)$. $s'_0 s_0 =_V s$ since $s'_0 =_V s$.

Case $e = \lambda x.e_1$. We suppose $s(U), x : v_1 \vdash e_1 : v_2$ and $v_1 \to v_2 = s(u)$. Let $\widetilde{s} = s[\alpha := v_1, \beta := v_2]$. The typing problem $U, x : \alpha \vdash e_1 : \beta$ has a solution $\widetilde{s}$. Let $V_1 = V \cup \{\alpha, \beta\}$. By induction hypothesis for e_1 with V_1, there is a unifier s'_1 of E_1 such that $s'_1 s_1 =_{V_1} \widetilde{s}$. We can take $s'_0 = s'_1$. s'_0 is a unifier of E_0 since s'_1 is a unifier of E_1, and $s'_1 s_1 = \widetilde{s}$ for α, β, u. $s'_0 s_0 =_V s$ since $s'_1 s_1 = \widetilde{s} = s$ for V.

Case $e = e_1 e_2$. We suppose $s(U) \vdash e_1 : v \to s(u)$ and $s(U) \vdash e_2 : v$. Let $\widetilde{s} = s[\alpha := v]$. The typing problem $U \vdash e_1 : \alpha \to u$ has a solution $\widetilde{s}$. Let $V_1 = V \cup \{\alpha\}$. By induction hypothesis for e_1 with V_1, we have a unifier s'_1 of E_1 such that $s'_1 s_1 =_{V_1} \widetilde{s}$. We have $\widetilde{s}(U) \vdash e_2 : \widetilde{s}(\alpha)$. Hence the typing problem $s_1(U) \vdash e_2 : s_1(\alpha)$ has a solution s'_1. Let $V_2 = s_1(V_1) \cup \mathrm{FTV}(E_1)$. By induction hypothesis for e_2 with V_2, E_2 has a unifier s'_2 and $s'_2 s_2 =_{V_2} s'_1$. We can take $s'_0 = s'_2$. s'_0 is a unifier of E_0 since s'_2 is a unifier of E_2, and $s'_2 s_2$ is a unifier of E_1 by $s'_2 s_2 = s'_1$ for $\mathrm{FTV}(E_1)$. $s'_0 s_0 =_V s$ since $s = \widetilde{s} = s'_1 s_1 = s'_2 s_2 s_1$ for V.

Case $e = \mathrm{rec}\{x = e_1\}$. We suppose $s(U), x : r_1(v) \vdash e_1 : v$ and $s(u) = r_2(v)$ where $\mathrm{Dom}(r_1), \mathrm{Dom}(r_2) \subseteq \mathrm{FTV}(v) - \mathrm{FTV}(s(U))$. We can assume $\mathrm{FTV}(v) - \mathrm{FTV}(s(U))$ is fresh and $(\mathrm{FTV}(v) - \mathrm{FTV}(s(U))) \cap \mathrm{FTV}(s(V)) = \phi$. Hence we have $r_1 s =_V s$ and $r_2 s =_V s$. Let $\widetilde{s} = s[\alpha := v, \beta := r_1(v)]$. We have $\widetilde{s}(U), x : \widetilde{s}(\beta) \vdash e : \widetilde{s}(\alpha)$. Hence the typing problem $U, x : \beta \vdash e : \alpha$ has a solution $\widetilde{s}$. Let V_1 be $V \cup \{\alpha, \beta\}$. By induction hypothesis for e_1 with V_1, we have a unifier s'_1 of E_1 such that $s'_1 s_1 =_{V_1} \widetilde{s}$. Then s'_1 is a semiunifier of $E_1 \cup \{s_1(\alpha \times \vec{u}) \leq s_1(\beta \times \vec{u})\}$, since $r_1 s'_1 s_1(\alpha \times \vec{u}) = s'_1 s_1(\beta \times \vec{u})$, which is proved as follows: $r_1 s'_1 s_1(\alpha) = s'_1 s_1(\beta)$ since $s'_1 s_1 = \widetilde{s}$ for α, β. $r_1 s'_1 s_1(\vec{u}) = s'_1 s_1(\vec{u})$ since $s'_1 s_1(\vec{u}) = \widetilde{s}(\vec{u}) = s(\vec{u})$ and $r_1 s(\vec{u}) = s(\vec{u})$.

Since s_2 is a most general semiunifier, we have s'_2 such that $s'_2 s_2 = s'_1$. We can take $s'_0 = r_2 s'_2$. $s'_0 s_2 s_1(u) = s'_0 s_2 s_1(\alpha)$ since $r_2 s'_2 s_2 s_1 = r_2 s'_1 s_1 = r_2 \widetilde{s}$ for α and $\mathrm{FTV}(u)$, $r_2 \widetilde{s}(u) = r_2 s(u) = s(u)$, and $r_2 \widetilde{s}(\alpha) = r_2(v) = s(u)$. $s'_0 s_0 =_V s$ since $r_2 s'_2 s_2 s_1 = r_2 s'_1 s_1 =_V r_2 \widetilde{s} =_V r_2 s =_V s$. □

Theorem 3.4 *If $E(U \vdash e : u) = (E_0, s_0)$ and s is a unifier of E_0, then ss_0 is a solution of the typing problem $U \vdash e : u$.*

Proof. By induction on e. We consider cases according to e.

Case $e = x$. Since $\{\mathrm{bool} = \mathrm{int}\}$ does not have any unifier, we have $x \in \mathrm{Dom}(U)$. We suppose U is $U_1, x : u$. We have $(E_0, s_0) = (\{u = v\}, 1)$ and $s(u) = s(v)$. By the rule (var), we have $s(U_1), x : s(u) \vdash x : s(v)$ and ss_0 is a solution.

Case $e = c$. We suppose $\mathrm{type}(c) = v$. We have $(E_0, s_0) = (\{u = v\}, 1)$ and $s(u) = s(v)$. By the rule (con), we have $s(U) \vdash c : s(u)$ and ss_0 is a solution.

Case $e = \lambda x.e_1$. Since s is a unifier of E_1, by induction hypothesis for e_1, ss_1 is a solution of the typing problem $U, x : \alpha \vdash e_1 : \beta$. Hence $ss_1(U), x : ss_1(\alpha) \vdash e_1 : ss_1(\beta)$. By the rule $(\to I)$, we have $ss_1(U) \vdash \lambda x.e_1 : ss_1(\alpha \to \beta)$. Since s is a unifier of $\{s_1(\alpha \to \beta) = s_1(u)\}$, we have $ss_1(\alpha \to \beta) = ss_1(u)$. Hence we have $ss_1(U) \vdash e : ss_1(u)$. Therefore ss_0 is a solution of the typing problem $U \vdash e : u$.

Case $e = e_1 e_2$. Since s is a unifier of E_2, by induction hypothesis for e_2, ss_2 is a solution of the typing problem $s_1(U) \vdash e_2 : s_1(\alpha)$. Then $ss_2 s_1(U) \vdash e_2 : ss_2 s_1(\alpha)$. Since ss_2 is a unifier of E_1, by induction hypothesis for e_1, $ss_2 s_1$ is a solution of the typing problem $U \vdash e_1 : \alpha \to u$. Then $ss_2 s_1(U) \vdash e_1 : ss_2 s_1(\alpha) \to ss_2 s_1(u)$. By the rule $(\to E)$, we have $ss_2 s_1(U) \vdash e_1 e_2 : ss_2 s_1(u)$. Therefore ss_0 is a solution of the typing problem $U \vdash e : u$.

Case $e = \mathrm{rec}\{x = e_1\}$. Since $\{\mathrm{bool} = \mathrm{int}\}$ does not have any unifier, we have $s_2 = \mathrm{mgu}(E_1 \cup \{s_1(\alpha \times \vec{u}) \le s_1(\beta \times \vec{u})\})$. Since s_2 is a unifier of E_1, by induction hypothesis for e_1, we have $s_2 s_1(U), x : s_2 s_1(\beta) \vdash e_1 : s_2 s_1(\alpha)$. Since $s_2 s_1(\alpha \times \vec{u}) \le s_2 s_1(\beta \times \vec{u})$ holds, we have s_3 such that $s_3 s_2 s_1(\alpha \times \vec{u}) = s_2 s_1(\beta \times \vec{u})$. We can suppose $\mathrm{Dom}(s_3) \subseteq \mathrm{FTV}(s_2 s_1(\alpha))$. Then we have $s_3 s_2 s_1(\alpha) = s_2 s_1(\beta)$ and $s_3 s_2 s_1(\vec{u}) = s_2 s_1(\vec{u})$. Hence we have $s_2 s_1(U), x : s_3 s_2 s_1(\alpha) \vdash e_1 : s_2 s_1(\alpha)$ and $\mathrm{Dom}(s_3) \subseteq \mathrm{FTV}(s_2 s_1(\alpha)) - \mathrm{FTV}(s_2 s_1(U))$. By the rule (rec), we have $s_2 s_1(U) \vdash \mathrm{rec}\{x = e_1\} : s_2 s_1(\alpha)$. By Proposition 3.2, we have $ss_2 s_1(U) \vdash e : ss_2 s_1(\alpha)$. Since s is a unifier of $\{s_2 s_1(u) = s_2 s_1(\alpha)\}$, we have $ss_2 s_1(u) = ss_2 s_1(\alpha)$. Then $ss_2 s_1(U) \vdash e : ss_2 s_1(u)$. Hence ss_0 is a solution of the typing problem $U \vdash e$. □

Proof of Theorem 2.1. We define the algorithm as follows. Suppose e is given. We will provide its principal type if e has a type and return the fail if e does not have any type. Let α be a fresh type variable. Let $E(\vdash e : \alpha) = (E_0, s_0)$. If E_0 does not have any unifier, we return the fail. Otherwise let s_1 be a most general unifier of E_0. Let u be $s_1 s_0(\alpha)$. We return u.

We will show that if the algorithm fails then e does not has any type. We assume the algorithm fails and $\vdash e : v$. We will show a contradiction. We define r by $r(\alpha) = v$. Then r is a solution of the typing problem $\vdash e : \alpha$. By Theorem 3.3 for $\vdash e : \alpha$ and r, we have a unifier r' of E_0. Hence the algorithm does not fail, which leads to a contradiction.

We will show that if the algorithm returns a type then it is a principal type. Suppose the algorithm returns u. We will show u is a principal type of e. First we will show $\vdash e : u$. By Theorem 3.4 for (E_0, s_0) and s_1, $s_1 s_0$ is a solution of the typing problem $\vdash e : \alpha$. Hence $\vdash e : s_1 s_0(\alpha)$ and $\vdash e : u$. Next we will show $\vdash e : v$ implies $u \le v$. We define r by $r(\alpha) = v$. Then r is a solution of the typing problem $\vdash e : \alpha$. By Theorem 3.3 for $\vdash e : \alpha$ and r with $V = \{\alpha\}$, we have a unifier r' of E_0 such that $r' s_0 =_V r$. Since s_1 is a most general unifier of E_0, we have $s_2 s_1 = r'$ for some s_2. We have $s_2(u) = v$ since $s_2(u) = s_2 s_1 s_0(\alpha) = r' s_0(\alpha) = r(\alpha) = v$. □

4 Bimorphic Recursion and Polymorphic Let

The system BR of bimorphic recursion can be extended with the standard polymorphic let constructor. The resulting system also has principal types and decidable type inference. We will discuss this extension.

We will define the type system BR+let.

The types in BR will be called mono types. Mono types u, v, w are defined by

$u, v, w ::= \alpha | \mathrm{bool} | \mathrm{int} | u \to u | u \times u | u\ \mathrm{list}$.

Type types in BR+let include polymorphic types. Types A, B, C are defined by

$A, B, C ::= u | \forall \alpha.A$.

A type environment U is the set $\{x_1 : A_1, \ldots, x_n : A_n\}$ where $A_i = A_j$ for $x_i = x_j$.

A judgment is of the form $U \vdash e : u$.

A mono type substitution s is a function from type variables to mono types such that $\{\alpha | s(\alpha) \neq \alpha\}$ is finite.

The inference rules are those in BR except that the rule (var) is replaced by the following $(var-P)$, a mono type substitution is used instead of a substitution in the rule (rec), and the following rule (let) is added.

$$\frac{}{U, x : \forall \alpha_1 \ldots \alpha_n . u \vdash x : s(u)} \ (var-P) \quad (\mathrm{Dom}(s) \subseteq \{\alpha_1, \ldots, \alpha_n\})$$

$$\frac{U \vdash e_1 : v \quad U, x : \forall \alpha_1 \ldots \alpha_n . v \vdash e_2 : u}{U \vdash \text{let } x = e_1 \text{ in } e_2 : u} \ (let) (\alpha_1, \ldots, \alpha_n \in \mathrm{FTV}(v) - \mathrm{FTV}(U))$$

Theorem 4.1 *There is a type inference algorithm for the type system BR+let. That is, there is an algorithm such that for a given term it returns its principal type if the term is typable, and it returns the fail if the term is not typable.*

This is proved by extending the type inference procedure E for BR in Section 3 to BR+let by replacing the variable case by

$$E(U, x : \forall \vec{\alpha} . u \vdash x : v) = (\{u[\vec{\alpha} := \vec{\beta}] = v\}, 1) \text{ where}$$
$$\vec{\beta} \text{ fresh type variables},$$

and adding the following let cases:

$$\begin{aligned}
&E(U \vdash \text{let } x = e_1 \text{ in } e_2 : u) = (E_2, s_3 s_2 s_1) \text{ where} \\
&\quad \alpha \text{ a fresh type variable}, \\
&\quad E(U \vdash e_1 : \alpha) = (E_1, s_1), \\
&\quad \mathrm{mgu}(E_1) = s_2, \\
&\quad \vec{\beta} = \mathrm{FTV}(s_2 s_1(\alpha)) - \mathrm{FTV}(s_2 s_1(U)), \\
&\quad E(s_2 s_1(U), x : \forall \vec{\beta} . s_2 s_1(\alpha) \vdash e_2 : s_2 s_1(u)) = (E_2, s_3), \\
&E(U \vdash \text{let } x = e_1 \text{ in } e_2 : u) = (\{\text{bool} = \text{int}\}, 1) \text{ where} \\
&\quad \alpha \text{ a fresh type variable}, \\
&\quad E(U \vdash e_1 : \alpha) = (E_1, s_1), \\
&\quad \mathrm{mgu}(E_1) = \text{fail}.
\end{aligned}$$

5 Bimorphic Recursion with No Instantiation

This section discusses the type system BRNI which is obtained from the type system BR by removing the instantiation property. We will show the type inference for BRNI is undecidable by reducing semi-unification problems to it.

Semiunification terms M, N are defined by $M, N ::= \alpha | M \times M$ where α is a type variable. Note that a semiunification term is a type of BR.

The following fact is well known for semi-unification problems.

Theorem 5.1 ([8]) *The existence of a semiunifier of the set of two inequations is undecidable. That is, there is no algorithm that decides if there is some s such that $s_1(s(M_1)) = s(N_1)$ and $s_2(s(M_2)) = s(N_2)$ for some s_1, s_2 for a given semiunification problem $\{M_1 \leq N_1, M_2 \leq N_2\}$.*

We define the type system BRNI for bimorphic recursion with no instantiation.

Definition 5.2 The system BRNI is defined as the type system obtained from the system BR by replacing the rule (rec) by the rule $(recni)$:

$$\frac{U, x : s_1(u) \vdash e : u}{U \vdash \mathrm{rec}\{x = e\} : u}\ (recni) \qquad (\mathrm{Dom}(s_1) \subseteq \mathrm{FTV}(u) - \mathrm{FTV}(U))$$

This system is an extension of monomorphic recursions where every recursive call has the same type that is an instantiation of the type of the function definition. Since every recursive call has the same type as the type of the function definition in monomorphic recursion, the system BRNI can type more expressions than monomorphic recursion. For example, the function DB2 in Example 2.2 can be typed with $\vdash \mathrm{DB2} : (\beta\ \mathrm{list} \to \alpha) \to \beta\ \mathrm{list} \to \alpha$ in this system.

The difference between (rec) and $(recni)$ is that (rec) has s_2 but $(recni)$ does not have s_2. By $(recni)$, the type of a recursively defined function is always its general type. For this reason, The system types less expressions than our system BR. For example, the function DB in Example 2.2 cannot be typed because we have to instantiate α by $(\beta\ \mathrm{list})$ in the type of DB2 in order to type DB. For the same reason, the system BRNI does not have the instantiation property described by Proposition 3.2.

We define

$$\begin{aligned}
&(e_1, e_2) = \mathrm{pair}\ e_1 e_2,\\
&e.1 = \mathrm{fst}\ e,\\
&e.2 = \mathrm{snd}\ e,\\
&K = \lambda xy.x,\\
&(e_1 \doteq e_2) = \lambda y.(y e_1, y e_2) \qquad (y \notin \mathrm{FTV}(e_1 e_2))
\end{aligned}$$

We suppose variables $z_1, z_2, \ldots$ are chosen for type variables $\alpha_1, \alpha_2, \ldots$. $\widetilde{M}$ is defined by $\widetilde{\alpha_i} = z_i$ and $\widetilde{M_1 \times M_2} = (\widetilde{M_1}, \widetilde{M_2})$.

Note that when $e_1 \doteq e_2$ is typable, the expressions e_1 and e_2 have the same type. The principal type of $\widetilde{M}$ is M. When $\widetilde{M} \doteq \widetilde{N}$ is typable, we can unify M and N.

Lemma 5.3 *(1) $\vec{z} : \vec{u} \vdash \widetilde{M} : s(M)$ where $s(\alpha_i) = u_i$.*
(2) If $U \vdash e : u$, $U \vdash e : v$, and e is defined by $e ::= x | \lambda x.e | ee | (e,e) | e.1 | e.2$, then $u = v$.

Proof. (1) By induction on M.
(2) By induction on e. □

Lemma 5.4 *Let $\vec{\alpha} = FTV(M_1, M_2, M_3, M_4)$ and $\vec{z} = \widetilde{\vec{\alpha}}$. Let*

$$\begin{aligned}
e_1 &= rec\{f = \lambda \vec{z}.K(\widetilde{M_1}, \widetilde{M_2})(\lambda \vec{y}.(f\vec{y}.1 \doteq \widetilde{N_1}))\},\\
e_2 &= rec\{f = \lambda \vec{z}.K(\widetilde{M_1}, \widetilde{M_2})(\lambda \vec{y}.(f\vec{y}.2 \doteq \widetilde{N_2}))\},
\end{aligned}$$

where $\vec{y}$ are fresh variables of the same length as $\vec{z}$. The judgment $\vdash e_1 \doteq e_2 : u$ is provable in BRNI for some u if and only if the semiunification problem $\{M_1 \le N_1, M_2 \le N_2\}$ has a semiunifier.

We explain proof ideas. Since K is the constant function combinator, both e_1 and e_2 are equal to $\lambda \vec{z}.(\widetilde{M_1}, \widetilde{M_2})$. By $e_1 \doteq e_2$, the expressions e_1 and e_2 have the same type. Since f is a recursive call, the type of f in the body of the recursive definition in e_1 is some instantiation of the type of e_1. Hence the type of $f\vec{y}.1$ is some instantiation of the type of $\widetilde{M_1}$. Since $f\vec{y}.1 \doteq \widetilde{N_1}$, the expressions $f\vec{y}.1$ and $\widetilde{N_1}$ is the same type, and therefore the type of $\widetilde{N_1}$ is some instantiation of the type of $\widetilde{M_1}$. For a similar reason, the type of $\widetilde{N_2}$ is some instantiation of the type of $\widetilde{M_2}$.

Proof. Let

$$e_3 = K(\widetilde{M}_1, \widetilde{M}_2)(\lambda\vec{y}.(f\vec{y}.1 \doteq \widetilde{N}_1)),$$
$$e_4 = K(\widetilde{M}_1, \widetilde{M}_2)(\lambda\vec{y}.(f\vec{y}.2 \doteq \widetilde{N}_2)).$$

From the left-hand side to the right-hand side. We suppose $\vdash e_1 \doteq e_2 : u_0$. Then we have $\vdash e_1 : u$ and $\vdash e_2 : u$ for some u.

We have $f : s_1(u) \vdash \lambda\vec{z}.e_3 : u$ for some s_1. Hence $f : s_1(u), \vec{z} : \vec{u} \vdash e_3 : u'$ and $u = \vec{u} \to u'$ for some u' and some $\vec{u}$. Let $s(\alpha_i) = u_i$. Since $\vec{z} : \vec{u} \vdash (\widetilde{M}_1, \widetilde{M}_2) : s(M_1 \times M_2)$ by Lemma 5.3 (1), we have $u' = s(M_1 \times M_2)$ by Lemma 5.3 (2). Hence $u = \vec{u} \to s(M_1 \times M_2)$. Therefore $f : s_1(u), \vec{y} : s_1(\vec{u}) \vdash f\vec{y}.1 : s_1 s(M_1)$. Since $\vec{z} : \vec{u} \vdash \widetilde{N}_1 : s(N_1)$ by Lemma 5.3 (1), we have $s_1 s(M_1) = s(N_1)$ by Lemma 5.3 (2).

Similarly we have $s_2 s(M_2) = s(N_2)$ for some s_2.

Hence the semiunification problem $\{M_1 \leq N_1, M_2 \leq N_2\}$ has a semiunifier s.

From the right-hand side to the left-hand side. We suppose $s_1(s(M_1)) = s(N_1)$ and $s_2(s(M_2)) = s(N_2)$. Let $\vec{u}$ be $s(\vec{\alpha})$, U be $\vec{z} : \vec{u}$, and u be $\vec{u} \to s(M_1 \times M_2)$.

By Lemma 5.3 (1), we have $U \vdash \widetilde{M}_1 : s(M_1)$, $U \vdash \widetilde{M}_2 : s(M_2)$, $U \vdash \widetilde{N}_1 : s(N_1)$, and $U \vdash \widetilde{N}_2 : s(N_2)$.

We have $f : s_1(u), \vec{y} : s_1(\vec{u}) \vdash f\vec{y}.1 : s_1(s(M_1))$. Hence $U, f : s_1(u), \vec{y} : s_1(\vec{u}) \vdash f\vec{y}.1 \doteq \widetilde{N}_1 : v$ for some v. Hence $U, f : s_1(u) \vdash \lambda\vec{y}.(f\vec{y}.1 \doteq \widetilde{N}_1) : s_1(\vec{u}) \to v$. Combining it with $U \vdash (\widetilde{M}_1, \widetilde{M}_2) : s(M_1 \times M_2)$, we have $U, f : s_1(u) \vdash e_3 : s(M_1 \times M_2)$. Hence $f : s_1(u) \vdash \lambda\vec{z}.e_3 : u$. By the (rec) rule, we have $\vdash \mathrm{rec}\{f = \lambda\vec{z}.e_3\} : u$.

Similarly we have $\vdash \mathrm{rec}\{f = \lambda\vec{z}.e_4\} : u$. Hence we have $\vdash e_1 \doteq e_2 : u_0$ for some u_0. □

Theorem 5.5 *The typability in BRNI is undecidable.*

Proof. If it were decidable, by Lemma 5.4, there would be an algorithm solving semiunification problems of the form $\{M_1 \leq N_1, M_2 \leq N_2\}$. Since semiunification problems of the form $\{M_1 \leq N_1, M_2 \leq N_2\}$ are undecidable by Theorem 5.1, the typability in BRNI is undecidable. □

The difference between BR and BRNI comes from the instantiation property. Since the $(recni)$ rule does not have s_2, the system BRNI does not have the instantiation property like Proposition 3.2. So we cannot use the same idea for BRNI since we cannot replace a uppermost $(recni)$ rule by

$$\dfrac{\dfrac{}{s(U) \vdash \mathrm{rec}\{x = e\} : s_1(v)}\ (axiom)}{\vdots\ s(\pi_2)}$$

for some s_1. It is because $s(U) \vdash \mathrm{rec}\{x = e\} : s_1(v)$ may not be provable for some s_1, even if $s(U) \vdash \mathrm{rec}\{x = e\} : v$ is provable.

6 Concluding Remarks

We have proposed the type system BR with bimorphic recursion. Bimorphic recursion is restricted polymorphic recursion such that each recursive call in the body of the function definition has the same type, and recursive definitions can be nested. We have proved that this type system has principal types and decidable type inference. We have also shown that the extension of bimorphic recursion with the let polymorphism also has principal types and decidable type inference.

Trying to show the decidability of the abstract interpretation given in [7] will be a future work. We have shown that the type inference of bimorphic recursion is decidable, and our bimorphic recursion is inspired by [1]. By clarifying the relationship among the abstract interpretation, the type system in [1], and our bimorphic recursion, we could show the decidability of the abstract interpretation.

Characterizing a class of semi-unification problems that correspond to the type inference for our bimorphic recursion will be another future work. We can expect the class will be larger than semi-unification problems with a single inequation. The computational complexity of the class would be another future work.

Acknowledgments

We would like to thank Prof. Fritz Henglein, Prof. Marco Comini, Prof. Stefano Berardi, and Prof. Kazushige Terui for discussions and comments. We would also like to thank the anonymous referees for valuable comments.

References

[1] M. Comini, F. Damiani & S. Vrech (2008): *On Polymorphic Recursion, Type Systems, and Abstract Interpretation. Proceedings of SAS 2008, LNCS* 5079, pp. 144–158, doi:10.1007/978-3-540-69166-2_10.

[2] M. Coppo (1980): *An extended polymorphic type system. Proceedings of MFCS'80, LNCS* 88, pp. 194–204.

[3] P. Cousot (1997): *Types as abstract interpretation. Proceeding of POPL 97*, pp. 316–331, doi:10.1145/263699.263744.

[4] L. Damas & R. Milner (1982): *Principal type schemes for functional programs. Proceedings of POPL 82*, pp. 207–212, doi:10.1145/582153.582176.

[5] F. Damiani (2007): *Rank 2 intersection for recursive definitions. Fundamenta Informaticae* 77(4), pp. 451–488.

[6] R. Gori & G. Levi (2002): *An experiment in type inference and verification by abstract interpretation. Proceedings of VMCAI'02, LNCS* 2294, pp. 225–239.

[7] R. Gori & G. Levi (2003): *Properties of a type abstract interpreter. Proceedings of VMCAI'03, LNCS* 2575, pp. 132–145, doi:10.1007/3-540-36384-X_13.

[8] F. Henglein (1989): *Polymorphic Type Inference and Semi-Unification.* Ph.D. thesis, the state university of New Jersey.

[9] F. Henglein (1993): *Type inference with polymorphic recursion. ACM TOPLAS* 15(2), pp. 253–289.

[10] R. Hindley (1997): *Basic Simple Type Theory*. Cambridge University Press.

[11] T. Jim (1996): *What are principal typings and what are they good for? Proceedings of POPL'96*, pp. 42–53.

[12] A. J. Kfoury & S. M. Pericas-Geertsen (1999): *Type inference for recursive definitions. Proceedings of LICS'99*, pp. 119–128, doi:10.1109/LICS.1999.782600.

[13] A.J. Kfoury, J. Tiuryn, & P. Urzyczyn (1993): *Type Reconstruction in the Presence of Polymorphic Recursion. ACM TOPLAS* 15 (2), pp. 290–311.

[14] L. Meertens (1983): *Incremental polymorphic type checking in B. Proceedings of POPL'83*, pp. 265–275.

[15] B. Monsuez (1992): *Polymorphic typing by abstract interpretation. Theoretical Computer Science* 652, pp. 217–228.

[16] B. Monsuez (1993): *Polymorphic types and widening operators. Proceedings of SAS'93, LNCS* 724, pp. 224–281.

[17] A. Mycroft (1984): *Polymorphic Type Schemes and Recursive Definitions. LNCS* 167, pp. 217–228.

[18] M. Rittri (1995): *Dimension inference under polymorphic recursion. Proceedings of FPCA '95*, pp. 147–159.

[19] T. Terauchi & A. Aiken (2006): *On typability for polymorphic recursive rank-2 intersection types. Proceedings LICS'06*, pp. 111–122, doi:10.1109/LICS.2006.41.