On Typability for Rank-2 Intersection Types with Polymorphic Recursion *

Tachio Terauchi
EECS Department
University of California, Berkeley

Alex Aiken
Computer Science Department
Stanford University

Abstract

We show that typability for a natural form of polymorphic recursive typing for rank-2 intersection types is undecidable. Our proof involves characterizing typability as a context free language (CFL) graph problem, which may be of independent interest, and reduction from the boundedness problem for Turing machines. We also show a property of the type system which, in conjunction with the undecidability result, disproves a misconception about the Milner-Mycroft type system. We also show undecidability of a related program analysis problem.

1 Introduction

Among the interesting aspects of intersection types is the decidability of type inference for any finite rank for the pure λ-calculus (i.e., without recursive definitions) [8, 6], principal typing [5, 17, 8], the rank-2 fragment [9, 4], which is closely related to ML-types, and connections with polyvariant flow analysis [12]. Recursive definitions such as `fix` $x.e$ are important in practice. Indeed, it is difficult to find a real-world programming language without some form of recursive definitions. If x appears more than once in the body of e of the recursive definition, it may be desirable to give an polymorphic type to x, which leads to polymorphic recursive typing. Jim [4] proposed a natural way to use intersection types for this purpose in the rank-2 fragment. He named the type system $\mathbf{I}_2+$**REC-INT**, where $\mathbf{I}_2$ refers to rank-2 intersection types and **REC-INT** is the name of the rule used to type recursive definitions. While it is known that type inference without polymorphic recursion is decidable for any finite rank intersection types [8, 6], the decidability question has been open for $\mathbf{I}_2 +$ **REC-INT**.

$\mathbf{I}_2+$**REC-INT** is not the most powerful polymorphic recursive type system, but it appears to be capable of typing many programming situations requiring polymorphic recursion (for example, see [2] which studies a similar system). To the best of our knowledge, there is no known polymorphic recursive type system with decidable typability that is both sound and more powerful than $\mathbf{I}_2 +$ **REC-INT**. This paper shows that typability for even $\mathbf{I}_2 +$ **REC-INT** is undecidable.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of $\mathbf{I}_2 +$ **REC-INT**. Section 4 gives a novel reduction of a context free language (CFL) graph problem to typability for $\mathbf{I}_2+$**REC-INT**. Section 5 reduces the boundedness problem to the CFL graph problem to complete the proof of undecidability of typability of $\mathbf{I}_2 +$ **REC-INT**. The last two sections show related results that follow from the proof of undecidability. Section 6 shows a property of $\mathbf{I}_2 +$ **REC-INT** that disproves a misconception about the Milner-Mycroft type system. Section 7 proves undecidability of a related program analysis problem. The companion technical report contains the proofs omitted from the conference version [16].

2 Related Work

While being careful to leave the question open, Jim in his original paper [4] considered the possibility of undecidability of $\mathbf{I}_2+$**REC-INT** typability citing the resemblance to the Milner-Mycroft type system [10] whose typability was already known to be undecidable [7, 3]. More recently, Damiani [1] noted that there seems to be no "obvious way" to find a bound on the size of $|I|$ (see the type rule **REC-INT** in Section 3). Our result confirms these suspicions.

Our proof reduces typability to the boundedness problem of Turing machines. The boundedness problem was also used in the undecidability proof of semi-unification [7].

A step in our proof shows an equivalence between unification type constraints and a CFL graph problem that may be of independent interest to researchers interested in relating type-based program analysis to CFL-based program analysis. While it is suspected that many CFL-based program analyses correspond closely to type-based ones, there

*This research was supported in part by NSF Grant No. CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

$$
\begin{array}{lcl}
e & ::= & x \mid e\,e' \mid \lambda x.e \mid \texttt{fix}\ x.e \\
\tau & ::= & \alpha \mid \tau \rightarrow \tau \\
\sigma & ::= & \tau \mid (\bigwedge_{i\in I} \tau_i) \rightarrow \sigma
\end{array}
$$

Figure 1. Terms and types language.

$$
\frac{\Gamma(x) = \bigwedge_{i\in I} \tau_i \qquad j \in I}{\Gamma \vdash x : \tau_j}\ \textbf{VAR}
$$

$$
\frac{\Gamma, x{:}\bigwedge_{i\in I} \tau_i \vdash e : \sigma}{\Gamma \vdash \lambda x.e : (\bigwedge_{i\in I} \tau_i) \rightarrow \sigma}\ \textbf{FUN}
$$

$$
\frac{\Gamma \vdash e : (\bigwedge_{i\in I} \tau_i) \rightarrow \sigma \qquad \forall i \in I.(\Gamma \vdash e' : \tau_i)}{\Gamma \vdash e\,e' : \sigma}\ \textbf{APP}
$$

$$
\frac{\forall i \in I.(\Gamma, x{:}\bigwedge_{j\in I} \tau_j \vdash e : \tau_i) \qquad k \in I}{\Gamma \vdash \texttt{fix}\ x.e : \tau_k}\ \textbf{REC-INT}
$$

Figure 2. $\mathbf{I}_2 +$ REC-INT.

have been few formal results [14, 13]. One benefit of such correspondences is for proving the soundness of a CFL-based program analysis, which is almost never done, by proving the soundness of an equivalent type-based one, which is, in contrast, a common practice.

Our work seems to be the second time CFL graphs have been used to prove an undecidability result in program analysis. Reps proved undecidability of context-sensitive data-dependence analysis via undecidability of a CFL graph reachability problem [15]. However, the proof strategy used in this paper is different from his.

3 $\mathbf{I}_2 +$ REC-INT

Terms and types are defined in Figure 1. Function application $e\,e'$ is left associative, i.e., $e_1\ e_2\ e_3 = (e_1\ e_2)\ e_3$. Binding of variables extends as far to the right as possible. Types consist of rank-0 types τ and rank-2 types σ. (Rank-1 types are of the form $\bigwedge_{i\in I} \tau_i$.) I is a finite non-empty set of indices. Function types are right associative, i.e., $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

The rank-2 intersection type system with recursive definitions, $\mathbf{I}_2 +$ **REC-INT**, is defined in Figure 2. $\mathbf{I}_2 +$ **REC-INT** is similar to the Milner-Mycroft type system, though not exactly equivalent. For example $\texttt{fix}\ x.x\,x$ is typable in the Milner-Mycroft type system but not in $\mathbf{I}_2 +$ **REC-INT**.

$\mathbf{I}_2 +$ **REC-INT** with a monomorphism restriction for **REC-INT**, i.e.,

$$
\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}\ x.e : \tau}\ \textbf{REC-INT'}
$$

$$
\begin{array}{rcl}
\mathit{id} & \equiv & \lambda x.x \\
e; e' & \equiv & (\mathit{id}\ (\lambda x.e'))\ e \\
 & & \text{where } x \notin \mathit{fvars}(e') \\
\texttt{same}(e, e') & \equiv & \mathit{id}\ (\lambda x.x\ e; x\ e') \\
 & & \text{where } x \notin \mathit{fvars}(e) \cup \mathit{fvars}(e') \\
e \times e' & \equiv & \mathit{id}\ (\lambda x.\texttt{same}(x, e); e') \\
 & & \text{where } x \notin \mathit{fvars}(e) \cup \mathit{fvars}(e')
\end{array}
$$

Figure 3. Encoding of $e; e'$, $e \times e'$, and $\texttt{same}(e, e')$.

is closely related to ML-types and the type inference is decidable, in fact, it is DEXP-time complete [4].

The main result of this paper is the undecidability of $\mathbf{I}_2 +$ **REC-INT** typability. Formally, the typability problem of $\mathbf{I}_2 +$ **REC-INT** is defined as follows: Given a closed term e, is e typable, i.e., is there a type derivation $\Gamma \vdash e : \tau$ for some τ and Γ? Note that it is safe to restrict $\mathit{dom}(\Gamma) = \mathit{fvars}(e)$. (Here, $\mathit{fvars}(e)$ denotes the set of free variables of e.) Our proof shows that even when e is restricted to closed terms (i.e., $\mathit{fvars}(e) = \emptyset$), the typability problem is undecidable.

3.1 Example

One might naively think that, at each **REC-INT**, $|I|$ should at most be the number of occurrences of x in e (see Figure 2). Such a bound on $|I|$ would make type inference easy. However, because any computable bound on $|I|$ would imply decidability, the result in this paper shows that there is no computable way to obtain a bound in general.

We define some syntactic shortcuts to show an example where $|I|$ is greater than the number of variable occurrences. Let $e; e'$ be a sequential composition, $e \times e'$ be a pair, and let $\texttt{same}(e, e')$ force the types of e and e' to be equal. Sequential composition associates to the left and has the weakest precedence, e.g., $e_1\ e_2; e_3; e_4 = ((e_1\ e_2); e_3); e_4$. These expressions are encoded as shown in Figure 3. The reason for the use of *id* in the encodings is to force types to be of rank 0. For example, if we want to ensure that e can be typed rank-0, we apply *id* to e to force existence of a sub-derivation where e has a rank-0 type. (See the **APP** rule in Figure 2.) Note that in this encoding, a pair $\tau \times \tau'$ has a function type $\tau \rightarrow \tau'$. While the encoded pair does not have the expected semantics, it has the expected types. Both pair terms and pair types are right associative.

Let e be the following term:

$$
\begin{array}{ll}
\lambda x.\lambda y.\lambda z. & (\lambda u.\lambda v.\lambda w.\texttt{same}(f\ u\ v\ w, x \times y)); \\
 & (\lambda u.\lambda v.\lambda w.\texttt{same}(f\ u\ v\ w, (x \times x) \times y)); \\
 & y \times z
\end{array}
$$

$$\frac{\Gamma(x) = \bigwedge_{i\in I} \tau_i \quad j \in I}{\Gamma \vdash_0 x : \tau_j}\ \textbf{VAR}$$

$$\frac{\Gamma, x{:}\tau \vdash_0 e : \tau'}{\Gamma \vdash_0 \lambda x.e : \tau \rightarrow \tau'}\ \textbf{FUN}$$

$$\frac{\Gamma \vdash_0 e : \tau \rightarrow \tau' \quad \Gamma \vdash_0 e' : \tau}{\Gamma \vdash_0 e\, e' : \tau'}\ \textbf{APP}$$

$$\frac{\forall i \in I.(\Gamma, x{:}\bigwedge_{j\in I} \tau_j \vdash_0 e : \tau_i) \quad k \in I}{\Gamma \vdash_0 \texttt{fix}\ x.e : \tau_k}\ \textbf{REC-INT}$$

Figure 4. I + REC-INT.

We show that $\texttt{fix}\ f.e$ is typable. Let

$$\begin{aligned}
\tau_1 &= \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow (\alpha \times \alpha) \\
\tau_2 &= \alpha \rightarrow \alpha \rightarrow (\alpha \times \alpha) \rightarrow (\alpha \times (\alpha \times \alpha)) \\
\tau_3 &= \alpha \rightarrow (\alpha \times \alpha) \rightarrow \alpha \rightarrow ((\alpha \times \alpha) \times \alpha) \\
\tau_4 &= \alpha \rightarrow (\alpha \times \alpha) \rightarrow (\alpha \times \alpha) \rightarrow ((\alpha \times \alpha) \times (\alpha \times \alpha))
\end{aligned}$$

Note that $f : \tau_1 \wedge \tau_3 \vdash e : \tau_1$ by assigning τ_1 to the first occurrence of f and τ_3 to the second occurrence of f. Similarly,

$$\begin{aligned}
f{:}\tau_1 \wedge \tau_3 &\vdash e : \tau_2 \\
f{:}\tau_2 \wedge \tau_4 &\vdash e : \tau_3 \\
f{:}\tau_2 \wedge \tau_4 &\vdash e : \tau_4
\end{aligned}$$

Therefore,

$$\frac{\forall i \in I.(f{:}\bigwedge_{j\in I} \tau_j \vdash e : \tau_i) \quad k \in I}{\emptyset \vdash \texttt{fix}\ f.e : \tau_k}$$

where $I = \{1, 2, 3, 4\}$.

On the other hand, there is no derivation that can type this term with $|I| < 4$. It is immediately obvious from $x \times y$ and $(x \times x) \times y$ that $f\ u\ v\ w$ must be given the types $\tau_x \times \tau_y$ and $(\tau_x \times \tau_x) \times \tau_y$ for some τ_x, τ_y. But due to $y \times z$, this implies that y must have the types τ_x and $\tau_x \times \tau_x$. Therefore, we actually need two kinds of τ_y's, i.e., τ_x and $\tau_x \times \tau_x$, which implies that there must be at least four types for $f\ u\ v\ w$.

4 Typability as a CFL Graph Problem

For this proof, we introduce the simpler type system **I + REC-INT** shown in Figure 4. In general, typability in **I + REC-INT** does not coincide with typability in $\mathbf{I}_2$ **+ REC-INT** (e.g., $\lambda x.x\, x$). However, we prove that even when restricted to the set of terms that are typable in $\mathbf{I}_2$ **+ REC-INT** iff typable in **I + REC-INT**, the typability problem is undecidable. More generally, let us define the subset of terms B as follows:

$$B \quad ::= \quad x \mid id\,(\lambda x.B) \mid B\, B \mid \texttt{fix}\ x.B$$

$$\begin{aligned}
&[\![\lambda x.e]\!]_\Gamma = (\beta, C \cup \{\beta = \alpha \rightarrow \tau\}, X \cup \{\alpha, \beta\}) \\
&\text{where } \alpha \in \mathit{Base} \setminus (X \cup \mathit{ran}(\Gamma)) \\
&\qquad \beta \in \mathit{Base} \setminus (X \cup \mathit{ran}(\Gamma) \cup \{\alpha\}) \\
&\qquad (\tau, C, X) = [\![e]\!]_{\Gamma, x:\alpha}
\end{aligned}$$

$$\begin{aligned}
&[\![e_1\, e_2]\!]_\Gamma = (\beta, C, X_1 \cup X_2 \cup \{\alpha, \beta\}) \\
&\text{where } \alpha \in \mathit{Base} \setminus (X_1 \cup X_2 \cup \mathit{ran}(\Gamma)) \\
&\qquad \beta \in \mathit{Base} \setminus (X_1 \cup X_2 \cup \mathit{ran}(\Gamma) \cup \{\alpha\}) \\
&\qquad X_1 \cap X_2 = \emptyset \\
&\qquad (\tau_1, C_1, X_1) = [\![e_1]\!]_\Gamma \\
&\qquad (\tau_2, C_2, X_2) = [\![e_2]\!]_\Gamma \\
&\qquad C = C_1 \cup C_2 \cup \{\tau_1 = \alpha \rightarrow \beta, \tau_2 = \alpha\}
\end{aligned}$$

$$\begin{aligned}
&[\![x]\!]_\Gamma = (\Gamma(x), \emptyset, \emptyset) \\
&\text{where } x \in \mathit{dom}(\Gamma)
\end{aligned}$$

$$\begin{aligned}
&[\![\texttt{fix}\ x.e]\!]_\Gamma = (\tau, C, X') \\
&\text{where } \Sigma = \{a \mid x^a \in \mathit{fvars}(e)\} \\
&\qquad X' = X \cup \{\alpha_a \mid a \in \Sigma\} \\
&\qquad \forall a \in \Sigma.\alpha_a \in \mathit{Base} \setminus (\mathit{ran}(\Gamma) \cup X' \setminus \{\alpha_a\}) \\
&\qquad (\tau, C', X) = [\![e]\!]_{\Gamma \cup \bigcup_{a\in\Sigma}\{x^a : \alpha_a\}} \\
&\qquad C'' = \textstyle\bigcup_{a\in\Sigma}\{\tau'^a = \tau' \mid \tau' \in \mathit{ran}(\Gamma)\} \cup \{\tau^a = \alpha_a\} \\
&\qquad C = \textstyle\bigcup_{s\in\Sigma^*}(C' \cup C'')^s
\end{aligned}$$

Figure 5. Constraint generation.

Lemma 4.1 *For all closed B, $\emptyset \vdash_0 B : \sigma$ iff $\emptyset \vdash B : \sigma$.*

Closed B terms do not include all of the terms whose typability in $\mathbf{I}_2$ **+ REC-INT** coincides with typability in **I + REC-INT** but are sufficient for our purpose. In the rest of the paper, typable means typable in $\vdash_0$ and type means rank-0 type unless stated otherwise.

4.1 Type Constraints

As in conventional type inference algorithms, we formulate the typability problem as a constraint satisfaction problem. However, the purpose here is not to solve the constraints but to show its undecidability.

We warn that the phrase "constraint generation" is somewhat misleading because there is no terminating algorithm to generate the constraints. (The set of constraints may be infinite.) When we say that the set of constraints is generated, we mean that the set exists (in standard set theory). Existence is sufficient for our purpose of proving undecidability.

The generated constraint set may contain infinitely many type variables. To this end, we annotate type variables with superscripts. Let Base be the set of type variables without superscripts, or equivalently, with an empty string as the

superscript. Let meta variables α, β, etc. range over type variables with a (possibly empty) superscript. For a type variable α and a string s, α^s is a type variable whose superscript is a concatenation of the superscript of α followed by s. For example, $(\beta^{s_1})^{s_2} = \beta^{s_1 s_2}$. For a type τ and a string s, τ^s is a type obtained by replacing each type variable α in τ by α^s. For example, $(\alpha^{s_1} \rightarrow \beta^{s_1})^{s_2} = \alpha^{s_1 s_2} \rightarrow \beta^{s_1 s_2}$. For a set of type equality constraints C, $C^s = \{(\tau_1^s = \tau_2^s) \mid (\tau_1 = \tau_2) \in C\}$.

We annotate term variables with superscripts so that each occurrence of a `fix`-bound variable is annotated with a distinct number, e.g., $\texttt{fix}\ x.\texttt{fix}\ y.x^0\ \lambda z.x^1\ y^2\ z$. These numbers form the alphabet of the strings annotating the type variables. We use meta variables x, y, etc. to range over variables with a (possibly empty) superscript.

Constraint generation is shown in Figure 5. A mapping Γ from variables to types is a *type environment*. Intuitively, $[\![e]\!]_\Gamma$ returns a triple (τ, C, X) such that τ is the type of e, X is the set of base type variables introduced while analyzing e, and C is the set of constraints generated while analyzing e. The use of set X is a standard technique for avoiding unnecessary introduction of the same type variable in two different contexts. The first three rules are self-explanatory, and coincide with a typical constraint-based type inference algorithm for simply typed λ-calculus.

The fourth rule handles $\texttt{fix}\ x.e$. The goal is to build a constraint set representing the infinite unrolling of the recursive body e. Recall that occurrences of `fix`-bound variables are annotated with distinct numbers. In the rule, Σ is the set of numbers annotating x. Each $a \in \Sigma$ has the associated base type variable α_a. The line $\forall a \in \Sigma.\alpha_a \in \mathit{Base} \setminus (\mathit{ran}(\Gamma) \cup X' \setminus \{\alpha_a\})$ ensures that these variables are distinct. Thus (τ, C', X) is the result of analyzing the body of the recursive definition e by assigning a distinct type variable to each x^a. Intuitively, C' is the template constraint that should be repeated indefinitely, and C contains infinitely many copies of C' distinguished by superscripts. Therefore, C'^{as} represents the constraint of the body e unrolled at x^a appearing in the body e that itself was unrolled from the root according to s. C also contains copies of C'', which is used to connect the copies of C' (note that C'^{s_1} and C'^{s_2} share no type variables when $s_1 \neq s_2$). C'' consists of two parts. The first part, $\bigcup_{a\in\Sigma}\{\tau'^a = \tau' \mid \tau' \in \mathit{ran}(\Gamma)\}$, ensures that free variables in e get the same types in the unrolling.[1] The second part, $\bigcup_{a\in\Sigma}\{\tau^a = \alpha_a\}$, equates the type of x^a (i.e., α_a) with the type of the body e unrolled at x^a (i.e., τ^a).

We connect typability to constraint satisfaction as follows. An assignment S is a mapping from type variables to types. For τ, $S(\tau)$ is the type obtained by replacing each type variable α in τ by $S(\alpha)$. An assignment S is a solution of C, written $S \models C$, if for each $\tau = \tau' \in C$, $S(\tau) = S(\tau')$. We say that S is a *finite-range solution* if the range of S, $\mathit{ran}(S)$, is a finite set. We write $S \models_{\mathit{fin}} C$ if S is finite-range and $S \models C$. We write $\models C$ if C is satisfiable, i.e., if there exists S such that $S \models C$. We write $\models_{\mathit{fin}} C$ if C is finitary-satisfiable, i.e., if there exists S such that $S \models_{\mathit{fin}} C$. A term e is typable iff the constraints generated for e are finitary-satisfiable, i.e.,

Lemma 4.2 *Let e be a closed term. Let e' be e such that each occurrence of a `fix`-bound variable is annotated with a distinct number. Then e is typable in* **I + REC-INT** *iff $\models_{\mathit{fin}} C$ where $(\tau, C, X) = [\![e']\!]_\emptyset$ for some τ and X.*

Example Consider the term $\texttt{fix}\ x.x^0\ x^1$. Then,

$$\begin{array}{l}
[\![x^0]\!]_{x^0:\gamma,x^1:\kappa} = (\gamma, \emptyset, \emptyset) \\
[\![x^1]\!]_{x^0:\gamma,x^1:\kappa} = (\kappa, \emptyset, \emptyset) \\
[\![x^0\ x^1]\!]_{x^0:\gamma,x^1:\kappa} = (\beta, \{\gamma = \alpha\rightarrow\beta, \kappa = \alpha\}, \{\alpha, \beta\}) \\
[\![\texttt{fix}\ x.x^0\ x^1]\!]_\emptyset = (\beta, C, \{\alpha, \beta, \gamma, \kappa\})
\end{array}$$

where

$$C = \bigcup_{s\in\{0,1\}^*} \{\gamma = \alpha\rightarrow\beta, \kappa = \alpha, \beta^0 = \gamma, \beta^1 = \kappa\}^s$$

For any S such that $S \models C$, it must be the case that $S \models C'$ where $C' = \{\beta^{0s} = \beta^{1s}\rightarrow\beta^s \mid s \in \{0,1\}^*\}$. But C' clearly has no finite range solution. Therefore, C has no finite range solution, and $\texttt{fix}\ x.x\ x$ is not typable. (Note that there is an infinite-range solution for C'. However, it is not always the case that an untypable term has an infinite-range solution.)

4.2 Constraints as a CFL Graph

The next step of the proof is to represent constraints as a context free language (CFL) graph. We treat constraints symmetrically, i.e, $\tau = \tau'$ is equivalent to $\tau' = \tau$. Let C be a constraint generated from a closed term e, i.e., $(\tau, C, X) = [\![e]\!]_\emptyset$ for some τ and X. Note that all of the constraints in C are of the form $\alpha = \beta$ or $\alpha = \beta\rightarrow\gamma$. We use the notation $\mathit{ftvars}(\tau)$ to denote the set of types variables in τ. Let $\mathit{ftvars}(C) = \bigcup_{(\tau=\tau')\in C}(\mathit{ftvars}(\tau) \cup \mathit{ftvars}(\tau'))$. The CFL graph of C, written $\mathit{graph}(C)$, is the graph (V, E) where

$$\begin{array}{rcl}
V & = & \mathit{ftvars}(C) \\
E & = & \{\alpha \xrightarrow{\epsilon} \beta \mid (\alpha = \beta) \in C\} \\
 & & \cup\{\alpha \xrightarrow{(} \gamma, \beta \xrightarrow{[} \gamma \mid (\alpha\rightarrow\beta = \gamma) \in C\} \\
 & & \cup\{\gamma \xrightarrow{)} \alpha, \gamma \xrightarrow{]} \beta \mid (\alpha\rightarrow\beta = \gamma) \in C\}
\end{array}$$

For example, let $C = \{\beta_0 = \alpha_1\rightarrow\beta_1, \beta_1 = \alpha_2\rightarrow\beta_2, \beta_0 =$

[1] Technically, this part is inessential as all $\texttt{fix}\ x.e$ used in the rest of the paper are closed. However, it is included here for completeness and to make the proof of Lemma 4.2 succinct.

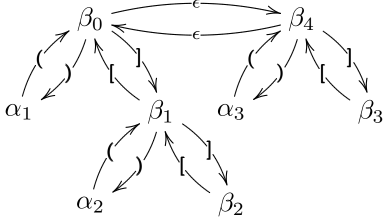

Figure 6. Example.

$$
\begin{array}{lcl}
proj(\tau, \epsilon) & = & \tau \\
proj(\tau \to \tau', \mathbf{)}s) & = & proj(\tau, s) \\
proj(\tau \to \tau', \mathbf{]}s) & = & proj(\tau', s)
\end{array}
$$

Figure 7. Path projection.

$\beta_4, \beta_4 = \alpha_3 \to \beta_3\}$. Then *graph*$(C)$ is as shown in Figure 6.

Given a path p in the graph, let $s(p)$ be the string obtained by concatenating in order the labels of edges in p. Let ϵ denote an empty string. Let $L(A)$ be the set of strings generated by the following grammar:

$$A ::= \epsilon \mid A\ A \mid \mathbf{(}\ A\ \mathbf{)} \mid \mathbf{[}\ A\ \mathbf{]}$$

A *match elimination* $\to_m$ is defined as follows:

$$
\begin{array}{c}
t_1\ \mathbf{(}\,\mathbf{)}\ t_2 \to_m t_1 t_2 \\
t_1\ \mathbf{[}\,\mathbf{]}\ t_2 \to_m t_1 t_2
\end{array}
$$

For a path p, the *match-eliminated string* of p, written *sm*(p), is a $\to_m$-normalized $s(p)$, i.e., a string t such that $s(p) \to_m^* t$ where no substring of t is in $L(A)$. We write $\alpha \overset{t}{\leadsto} \beta$ to denote a path p from α to β such that *sm*$(p) = t$. For example, there is a path $\beta_2 \overset{\mathbf{]}}{\leadsto} \beta_3$ in Figure 6.

We call p a *matched path* if *sm*(p) is an empty string. For convenience, we say that every variable has a (self) matched path to itself, i.e., $\alpha \overset{\epsilon}{\leadsto} \alpha$. We say that a string t is *positive* if t consists only of '**)**' and '**]**'. We call p a *positive path* if it is a matched path or if *sm*(p) is a positive string. The *depth* of a positive path p, *depth*(p), is the length of *sm*(p).

We want to show that $\models_{\mathit{fin}} C$ iff the depth of positive paths in *graph*(C) is bounded. To this end, we relate types to paths as follows. For a positive string t and a type τ, the t-projection of τ, *proj*(τ, t) is defined as shown in Figure 7. Note that *proj*(τ, t) may be undefined. For example, *proj*$(\alpha \to \alpha \to \beta, \mathbf{))})$ is undefined. The following lemma says that positive paths imply type-structural constraints.

Lemma 4.3 *Let p be a path from α to β in graph(C) such that sm(p) is a positive string. Suppose $S \models C$. Then there exists τ such that proj$(\tau, sm(p)) = \beta$ and $S(\alpha) = S(\tau)$.*

We say τ is smaller than τ' if *size*$(\tau) <$ *size*(τ') where *size* is defined

$$
\begin{array}{l}
size(\alpha) = 1 \\
size(\tau \to \tau') = size(\tau) + size(\tau')
\end{array}
$$

For a set X of positive strings and a type variable α, we define *pathsType*(X, α) to be the smallest type τ containing only α such that for each $t \in X$, *proj*(τ, t) is defined (so *proj*$(\tau, t) = \alpha$). For example,

$$pathsType(\{\ \mathbf{)]}),\ \mathbf{]]}\ \}, \alpha) = (\alpha \to (\alpha \to \alpha)) \to \alpha \to \alpha$$

Note that for X finite, *pathsType*(X, α) is always defined. Given a type variable α in a CFL graph G, let *posPaths*(α, G) be the set of all positive paths from α. We are now ready to prove the main result of this section.

Lemma 4.4 $\models_{\mathit{fin}} C$ *iff there exists a positive integer n such that for any positive path p in graph(C), depth$(p) \leq n$.*

Proof:

If

Let n be a positive integer such that for any positive path p in *graph*$(C) = G = (V, E)$, *depth*$(p) \leq n$. Fix a type variable δ. Let F be a mapping from type variables to sets of positive strings such that for each $\alpha \in V$, $F(\alpha) = \{sm(p) \mid p \in posPaths(\alpha, G)\}$. Define S as follows

$$S = \{\alpha \mapsto pathsType(F(\alpha), \delta) \mid \alpha \in V\}$$

Because depths of positive paths are bounded, $F(\alpha)$ must be finite for every α. Hence each *pathsType*$(F(\alpha), \delta)$ is defined, and so S is defined. Furthermore, *ran*(S) is finite, in particular, $|ran(S)| < 2^{n+1}$. Hence it suffices to show that $S \models C$.

Pick $(\alpha = \beta) \in C$. By construction, $\alpha \xrightarrow{\epsilon} \beta$ and $\beta \xrightarrow{\epsilon} \alpha$. Hence $F(\alpha) = F(\beta)$. Therefore $S(\alpha) = S(\beta)$ as required.

Pick $(\alpha = \beta \to \gamma) \in C$. Suppose $t \in F(\beta)$. then there exists a path $\beta \overset{t}{\leadsto} \kappa$ for some κ. By construction, there is an edge $\alpha \xrightarrow{\mathbf{)}} \beta$. Hence there is a path $\alpha \xrightarrow{\mathbf{)}} \beta \overset{t}{\leadsto} \kappa$, and so there is a path p' from α such that $sm(p') = \mathbf{)}t$. Thus, $\mathbf{)}t \in F(\alpha)$. Conversely, suppose $\mathbf{)}t \in F(\alpha)$. Let p be a path from α such that $sm(p) = \mathbf{)}t$. Let β' be a node such that p is $\alpha \overset{\mathbf{)}}{\leadsto} \beta' \overset{t}{\leadsto} \kappa$ where κ is the end vertex of p. By construction, there is an edge $\beta \xrightarrow{\mathbf{(}} \alpha$. Therefore, there is a path $\beta \xrightarrow{\mathbf{(}} \alpha \overset{\mathbf{)}}{\leadsto} \beta' \overset{t}{\leadsto} \gamma$, and so there is a path p' from β such that $sm(p') = t$. Thus, $t \in F(\beta)$.

Hence $t \in F(\beta)$ iff $\mathbf{)}t \in F(\alpha)$. By a similar argument, $t \in F(\gamma)$ iff $\mathbf{]}t \in F(\alpha)$. Therefore $S(\alpha) = S(\beta) \to S(\gamma)$ as required, and $S \models C$.

Only If

Suppose there exists no n such that for any positive path p in *graph*(C), *depth*$(p) \leq n$. For the sake of obtaining a contradiction, suppose there exists S such that $S \models_{\mathit{fin}} C$. Let

m be a number such that for any $\tau \in \mathit{ran}(S)$, $\mathit{size}(\tau) < m$. Pick a path p in *graph*(C) such that *depth*$(p) > m$. Let α be the starting vertex and β be the ending vertex of p. Then by Lemma 4.3, there exists τ such that $\mathit{proj}(\tau, \mathit{sm}(p)) = \beta$ and $S(\alpha) = S(\tau)$. But $|\mathit{sm}(p)| > m$ implies $\mathit{size}(S(\alpha)) = \mathit{size}(S(\tau)) > m$, a contradiction.
□

5 Reduction from the Boundedness Problem

We reduce the *boundedness problem* to the problem of finding a bound on the depth of positive paths in *graph*(C). The boundedness problem is known to be undecidable [7], and hence this reduction shall show that the problem of finding a bound on the depth of positive paths in *graph*(C) is undecidable, which in turn implies the undecidability of typability. Here, we present the boundedness problem as it is defined in [7].

An Intercell Turing Machine (symmetric ITM) is a triple of the form $Y = \langle Q, A, T\rangle$, where

- Q is a finite set of states,
- A is a finite tape alphabet, and
- $T \subseteq Q \times \{-1, +1\} \times A \times A \times Q$ is a transition relation.

An instantaneous description (ID) of Y takes the form $\langle w_1, \alpha, m, w_2\rangle$ where $w_1 w_2$ is the tape content with all but finitely many blank symbols and the head is positioned between the $(m-1)$-th and the m-th cells, which is between w_1 and w_2. [2] The next move relation $\vdash_Y$ on ID's of Y is defined as follows:

$$\begin{array}{l} \text{for } \langle \alpha, -1, a, b, \beta\rangle \in T \\ \quad \langle w_1 a, \alpha, m, w_2\rangle \vdash_Y \langle w_1, \beta, m-1, b w_2\rangle \\ \text{for } \langle \alpha, +1, a, b, \beta\rangle \in T \\ \quad \langle w_1, \alpha, m, a w_2\rangle \vdash_Y \langle w_1 b, \beta, m+1, w_2\rangle \end{array}$$

An ITM Y is *bounded* if there exists a positive integer n such that if M is an arbitrary ID of Y, then the number of different ID's reachable by Y from M is at most n.

Let $Y = \langle Q, A, T\rangle$ be an ITM. The *symmetric closure* of Y is $Y_S = \langle Q, A, T_S\rangle$ where

$$T_S = T \cup \{\langle \alpha, -x, a, b, \beta\rangle \mid \langle \beta, x, b, a, \alpha\rangle \in T\}$$

The *boundedness problem for symmetrically-closed* ITM*s* is the problem of deciding for a given deterministic ITM $Y = \langle Q, \{0, 1\}, T\rangle$ with 0 as the blank symbol, whether Y_S is bounded.

Theorem 5.1 ([7]) *The boundedness problem for symmetrically-closed* ITM*s is undecidable.*

[2] Strictly speaking, m is redundant since w_1 and w_2 precisely determine the location of the head. But m makes the proof more readable.

We now reduce the boundedness problem of symmetrically-closed ITMs to the problem of finding a bound on the depth of positive paths. Our goal is to construct a closed term e_Y for an ITM Y such that Y_S is bounded iff the depth of positive paths in *graph*(C) is bounded where $(\tau, C, X) = [\![e_Y]\!]_\emptyset$ for some X and τ. The idea is that C would look like an infinite binary tree in which each left move of Y_S is represented by a down move in the tree (from a parent to a child), and each right move of Y_S is represented by an up move in the tree (from a child to the parent). The tape content to the right of the head records which branch was taken at each down move. This ensures that up moves use the edges actually belonging to the tree. The tape content to the left of the head records whether a (edge or a [edge is followed at each up move so that a down move must use a) edge to match a (up move and a] edge to match a [up move. Symmetry of ITM is needed in part because our CFL graphs are bi-directional, i.e., $(\alpha \xrightarrow{)} \beta) \in E$ iff $(\beta \xrightarrow{(} \alpha) \in E$, and similarly for], [, and ϵ edges. However, it turns out that the CFL graphs must be bi-directional anyway to simulate a symmetric-or-asymmetric ITM with our proof technique.

Instead of introducing e_Y at this point, it is more helpful to describe the constraint C_Y such that *graph*(C_Y) simulates Y_S in the way described above. We then construct the term e_Y that generates C_Y. Let $Y = \langle Q, \{0, 1\}, T\rangle$ be a deterministic ITM. Let $\{M_1, \ldots, M_n\} \subseteq T_S$ be the set of all left transitions of Y_S. For each $M_\ell \in T_S$, let γ_ℓ be a distinct type variable. For each $M_\ell = \langle \alpha, -1, b, a, \beta\rangle$, define types $\tau_{a,\ell}$ and $\kappa_{a,\ell}$ as follows:

$$\begin{array}{rcl} \tau_{a,\ell} & \equiv & \begin{cases} \beta \times \gamma_\ell & \text{if } b = 0 \\ \gamma_\ell \times \beta & \text{if } b = 1 \end{cases} \\ \kappa_{a,\ell} & \equiv & \alpha \end{array}$$

Note that we have intentionally picked type variable names that correspond to the state names in Y. C_Y is defined as follows:

$$C_Y = \bigcup_{s \in \{0,1\}^*} \{\kappa_{a,\ell} = \tau^a_{a,\ell} \mid a \in \{0, 1\}, \ell \in \{1, \ldots, p\}\}^s$$

As an example, consider $Y = \{\{\alpha_1, \alpha_2, \alpha_3\}, \{0, 1\}, T\}$ where

$$T = \{\langle \alpha_1, -1, 1, 0, \alpha_3\rangle, \langle \alpha_3, -1, 0, 1, \alpha_2\rangle\}$$

Figure 8 shows the subgraph of *graph*(C_Y) for the variables with superscripts s, $0s$, and $1s$. The entire *graph*(C_Y) is infinite. In particular, *graph*(C_Y) can be obtained by repeating the structure in the diagram. That is, there are edges between α_1^{0s} and α_3^{00s}, edges between α_1^{0s} and γ_1^{00s}, edges between α_3^{0s} and α_2^{10s}, and so on. Pictorially, *graph*(C_Y) is an infinite binary tree such that for any s, variables with

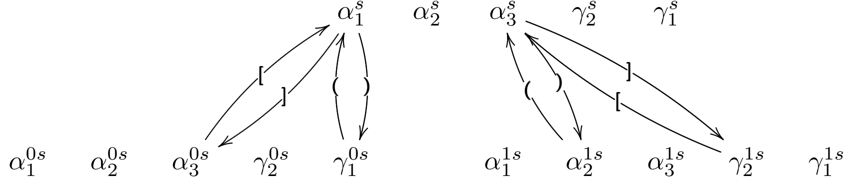

Figure 8. Example.

the superscript s collectively form a node (s-node) with the $0s$-node being the left child and the $1s$-node being the right child. Note that any edge may only connect a variable in a parent node with a variable in its child node.

We now construct the term e_Y. We use the vector notation $\overrightarrow{x}$ to denote a sequence of variables. We write $\lambda\overrightarrow{x}.e$ to mean $id(\lambda x_1.id\,(\lambda x_2.\ldots id\,(\lambda x_n.e)))$ where $\overrightarrow{x} = x_1,\ldots,x_n$. We write $e'\,\overrightarrow{x}$ to mean the sequence of function applications $e'\,x_1\,x_2\ldots x_n$ where $\overrightarrow{x} = x_1,\ldots,x_n$.

For each $\alpha \in Q$, let x_α be a distinct variable. For each γ_ℓ, let y_ℓ be a distinct program variable. For each $M_\ell = \langle\alpha,-1,b,a,\beta\rangle$, define terms $e_{a,\ell}$ and $v_{a,\ell}$ as follows:

$$e_{a,\ell} \equiv \begin{cases} x_\beta \times y_\ell & \text{if } b = 0 \\ y_\ell \times x_\beta & \text{if } b = 1 \end{cases}$$
$$v_{a,\ell} \equiv x_\alpha$$

For each $a \in \{0,1\}$, let $e_a = e_{a,1} \times e_{a,2} \times \ldots \times e_{a,n}$ and $v_a = v_{a,1} \times v_{a,2} \times \ldots \times v_{a,n}$. Recall that $e \times e'$ is defined in Figure 3. Let $X = \mathit{fvars}(e_0) \cup \mathit{fvars}(e_1) \cup \mathit{fvars}(v_0) \cup \mathit{fvars}(v_1)$. Let d_0 and d_1 be distinct variables not in X. Let $\overrightarrow{x}$ be a sequence of variables from $X \cup \{d_0,d_1\}$. Let $\overrightarrow{z}$ be distinct variables not in $\overrightarrow{x}$ such that $|\overrightarrow{z}| = |\overrightarrow{x}|$. Let

$$\begin{aligned} e_Y = {} & \texttt{fix}\ f.\lambda\overrightarrow{x}. \\ & (\lambda\overrightarrow{z}.\texttt{same}(f^0\ \overrightarrow{z}, v_0 \times d_0)); \\ & (\lambda\overrightarrow{z}.\texttt{same}(f^1\ \overrightarrow{z}, d_1 \times v_1)); \\ & (e_0 \times e_1) \end{aligned}$$

Recall that $e;e'$ and $\texttt{same}(e,e')$ are defined in Figure 3. Let $(\tau,C,X) = [\![e_Y]\!]_\emptyset$. C is not exactly C_Y, but simple algebraic manipulation shows that $\models_{\mathit{fin}} C$ iff $\models_{\mathit{fin}} C_Y$.

We now show that Y_S is bounded iff positive paths in $\mathit{graph}(C_Y)$ have a bounded depth. For a positive string t, let $r(t)$ be reverse of t with) replaced by 0 and] replaced by 1. For an infinitely long sequence w, let $w|_n$ be the string consisting of the first n symbols of w. For a string $s \in \{0,1\}^*$, let $s0^\infty$ be an infinitely long sequence w such that $w|_{|s|} = s$ and the ith symbol of w is 0 for all $i > |s|$. For clarity, we sometimes write $s_1@s_2$ to mean the concatenation s_1s_2. We show that if Y_S is bounded then positive paths in $\mathit{graph}(C_Y)$ have a bounded depth.

Lemma 5.2 *Let p be a path from α^{s_1} to β^{s_2} in $\mathit{graph}(C_Y)$ such that $\mathit{sm}(p)$ is a positive string. Then $\langle w@r(\mathit{sm}(p)),\alpha,m,s_10^\infty\rangle$ and $\langle w,\beta,m-|\mathit{sm}(p)|,s_20^\infty\rangle$ are reachable from each other in Y_S.*

We now prove the other direction, i.e., if positive paths in $\mathit{graph}(C_Y)$ have a bounded depth then Y_S is bounded.

Lemma 5.3 *Suppose $\langle w@r(t_1),\alpha,m,s_10^\infty\rangle$ and $\langle w@r(t_2),\beta,m-|t_1|+|t_2|,s_20^\infty\rangle$ are reachable from each other in Y_S without moving the head below the position $m-|t_1|$ and without moving the head above the position $m+|s_1|$. Further suppose $|t_1|+|s_1| = |t_2|+|s_2|$. Suppose $\alpha^{s_1} \overset{t_1}{\rightsquigarrow} \gamma^{s_3}$ in $\mathit{graph}(C_Y)$. Then $\beta^{s_2} \overset{t_2}{\rightsquigarrow} \gamma^{s_3}$ in $\mathit{graph}(C_Y)$.*

Lemma 5.4 *Suppose there is an ID of Y_S from which the head can be moved n positions left or right. Then there exists a positive path of depth n in $\mathit{graph}(Y_S)$.*

Proof: Suppose $\langle w_1,\alpha,m,w_1'\rangle$ and $\langle w_2,\beta,m+n,w_2'\rangle$ are reachable from each other. Consider a series of transitions from $\langle w_1,\alpha,m,w_1'\rangle$ to $\langle w_2,\beta,m+n,w_2'\rangle$. Let $\langle w_3,\gamma,m,w_3'\rangle$ be an intermediate ID in this series such that following transitions in the series do not move the head below the position m. Note that such an ID must always exist. Let m' be the highest position reached during the series of transitions. Let $s_3 = w_3'|_{m'-m}$ and $s_2 = w_2'|_{m'-(m+n)}$. Let t be the string such that $w_3t = w_2$. Note that $|t| = n$. We thus have $\langle w_3,\gamma,m,s_30^\infty\rangle$ and $\langle w_3t,\beta,m-|t|,s_20^\infty\rangle$ reachable from each other without moving the head below the position m and without moving the head above the position $m' = m+|s_3|$. Also, $|s_3| = m'-m = |t|+|s_2|$. Trivially, $\gamma^{s_3} \overset{\epsilon}{\rightsquigarrow} \gamma^{s_3}$. Therefore, by Lemma 5.3, $\beta^{s_2} \overset{t}{\rightsquigarrow} \gamma^{s_3}$. □

From Lemma 5.2 and Lemma 5.4, it follows that,

Lemma 5.5 *Y_S is bounded iff there exists a positive integer n such that for any positive path p in $\mathit{graph}(C_Y)$, $\mathit{depth}(p) \leq n$.*

Finally, by Lemma 4.1, Lemma 4.2, Lemma 4.4, Lemma 5.5 and Theorem 5.1,

Theorem 5.6 *Typability of* $\mathbf{I}_2 +$ **REC-INT** *is undecidable.*

6 Insufficiency of Unification Tests

One way to cope with the undecidability result is to reject some typable terms for the sake of an incomplete but terminating typability algorithm. Mycroft [10] proposed the following test as a rejection method for the Milner-Mycroft type system. For each `fix` $x.e$ and each occurrence of x in e, unify the type of the body e with the type of the occurrence of x and check that the constraints are satisfiable. The test rejects the term if any of the constraints are unsatisfiable, and otherwise runs the actual type inference algorithm hoping to have rejected any "bad" term that would make the algorithm diverge. For example, this method rejects the term `fix` $x.x\,x$ (which is typable in the Milner-Mycroft type system) because unifying the type of the first occurrence of x with the type of the body $x\,x$ results in a constraint of the form $\alpha{\rightarrow}\beta = \beta$, which is unsatisfiable. Here, we show that not only is this test insufficient for designing a terminating typability algorithm for $\mathbf{I}_2 + \textbf{REC-INT}$, but it is actually not sufficient even for the Milner-Mycroft type system.

We claim the following.

Lemma 6.1 *Suppose* $[\![\texttt{fix}\,x.e]\!]_\emptyset = (\tau, C, X)$ *and* C *is finitary satisfiable. Then for any* $s \in \{a \mid x^a \in \mathit{fvars}(e)\}^*$, $C \cup \{\tau^s = \tau\}$ *is still finitary satisfiable.*

Let $[\![\texttt{fix}\,x.e]\!]_\emptyset = (\tau, C, X)$. Suppose e contains no occurrence of `fix`. Then, applying the unification test in Lemma 6.1 for each string of length 1 (i.e., single characters) is more conservative than Mycroft's unification test. That is, if $C \cup \{\tau = \tau^a\}$ is satisfiable for each a in the alphabet $\{a \mid x^a \in \mathit{fvars}(e)\}$, then `fix` $x.e$ passes Mycroft's unification test. However, Lemma 4.2 and Lemma 6.1 imply that these tests do not reject any typable term of the form `fix` $x.e$. Recall e_Y from Section 5 is of the form `fix` $x.e$ such that e contains no occurrence of `fix`. Because the proof of Theorem 5.6 shows that even typability of e_Y terms is undecidable for $\mathbf{I}_2 + \textbf{REC-INT}$, it follows that Mycroft's unification test is insufficient for designing an incomplete but terminating typability algorithm for $\mathbf{I}_2 + \textbf{REC-INT}$.

Furthermore, it can be shown from the proof of undecidability of the Milner-Mycroft type system [7, 3] that a term of the form e_Y is typable in the Milner-Mycroft type system iff Y_S is bounded. Therefore, somewhat surprisingly, e_Y is typable in the Milner-Mycroft type system iff it is typable in $\mathbf{I}_2 + \textbf{REC-INT}$. Thus, Mycroft's unification test is insufficient for an incomplete but terminating typability algorithm even for the Milner-Mycroft type system.

In fact, Lemma 6.1 implies an even stronger result. An algorithm that tests $\tau = \tau^s$ for all strings s (not just single characters), regardless of whether such an algorithm exists or not, would be insufficient. More precisely,

$$
\begin{array}{rcl}
\alpha \sqcup \beta & = & \begin{cases} \alpha & \text{if } \beta < \alpha \\ \beta & \text{otherwise} \end{cases} \\
\alpha \sqcup (\tau{\rightarrow}\tau') & = & \tau{\rightarrow}\tau' \\
(\tau{\rightarrow}\tau') \sqcup \alpha & = & \tau{\rightarrow}\tau' \\
(\tau_1{\rightarrow}\tau_2) \sqcup (\tau_1'{\rightarrow}\tau_2') & = & (\tau_1 \sqcup \tau_1'){\rightarrow}(\tau_2 \sqcup \tau_2)
\end{array}
$$

Figure 9. $S_1 \sqcup S_2$.

Corollary 6.2 *Let*

$$
\begin{array}{rcl}
A & = & \{\texttt{fix}\,x.e \mid [\![\texttt{fix}\,x.e]\!]_\emptyset = (\tau, C, X) \\
 & & \wedge \forall s \in \{a \mid x^a \in \mathit{fvars}(e)\}^*. \\
 & & \models_{\mathit{fin}} C \cup \{\tau^s = \tau\}\}
\end{array}
$$

The following problem is undecidable. Let e *be a closed term such that there exists* $e' \in A$ *such that* e' *is* e *with each occurrence of a* `fix`*-bound variable annotated with a distinct number. Decide whether* e *is typable in* $\mathbf{I}_2{+}\textbf{REC-INT}$.

Corollary 6.2 follows from the fact that untypability of $\mathbf{I}_2 + \textbf{REC-INT}$ is not recursively enumerable (since typability is recursively enumerable). An analogous result holds for the Milner-Mycroft type system.

We now prove Lemma 6.1. Let $<$ be some total ordering over the type variables. Figure 9 defines the operation $\sqcup$ over the types. Note that $\sqcup$ is associative and commutative. We extend $\sqcup$ to constraint assignments as follows:

$$S_1 \sqcup S_2 = \{\alpha \mapsto S_1(\alpha) \sqcup S_2(\alpha) \mid \alpha \in \mathit{dom}(S_1) \cap \mathit{dom}(S_2)\}$$

Clearly, if S_1 and S_2 are both finite range then so is $S_1 \sqcup S_2$. Furthermore, if S_1 and S_2 both satisfy C, then so does $S_1 \sqcup S_2$, i.e.,

Lemma 6.3 *Suppose* $S_1 \models C$ *and* $S_2 \models C$. *Then* $S_1 \sqcup S_2 \models C$.

The following lemma says that we may "shift up" solutions for a constraint set of the form $\bigcup_{s \in \Sigma^*} C^s$.

Lemma 6.4 *Let* C *be a set of constraints and* Σ *be an alphabet. Let* $C' = \bigcup_{s \in \Sigma^*} C^s$. *Suppose* $S \models C'$. *Then for any* $s \in \Sigma^*$, $\{\alpha \mapsto S(\alpha^s)\} \models C'$.

We are now ready to prove Lemma 6.1, restated here.

Lemma 6.1 *Suppose* $[\![\texttt{fix}\,x.e]\!]_\emptyset = (\tau, C, X)$ *and* C *is finitary satisfiable. Then for any* $s \in \{a \mid x^a \in \mathit{fvars}(e)\}^*$, $C \cup \{\tau^s = \tau\}$ *is still finitary satisfiable.*

Proof: By inspection of the constraint generation rules (Figure 5), it must be the case that τ is a base type variable, say $\tau = \alpha$. Let $S \models_{\mathit{fin}} C$. We use the notation t^i to mean a string t concatenated i times. Since $\mathit{ran}(S)$ is finite, there must be m and n such that $m < n$ and $S(\alpha^{s^m}) = S(\alpha^{s^n})$.

For each $i \geq 0$, let $S_i = \{\beta \mapsto S(\beta^{s^i}) \mid \beta \in dom(S)\}$. By Lemma 6.4, it must be the case that $S_i \models C$ for each S_i. Furthermore, since $ran(S_i) \subseteq ran(S)$, each S_i is a finite range solution. Note that for each $i > 0$ and a type variable β, $S_i(\beta) = S_{i-1}(\beta^s)$. Also, $S(\alpha^{s^m}) = S(\alpha^{s^n})$ implies that $S_m(\alpha) = S_n(\alpha) = S_{n-1}(\alpha^s)$. Therefore,

$$\begin{aligned}
&(\textstyle\bigsqcup_{m \leq i \leq (n-1)} S_i)(\alpha) \\
&\quad = S_m(\alpha) \sqcup S_{m+1}(\alpha) \sqcup \ldots \sqcup S_{n-1}(\alpha) \\
&\quad = S_{n-1}(\alpha^s) \sqcup S_m(\alpha^s) \sqcup \ldots \sqcup S_{n-2}(\alpha^s) \\
&\quad = (\textstyle\bigsqcup_{m \leq i \leq (n-1)} S_i)(\alpha^s)
\end{aligned}$$

But by Lemma 6.3, $(\bigsqcup_{m \leq i \leq (n-1)} S_i) \models C$. Since each S_i is finite range, so is $\bigsqcup_{m \leq i \leq (n-1)} S_i$. Therefore, $(\bigsqcup_{m \leq i \leq (n-1)} S_i) \models_{fin} C \cup \{\alpha = \alpha^s\}$. □

7 Undecidability of REC-REACH

The constraint generation in Section 4 motivates the following program analysis problem. We extend the language with two constants, `red` and `blue`, and extend the constraint generation as follows:

$$[\![\texttt{red}]\!]_\Gamma = (\textit{red}, \emptyset, \emptyset)$$
$$[\![\texttt{blue}]\!]_\Gamma = (\textit{blue}, \emptyset, \emptyset)$$

Here, *red* and *blue* are base type variables distinct from all other type variables. The problem is to check that there exists no path of the form $\textit{red}^{s_1} \overset{\epsilon}{\leadsto} \textit{blue}^{s_2}$. Let us call this program analysis **REC-REACH**. This kind of *reachability* query is commonly seen in CFL-based program analyses [14] with applications in control flow analysis, points-to analysis, and other safety analyses. **REC-REACH** is a straightforward polymorphic recursive extension of a simple monomorphic unification-based flow analysis.

We use the framework developed in this paper to prove that **REC-REACH** is undecidable. In fact, it is not even recursively enumerable, which implies that there exists no type system equivalent to **REC-REACH** (in the sense of [11, 12]). While **REC-REACH** looks similar to the problem studied by Reps [15], our CFL graphs are more constrained, and we do not know whether his proof approach can be adopted.

Let $Y = \langle Q, A, T \rangle$ be an ITM such that $\textit{red}, \textit{blue} \in Q$. We build e_Y and obtain C_Y as in Section 5. Obviously, both Lemma 5.2 and Lemma 5.3 still hold. Furthermore, it is apparent from its proof that Lemma 5.2 can be strengthened to the following:

Lemma 7.1 *Let p be a path from α^{s_1} to β^{s_2} in $graph(C_Y)$ such that $sm(p)$ is a positive string. Then $\langle w@r(sm(p)), \alpha, m, s_1 0^\infty\rangle$ and $\langle w, \beta, m - |sm(p)|, s_2 0^\infty\rangle$ are reachable from each other in Y_S without moving the head below the position $m - |sm(p)|$.*

Combining Lemma 7.1 and Lemma 5.3, we have the following:

Lemma 7.2 *The following are equivalent:*

(1) There exists s_1 and s_2 such that there exists a path $\textit{red}^{s_1} \overset{\epsilon}{\leadsto} \textit{blue}^{s_2}$ in $graph(C_Y)$.

(2) There exists w_1 and w_2 such that $\langle w, \textit{red}, m, w_1\rangle$ and $\langle w, \textit{blue}, m, w_2\rangle$ are reachable from each other in Y_S without moving the head below the position m.

(3) There exists w_1, w_2, and an ID W such that $\langle w, \textit{red}, m, w_1\rangle$ reaches W in Y without moving the head below the position m and $\langle w, \textit{blue}, m, w_2\rangle$ reaches W in Y without moving the head below the position m.

Problem (3) can be proved to be undecidable via the reduction from the halting problem. Therefore, problem (1) is undecidable. It is easy to see that problem (1) is recursively enumerable. Since **REC-REACH** is the dual of problem (1), it follows that

Theorem 7.3 **REC-REACH** *is not recursively enumerable.*

8 Conclusions

This paper shows that typability of $\mathbf{I}_2 + $ **REC-INT** is undecidable by means of characterizing typability as a CFL graph problem and reducing from the boundedness problem of Turing machines. We found reducing to an infinite graph problem leads to a more understandable proof than reasoning directly on infinite type constraints. We suspect that a similar proof can be used to show that the problem remains undecidable for extensions to any higher rank (e.g., the system investigated in [2]).

As a corollary of the undecidability result, we showed that the unification test is insufficient to build an incomplete but terminating typability algorithm for $\mathbf{I}_2 + $ **REC-INT** or the Milner-Mycroft type system. We also proved undecidability of the related program analysis **REC-REACH** by using the same CFL graph framework.

One open question is whether the following problem is decidable. Given a closed e, is there S such that $S \models C$ where $[\![e]\!]_\emptyset = (\tau, C, X)$ for some τ and X? Note that if we strengthened the requirement to $S \models_{fin} C$ then the problem becomes the typability problem for $\mathbf{I}_2 + $ **REC-INT** and therefore becomes undecidable. There is a larger open question: where the boundary between decidability and undecidability is when it comes to polymorphic recursion (and how to state this question in a formal way).

References

[1] F. Damiani. Rank-2 intersection and polymorphic recursion. In *Typed Lambda Calculi and Applications: 7th International Conference (TLCA 2005)*, volume 3461 of *LNCS*, pages 146–161. Springer, Apr. 2005.

[2] J. J. Hallett and A. J. Kfoury. Programming examples needing polymorphic recursion. In *In Proceedings 3rd International Workshop Intersection Types and Related Systems (ITRS 2004)*, pages 57–102, 2004.

[3] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, Apr. 1993.

[4] T. Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Cambridge, MA, USA, 1995.

[5] T. Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53, St. Petersburg Beach, Florida, Jan. 1996.

[6] A. J. Kfoury, H. G. Mairson, F. A. Turbak, and J. B. Wells. Relating typability and expressiveness in finite-rank intersection type systems (extended abstract). In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 90–101, Paris, France, Sept. 1999.

[7] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, 1993.

[8] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311:1–70, 2004.

[9] D. Leivant. Polymorphic type inference. In *Proceedings of the 10th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 88–98, Austin, Texas, 1983.

[10] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Conference on Programming*, number 167 in LNCS, 1984.

[11] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, California, Jan. 1995.

[12] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, San Diego, California, Jan. 1998.

[13] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to cfl-reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, Jan. 2001.

[14] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11/12):701–726, November/December 1998.

[15] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

[16] T. Terauchi and A. Aiken. On typability for rank-2 intersection types with polymorphic recursion. Technical Report UCB/EECS-2006-66, University of California, Berkeley, May 2006.

[17] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.