東京大学
情報理工学系研究科 創造情報学専攻
修士論文

高水準動的型付け言語間 FFI における Type Description Helper の構築

Building Type Description Helper in FFI between High-level Dynamically Typed Languages

池崎 翔哉
Ikezaki Shoya

指導教員　千葉 滋 教授

2020 年 1 月

概要

他言語インターフェース（FFI）はある言語から他言語のライブラリを使用するための仕組みである．プログラミング言語の実用上，他言語で書かれたライブラリ資産を使用できることが重要である．近年では豊富な型を有する動的型付け言語にライブラリ資産が溜まってきているため，このような言語との FFI が望まれている．しかし型の豊富さゆえに，ホスト言語とライブラリ言語間の型変換規則が複雑なものとなり型変換規則の記述量が多くなってしまうという問題がある．本研究は Type Description Helper という型変換規則の半自動的な導出器を提案する．これが導出した型変換規則の分，ユーザーが記述する必要のある型変換規則の量をおさえることができる．Type Description Helper による型変換規則の導出の手法として二つの方法を考案した．ログベースド法は他言語関数呼び出し時のログから動的に型変換規則を導出し，型推論ベースド法はソースコードを静的に解析することで型変換規則を導出する．本研究では具体例として Python 及び EusLisp の 2 言語を選定し，この間の FFI を作成すると共に Type Description Helper の実装を行い，その有用性を確認した．

Abstract

A foreign function interface (FFI) is a mechanism that enables a programming language to use libraries written in another foreign language. The FFI is important since the language that does not have access to the rich libraries that already exist is not considered practical. Recently, the FFI between dynamically typed languages that have various types is required because a great number of useful libraries are written in those languages. However, it takes a high cost to describe the type conversion rules between host and library languages since the rules of it are complex on account of type-richness. We propose a Type Description Helper that derives the type conversion rules semi-automatically. The Type Description Helper reduces the number of type conversion rules that users have to write. As the type conversion rules that are derived by Type Description Helper are incomplete, users have to write the type conversion rules that have not yet been derived. Type Description Helper derives the type conversion rules in two ways: log-based approach and type-inference-based approach. The log-based approach derives the type conversion rules dynamically from the log of foreign function calls. The type-inference-based approach derives the type conversion rules by analyzing the source code statically. At the same time, it can detect some invalid arguments that cannot be applied by foreign functions. In this research, we choose Python and EusLisp as an example and implement FFI and Type Description Helper, then checked that it is useful.

目次

第 1 章

はじめに

ある言語が他言語で書かれたコードを関数単位で使用できるようにする機能のことを FFI（Foreign Function Interface）という．プログラミング言語が実用的であるためには，その言語が豊富なライブラリ資産を使用可能であることが重要である．洗練されたライブラリ資産の再利用は，その分のコードを書く手間を削減し開発コストを下げるとともに，不要なバグを埋め込む危険性も減少させる．FFI により，あるプログラミング言語が同言語で書かれたライブラリだけでなく他言語のライブラリを使用できるようになる．ホスト言語とライブラリ言語の間で値の参照渡しをできるようにしたり，その言語の有する関数やクラスについての情報をやりとりしたりできるように実装するのは多少複雑ではあるが，プロキシパターン [1] やリフレクション [2] といった既存の手法を用いて実装可能であると知られている．

従来は C 言語との FFI が重要であると考えられてきた．システムコールをはじめとした様々なライブラリ資産が溜まっている言語は C 言語であったので，C 言語の機能を他のホスト言語から使用できるようにすることは，ホスト言語がアクセス可能なライブラリ資産の質と量をあげる上で重要であった．そのため，Java, Python, Ruby, Rust といった現在幅広く用いられている言語では，C 言語との FFI が開発されている．

一方で，近年では C 言語以外の豊富な型を持つ動的型付け言語にも多様なライブラリ資産が溜まっており，それらとの FFI も重要になっている．例を挙げると，Python は NumPy[3], SciPy[4], PyTorch[5], TensorFlow[6] など数多くの強力なライブラリを有する．ROS (Robot Operating System) [7] の開発言語のひとつでもあるようにロボティクスの場でも広く用いられている．また，Lisp 言語の一つであり効率的なロボット開発のためのプログラミング言語である EusLisp[8] は，直感的に逆運動学を解くことのできるライブラリや，HRP2 といったロボットの制御ライブラリ群 [9][10] も持ち合わせている．こうした言語の有するライブラリの再利用が望まれている．

一般にある言語が他言語の関数を使用する際には型変換が必要となるが，この型変換規則の記述量が多くなってしまい，問題である．他言語関数使用の際 FFI はホスト言語のデータを他言語へと送るが，言語間で有する型は異なるため，同じデータでも両言語でそれを表現する型は異なる．そのため型同士の対応付けである型変換が必要となる．このとき，対象言語が共に豊富な型を有する場合，型同士の対応関係が多対多となる．このような言語の FFI にお

いては．型に対するデフォルトの変換規則を用意しておくだけでは対応しきれない．デフォルトの型変換の規則から外れて型変換を行いたい事例が多発してしまい，型変換規則を毎回ユーザーが記述し FFI に与えなくてはならなくなる．そのような言語同士の FFI の一例として，今回の実装のターゲットでもある Python, EusLisp の FFI を具体的に取り上げて説明を行う．両言語についてシーケンス型に対応する型を図 1.1 に示す．図のように，両言語で有する型が豊富であるがゆえに，型の対応関係が多対多となっている．EusLisp の float-vector を引数にとる他言語関数を Python から使用したい場合，Python には float-vector などという型はないため，対応するシーケンス型を float-vector へと型変換して他言語関数呼び出しを行いたい．しかし，FFI は他言語関数呼び出し時に引数として渡された Python のシーケンス型を EusLisp のどの型に変換すれば良いのか分からない．EusLisp のシーケンス型がただ一つであるならば変換先を決定可能だが，変換先の候補が複数考えられるためユーザーが何らかの形で明示的に型変換先を教える必要がある．このように，ユーザーは何らかの形で型変換規則の記述を行う必要があるが，複雑かつ膨大な型変換規則をユーザー任せで毎回記述させるのは親切ではない．このユーザーによる手動の型変換規則記述を削減するためのシステムが求められる．

ユーザーが FFI に型変換規則を手動で与える必要があり記述量が多くなってしまうという問題を解決するために，この型変換規則を半自動的に導出する仕組みを開発することで，ユーザーの与える必要のある型変換規則の量を減らし記述量を減らす．本研究では，高水準動的型付け言語間の型変換規則の半自動的な導出器である Type Description Helper を提案する．これは半自動的に型変換規則を導出するもので，Type Description Helper が導出できた型変換規則の分だけ，ユーザーが与えなくてはならない型変換規則の量は減り，記述量も減る．Type Description Helper が導出可能な型変換規則は，値に対するものと関数に対するものの 2 種類である．また，この Type Description Helper は，ユーザーが記述する必要のある型変換規則の量を減らすと共に，明らかに適用不可能な型の引数の検出も行う．他言語関数の呼び出し時にホスト言語のオブジェクトをそのまま渡すかのような引数記述が可能になり，一部の型エラーは他言語関数の実行前にホスト言語側のみで発見することが可能になる．

提案手法を試すにあたり，ホスト言語には Python を，そこから呼び出す他言語には EusLisp を選定し，従来型の FFI および提案手法の型変換規則の記述インターフェースを実装した．これらは両言語共にオブジェクト指向かつ動的型付け言語と，互いによく似た言語であると言うこともできるが，ともに型が豊富であり有する型が異なるため，それらの FFI は上述した問題を抱える．また実際に，EusLisp を Python から使いたいという要望は実際に我々の周囲のロボット研究グループの中にあり，実用上の意義もある．

以降，第 2 章では背景知識を，第 3 章では我々の動機を，第 4 章では提案手法を示す．第 5 章では提案手法の実装を，第 6 章では提案手法の有用性を具体的なコードとともに確認する．そして第 7 章では提案手法の有用性を，既存のライブラリを通して定量的に評価する．第 8 章では関連研究を示し，最後に第 9 章ではまとめと今後の展望について記す．

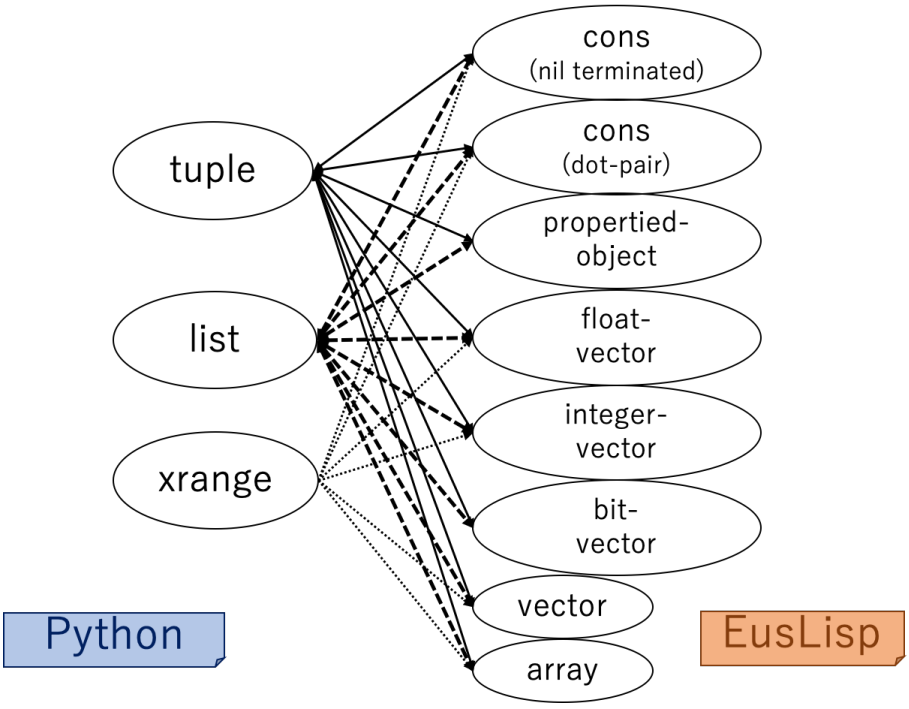

図 1.1. Python と EusLisp のシーケンス型とそのマッピング

第 2 章

背景知識

2.1 Foreign Function Interface (FFI) の位置付け

ある言語が他言語の機能を使用可能であることは，既存のライブラリ資産の有効活用に繋がるため，重要である．ホスト言語がライブラリ言語の機能を使用できるようにするには，言語間の差異を解消する必要がある．それを実現する手法の一つとして，ある言語が他言語で書かれたコードを関数単位で使用できるようにする機能である FFI（Foreign Function Interface）が考えられていることを第 1 章で述べた．本節ではそれを実現する他の例としてソーストゥソースコンパイラ, 共通言語での処理系の実装の二つを取り上げ，FFI との比較を行い，FFI の位置付けを考える．

言語間の差異の解消のためには，究極にはライブラリ言語で書かれたソースコードをホスト言語のソースコードに，もしくはホスト言語で書かれたソースコードをライブラリ言語のソースコードに完全に変換してしまえばよい．ソーストゥソースコンパイラは高水準言語のソースコードを入力に別の高水準言語のソースコードを出力する変換機であり，それを実現する．ソースコードレベルで同一言語にしてしまえば言語間の差異はそもそもなくなるという発想である．Kotlin コードから JavaScript コードの変換機である KotlinJS，TypeScript コードから JavaScript コードへの変換機である TypeScript コンパイラ，Cython コードから C コードへの変換機である Cython コンパイラなどがある．しかしながら，自身の言語が相手の言語への変換を念頭に置いていない場合，変換機の作成は難しい．言語機能の差異が大きすぎる場合，差異の解消は困難であると考えられる．

他にはホスト言語の処理系の解する言語でライブラリ言語の処理系を実装してしまうという手法がある．ソースコードレベルで直ちにライブラリ言語をホスト言語に変換するわけではないが，ホスト言語の処理系が解する言語（中間言語など）のレベルで変換が同一の言語へと変換がなされ，言語間の差異がそもそもなくなる．Truffle[11] は言語実装フレームワークであり，AST インタプリタ構築のための基盤を提供するものである．JIT コンパイラである Graal とともに用いることで Truffle インタプリタは自動的に JIT コンパイルされる．図 2.1 にその概念図を示す．ライブラリ言語のソースコードを AST へと変換し，その AST のインタプリタを構築しさえすれば，JVM で実行可能な Java バイトコードへと変換することが可

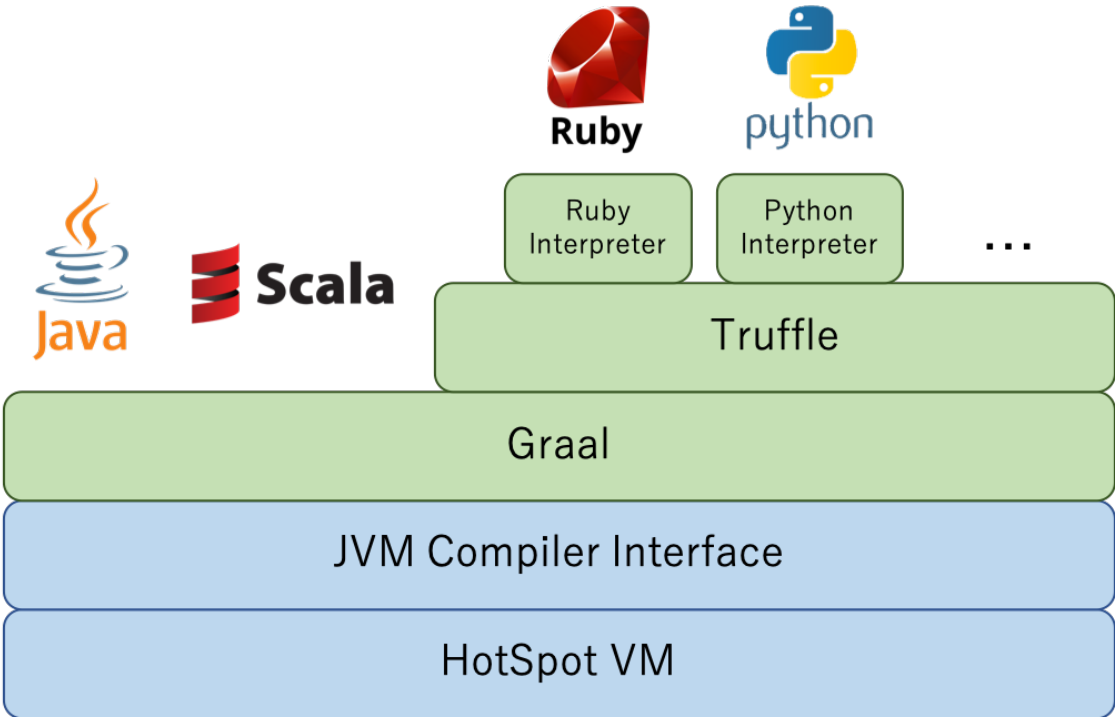

図 2.1. 共通言語での処理系の実装の一例

能である．しかし言語機能全体の差異を解消するために汎用の処理系の実装が必要となっており，コストがかかると言える．

対し，FFI はライブラリ言語で書かれたコードを関数単位に限りホスト言語から使用できるようにする手法である．使用できる機能を絞る代わりに，限られた言語間のやりとりにおける差異のみを吸収すれば十分になるという利点がある．解消するべき差異は関数呼び出し方法の違い，データ型の違いなどであり，特にデータ型の違いが FFI の形態によらず重要となる．図 2.2 にさまざまな FFI の形態の例を挙げる．左上は Python/C API[12] などに代表されるような単一プロセスでの FFI の例，右上は ErlPort[13] に代表されるような複数プロセスでの FFI の例，下は CORBA[14] に代表されるようなネットワークを介した複数プロセスでの FFI の例である．どの例についても異なる色で表されている環境については表現可能なデータ型が異なるため，変換が必要となる．データ型の変換が適切に行われれば，主要な差異は解消される．

2.2 FFI と型変換

ホスト言語とライブラリ言語で有する型が異なる場合，FFI は引数や戻り値に対して型変換を行う．一般に引数や戻り値の扱い方として，コピー渡しと参照渡しの 2 つが存在する．コピー渡しの場合，常に変換先の型を決定することが必要である．参照渡しの場合，特定のクラスのインターフェースを持たない汎用のプロキシーオブジェクトに変換してしまう場合以外は

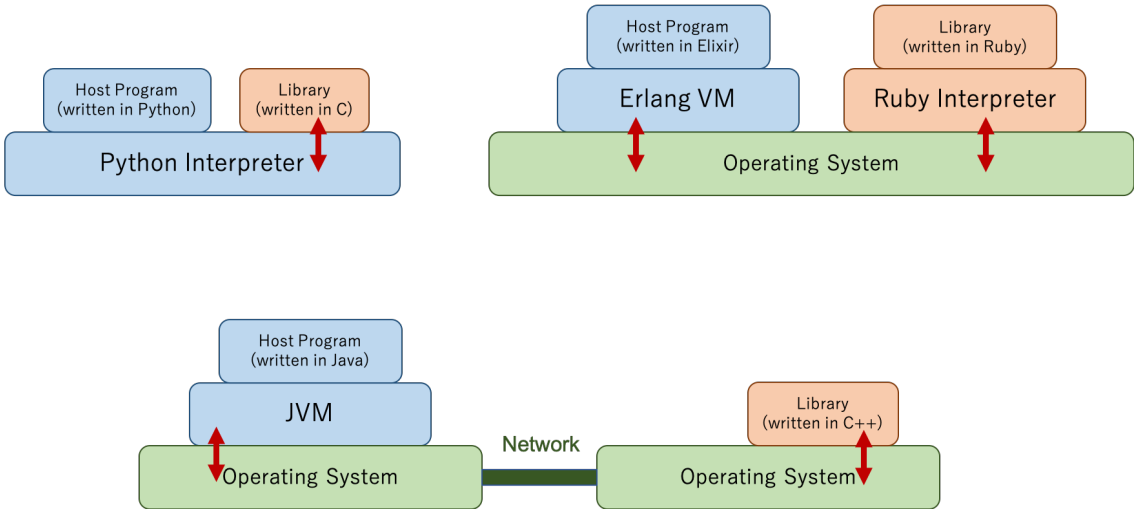

図 2.2. FFI は言語間のやりとりにおける差異を吸収する

変換先の型を決定することが必要である．

コピー渡しで他言語関数呼び出しを行う例を図 2.3 に示す．型変換の様子をより具体的に捉えるため，本研究で実装した FFI の両言語を例にとっているが一般化可能である．float-vector を引数にとる EusLisp 関数を Python から呼び出したい場合を考える．Python に float-vector という型は存在しないため，今回は Python の list を引数に呼び出すことにする．コピー渡しの戦略をとる場合，単純に引数の list に対応するような float-vector オブジェクトを EusLisp の環境にコピー生成し，それを引数に関数呼び出しを行えば良い．ここで変換先の型の決定および型変換が必要となった．戻り値についても同様である．EusLisp のオブジェクトの型に対応するような Python のオブジェクトを Python の環境にコピー生成することになるため，変換先の型の決定および型変換が必要となる．

参照渡しで他言語関数呼び出しを行う例を図 2.4 に示す．型変換の様子をより具体的に捉えるため，本研究で実装した FFI の両言語を例にとっているが一般化可能である．参照渡しの戦略をとる場合は基本的には，float-vector の適切なインターフェースを持つ，引数の Python の list を指すようなプロキシーオブジェクトを EusLisp の環境に作成し，それを引数に関数呼び出しを行えば良い．戻り値についても同様である．変換先の型の決定および型変換が必要となる．

一方，参照渡しについては変換先の型の決定を必要としない手法も考えられることを述べておく．特定のクラスのインターフェースを持たない汎用のプロキシーオブジェクトに変換してしまう場合である．引数についてこのような汎用的な変換を施した場合は，用途がかなり限定される．考えられる用途としては EusLisp で定義された高階関数を，Python の関数および Python のオブジェクトを引数に使用したい場合などだろう．この場合コールバックの際には Python 側でコールバック関数の適用を行うため，EusLisp 側で特定のクラスのインターフェースを持つ意味は薄い．戻り値についてこのような汎用的な変換を施した場合は使用用途

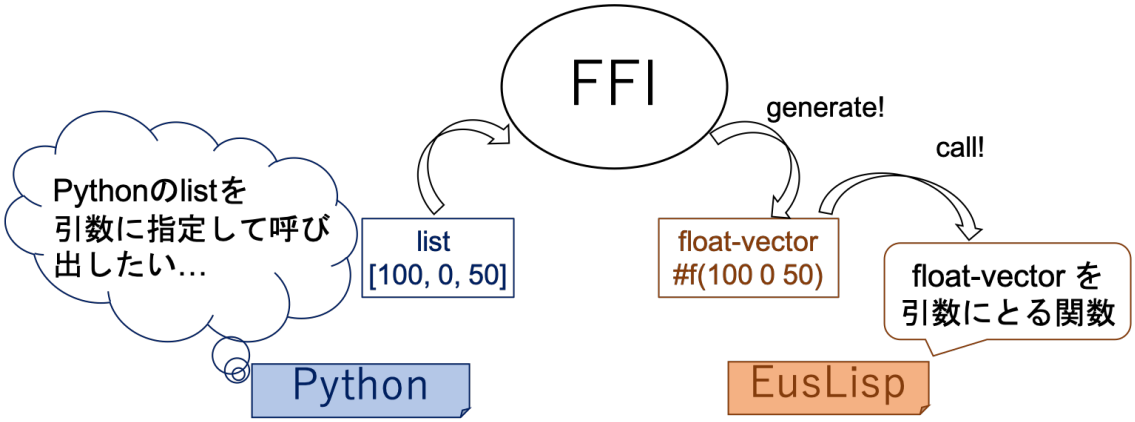

図 2.3. コピー渡しで他言語関数呼び出しを行う例

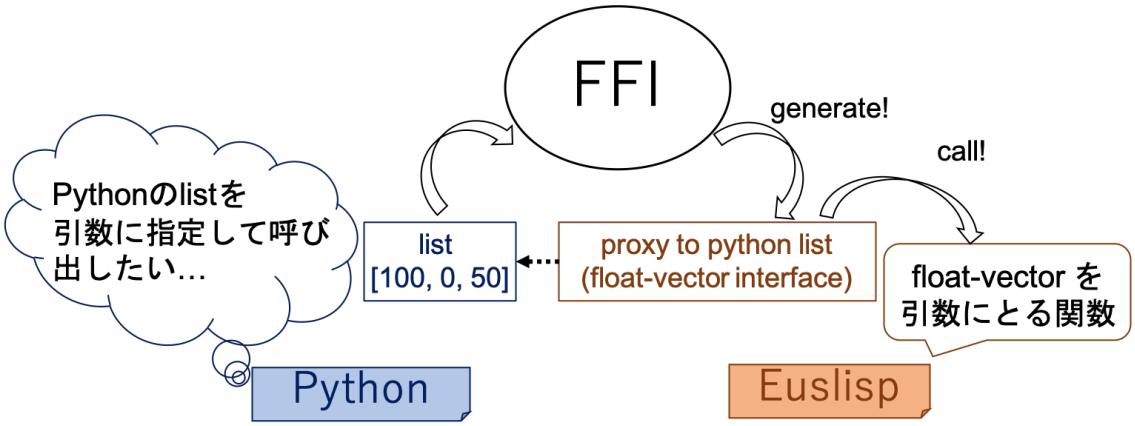

図 2.4. 参照渡しで他言語関数呼び出しを行う例

により有用である．EusLisp のオブジェクトを指す特定の Python のクラスのインターフェースを持たないプロキシーオブジェクトを，通常の Python オブジェクトと同等に扱うことはできない．しかし再び EusLisp 他言語関数に渡すことは可能である．Python オブジェクトへの変換メソッドをプロキシーオブジェクトに用意しておくことで，Python のオブジェクトのように扱いたい場合はいつでも変換可能にすることができる．

以上のように，FFI は関数呼び出しの際に引数と戻り値に対して型変換を行う．そして引数に関しては常に，戻り値に関しては実装により変換先の型の決定が必要である．

2.3　計算機科学における型および型システム

これまでに FFI は他言語関数呼び出し時に型の変換を行うことを確認した．本節では計算機科学における型および型システムについて，提案手法と深く関連する内容について述べる[*1]．

2.3.1　型および型システムとは

計算機科学における型システムとは，プログラムの各部分を予想される実行結果の値の種類によって分類することで，プログラムがある種の振る舞いを起こさないことを保障する手法である．この分類に用いられる情報を型と呼ぶ．プログラムが何らかの型の分類に当てはまることを，プログラムに型が与えられるという．分類が単純であるとエラーの誤検知が増えたりや検出できるエラーの範囲が狭まったりしてしまう．型の表現力を豊かにするとそれらの問題が解消されるが，プログラムに型を与えられるかどうかの判定が複雑になるというトレードオフが存在する．なお，一般にはこの分類は静的に行うもので，動的型付け言語が行うような，ヒープ中のデータ構造を区別するための型タグの情報に基づく動的な検査とは区別をする．

2.3.2　型付け判断，型付け規則と安全性

「式 e が何らかの型 τ の分類に当てはまる」という判断を型付け判断と呼び，$e : \tau$ と書く．型付け判断を導出するための推論規則は型付け規則と呼ぶ．型付け規則には，$0 : \mathrm{int}$ といった定数に対する型付け判断を前提なしにで導出するための規則や，「$e1$, $e2$ に int 型が与えられるならば $e1 + e2$ には int 型が与えられる」というような規則が与えられる．通常前提に現れる式が結論の判断に現れる式の部分式であるような形で与えられる．

型システムにおける基本的な性質は安全性であると言われる．これは正しく型付けされた項は行き詰まり状態（所定の最終的な値ではないが次に適用すべき評価規則もない状態）にならないということである．一般に，定義した型システムの安全性に関するメタ定理を証明したい場合，進行（正しく型付けされた項は行き詰まり状態でない）及び保存（正しく型付けされた項の評価ステップを進められるならば，評価後の項も正しく型付けされている）という二つの性質をもって証明することができる．本論文で扱う，型の上界を求めるアルゴリズムにおいてこの安全性は保障されないが，その有用性は後の章で確認する．

2.3.3　基底型，関数型，型環境

int, bool のような，型を構成する基本単位となる型を基底型という．基底型のみからなる型システムを拡張して関数を含むようにするには，関数に評価される項を分類する型が必要となる．一般の矢印型 $\rightarrow$ としてもよいが，$T_1 \rightarrow T_2$ という形の無限の型の属として分類すること

[*1] 本節を執筆するにあたって，全面的に「記号論理入門 [15]」「プログラミング言語の基礎概念 [16]」「型システム入門 [17]」「プログラム意味論 [18]」「プログラム検証論 [19]」を参考にした．

で，より詳細な型付けが可能となる．関数に型を付けるためには関数引数の型を知り，その型の引数に適用された結果の型を知る必要がある．引数の型が与えられる言語を明示的に型付けされた言語，与えられない言語を暗黙に型付けされている言語といい，後者では型検査器が引数の型を推論，再構築する必要がある．また，変数を含む式の型は式に含まれる変数の型に関する仮定が与えられてはじめて議論することができる．このような変数の型に関する仮定を型環境と呼び，型判断の形式は「型環境 Γ の下で e は τ 型の式である」という意味の $\Gamma \vdash e : \tau$ という形式を考える．以下に一例として純粋単純型付きラムダ計算の構文，評価規則，型付け規則を載せる．このシステムについて型の進行，保存が成立する．

構文定義（項 t，値 v，型 T，型環境 Γ）

$$
\begin{aligned}
t &::= x \mid \lambda x : T \,.\, t \mid t\ t \\
v &::= \lambda x : T \,.\, t \\
T &::= T \to T \\
\Gamma &::= \phi \mid \Gamma, x : T
\end{aligned}
$$

評価規則

$$
\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}
$$

$$
\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}
$$

$$
(\lambda x : T_{11} \,.\, t_{12})\ v_2 \to [x \mapsto v_2] t_{12}
$$

型付け規則

$$
\frac{x : T \in \Gamma}{\Gamma \vdash x : T}
$$

$$
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t_2 : T_1 \to T_2}
$$

$$
\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}
$$

2.3.4 部分型多相

部分型付け（部分型多相）[20] はオブジェクト指向言語において特徴的に見られるもので，しばしばオブジェクト指向の本質的な機能であると考えられる．部分型付けがなければ，関数引数の型が定義域の型と完全に一致していなければならないということを型システムが強く要求するせいで，よく振る舞うことが明らかなプログラムが型検査器に拒否されてしまう．そのような項を許すため，ある型が他の型より情報を持っているということを定式化する．型 S の

任意の項が型 T の項が期待されている文脈で安全に使用可能であるということを，S は T の部分型であるといい，$S <: T$ と書く．直感的にはこれを「S で記述される全ての値は T でも記述される」と読めば良い．すなわち，S の全ての要素からなる集合は T の全ての要素からなる集合の部分集合になっていると見る．この解釈を部分集合意味論といい，本論文ではこの解釈を前提とする．

部分型関係と型付け関係をつなぐための型付け規則は以下である．この規則は包摂規則と呼ばれる．

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

そして部分型付けは反射的かつ推移的であると定める．これらの規則により部分型関係は前順序となる．

$$S <: S$$

$$\frac{S <: U \qquad U <: T}{S <: T}$$

関数型についての部分型付け規則を以下のように定めることができる．左の前提を見ると，引数の型と関数の型について部分型関係の向きが反転している．これを反変 (contravariant) であるという．右の前提を見ると，戻り値の型と関数の型について，部分型関係の向きが一致している．これを共変 (covariant) であるという．

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

部分型関係の強力な改良は型の言語に交差演算子を導入することで得られる．交差型 (intersection type) $T_1 \wedge T_2$ の要素は，T_1 及び T_2 の両方に属す項である．交差型に関する部分型付け関係を以下に示す．また，交差型の双対な概念として合併型 (union type) $T_1 \vee T_2$ も考えることができる．

$$T_1 \wedge T_2 <: T_1$$

$$T_1 \wedge T_2 <: T_2$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2}$$

$$(S \to T_1) \wedge (S \to T_2) <: S \to (T_1 \wedge T_2)$$

$$T_1 \vee T_2 :> T_1$$

$$T_1 \vee T_2 :> T_2$$

$$\frac{S :> T_1 \qquad S :> T_2}{S :> T_1 \vee T_2}$$

$$(S \to T_1) \vee (S \to T_2) :> S \to (T_1 \vee T_2)$$

これまでに挙げた型のほかに，レコードのような型を導入し，並び替え規則（あるレコード型に対し，並び順を入れ変えただけの型は部分型となる），幅部分型付け規則（あるレコード型に対し，それらのフィールドを含むレコード型は部分型となる），深さ部分型付け規則（あるレコード型に対し，各フィールドが部分型となっているフィールドを持つレコード型は部分型となる）を定めることもできる．しかしレコードのような型を導入する場合，レコードの並び替え規則の影響で，異なる型の組でも互いに部分関係であるようなものが存在するようになるため，半順序とならないことがある．本論文では扱わないことを強調しておく．

また，オブジェクトの型（インターフェース）をオブジェクトの操作名と型の集まりと捉えると，オブジェクトのインターフェースは部分型関係に自然に適合する．レコードの部分型付け同様に扱うことが可能である．しかし本論文では，簡単のため条件を絞り，単純なサブクラス関係のみを判定してオブジェクトの部分型関係を決定することを強調しておく．（対象とする EusLisp では単一継承のみが許され，複数インターフェースの継承という概念がないためである）

2.3.5 結びと交わり

型の二つ組 S，T に対して，ある型 J が，$S <: J$ かつ $T <: J$ かつ任意の型 U に対して，$S <: U$ かつ $T <: U$ ならば $J <: U$ となるとき，J を S と T の結びと呼び，$S \vee T = J$ と書く．同様に，ある型 M が，$M <: S$ かつ $M <: T$ かつ任意の型 L に対して，$L <: S$ かつ $L <: T$ ならば $L <: M$ となるとき，M を S と T の交わりと呼び，$S \wedge T = M$ と書く．

ある部分型関係について，全ての型 S，T について S と T の結びとなる J が存在するとき，この部分型関係は結びを持つという．同様に，全ての型 S，T について S と T の交わりとなる J が存在するとき，この部分型関係は交わりを持つという．

また，型の二つ組 S，T が，ある型 J が存在して $S <: J$ かつ $T <: J$ となるとき，上に有界であるという．同様に，型の二つ組 S，T が，ある型 L が存在して $L <: S$ かつ $L <: T$ となるとき，下に有界であるという．下に有界である全ての型 S，T について，S と T の交わりとなる J が存在するとき，この部分型関係は有界な交わりを持つという．本論文で扱う部分型関係は，結び及び有界な交わりを持つ．

2.3.6 パラメトリック多相

一つの式が持つ複数の型のパターンがパラメータを使って統一的に表せるという多相性をパラメトリック多相という．例えば，恒等関数に与えられる型の集合は $\forall \tau \in Types.\ \tau \to \tau$ である．変数を通じて多相性の情報が伝播でき，変数参照の出現ごとに異なる型を与えることができる．let 多相（冠頭多相）はパラメトリック多相のうち限定されたものである．このシステムでは let により束縛される変数に限り多相性の伝播を許すというものである．（型変数は

量化子のない型（単型）のみを動き，量化された型（型スキーム）の上を動かない.）この制限を入れることで，let 多相の型システムに対して型再構築アルゴリズムを構成できることが知られている．本論文で扱うシステムでは let 多相を導入する．

2.3.7　有界量化

部分型付けとパラメトリック多相を組み合わせると興味深い結果が得られる．ここではパラメトリック多相のうち let 多相を考え，型変数は単型のみを動くとする．これに部分型付けを組み合わせることで，型変数の動く型を限定することができる．例えば，型 T の部分型であるいかなる引数もとり，それと同一の型を返す関数型 $\forall\tau <: T.\ \tau \to \tau$ といった型を付けることが可能となる．第 2.3.6 節で扱った非有界な恒等関数の型は，$\forall\tau <: Top.\ \tau \to \tau$ と型をつけることが可能となる（Top は型定数であり，部分型関係の最大の要素を表す）．本論文で扱うシステムでは let 多相で扱うパラメータに対し有界量化がなされている．

2.3.8　アドホック多相

アドホック多相（オーバーローディング）とは，一つの式に複数の型が与えられるものの，それらの型に特に一定のパターンがない場合を指す．例えば演算子 + が整数同士の加算のための演算子であるとともに，文字列の結合のための演算子であるような場合である．プログラマが定義する関数にアドホック多相を導入し，なおかつ型再構築をおこなえるようにするための言語レベルでの仕組みとして，Haskell の型クラス [21] が知られている．

2.3.9　型再構築

第 2.3.3 小節で見たように，明示的に型付けされた言語，暗黙に型付けされている言語が存在し，後者については型注釈のついていない項の主要型を型再構築により推論する必要があるのだった．部分型を伴うものについては複雑となるため，ここでは単純型のみの型再構築の問題を取り上げることとする．

まず「項 t のプレースホルダ X をある型で具体化した場合に，型付け可能な項が得られるか」という問いに応えるため，他の型を代入可能な型変数を考える．また，型を型変数へ代入する操作を以下の二つに分けて考える．すなわち，型変数から型への写像 σ（型代入）の定義と，この対応を具体的な型 T に適用して具体化 σT を得ることである．未知の型である型変数の存在を考えると，型再構築の問題は「与えられた型環境 Γ と式 e から，$S\Gamma \vdash e : \tau$ を満たすような型代入 S と型 τ を求める」という形に定式化することができる．ここで，型代入は Γ の中の未知の型を表す型変数の正体を明らかにするものであると言える．

また，式には複数の型が与えられるときがある．恒等関数は任意の型 τ に対して型 $\tau \to \tau$ を与えることができる．その中でも特に，型 $a' \to a'$ （a' は型変数）は a' に型代入を施すことで他の全ての型判断が得られるという意味で，この項の最も代表的な型であると言える．例で示したような，式に対する最も代表的な型のことを主要型と呼ぶ．

型再構築のための主要なアイディアは型に関する方程式を立て，解くことである．（単純型の場合，関数適用の部分で型に対する方程式が立つ．しかしながら部分型を導入する場合は関数適用の部分で型に対する不等式が立つことになる．）前者の立式部分について，型環境と式を受け取り，式の各構成要素について，それに対応する型付け規則にそって，未知の型については新しい型変数を生成しながら再帰的に型に関する方程式を集めていけばよい．後者の求解部分について，得られた型変数に対する連立方程式の解，すなわち各方程式の両辺を同一の型にするような型代入を求めればよい．この問題は単一化問題として知られる．一階の単一化問題については解（単一化子）が存在する場合には，全ての解を代表するような解（最汎単一化子）が存在することが知られている．これを求める一般的なアルゴリズムは Robinson[22] によるアルゴリズムが知られている．let 多相の型システムに対する型再構築アルゴリズムとして Hindley[23], Milner[24] によるアルゴリズムが知られている．let 多相の型システムに対して型再構築を行う場合，方程式を全て求めてからそれをまとめて単一化するのではなく，方程式の生成と単一化を交互に行っていくことで実現可能である．

第 3 章

従来の型変換の規則の記述法と問題点

ホスト言語とライブラリ言語で有する型が異なる場合，FFI は型変換を行う．この型変換の規則は明らかではないため，ユーザーが型変換の規則を記述し，FFI に与えてやる必要がある．既存の型変換規則を記述する方法としては値に対して型変換の規則を記述する手法，関数に対して型変換の規則を記述する手法の 2 つがあるが，いずれも記述量が多くなってしまうという問題がある．本章ではこれらの手法およびその具体例をそれぞれ説明し，問題点を示す．最後に実際にこれら手法を全て用いて実装した[*1]ときの様子を具体的に見ていく．

3.1 値に対する型変換の規則の記述

値に対する型変換規則の記述とは，ユーザーがホスト言語の各データに対して変換されてほしいライブラリ言語の型情報を適宜付与するというものである．ホスト言語とライブラリ言語の型が往々にして多対多対応していることが問題であるため，ライブラリ言語の型と一対一対応する型をホスト言語側に用意しておけばよい．こういった対応する型が明らかなオブジェクトしか他言語関数呼び出し時に渡せないことにすれば他言語側で適切に関数実行が可能である．

値に対する型変換規則の記述は，デフォルトの型変換の規則を補完する形で，ユーザーが必要に応じて動的に値に対し行う．ライブラリ言語側にホスト言語の型と一対一対応する型が存在する場合はその型へ自動的に変換を行う．ライブラリ言語側にホスト言語の型と対応する型が複数存在する場合はそのうちの一つへ自動的に変換を行う．しかしながら，このデフォルトの型変換の規則から外れて他の候補の型へと変換を行いたい場合は，データに対してユーザーが毎回明示的に型変換を行う必要がある．そのため，ユーザーが与える必要のある型変換規則の記述の量が多くなってしまい，問題である．

この値に対する型変換規則の記述の具体的な説明のため，Python と C++ の仮想の FFI を

[*1] 実際の完全な実装は https://github.com/ikeshou/PyEus から見ることができる．

考える．Python と C の FFI である ctypes のように，単純な整数や文字列に対するデフォルトの型変換規則を設定することは可能であるが，それ以外のデータ型を使用する場合型変換規則をユーザーが書く必要があるだろう．ソースコード 3.1 に Python の 3×3 list を用いて，C++ 言語の double 型 3×3 行列を要素にもつ vector を作成する場合の仮想的な API を示す．1 行目から 9 行目にて Python の 3×3 list を作成している．11 行目から 17 行目の to_cpp 関数は Python の二次元リストを受け取り，対応する行数及び列数で要素は double である C++ の行列を作成して返す．19 行目の cpp_vector 以下により double の 3×3 行列のポインタを要素にもつような vector を作成している．cpp_matrices は C++ 上の vector オブジェクトを指す Python の変数である．以降の行ではこれらを用いて，Python の list を C++ の配列に変換した後，それを vector オブジェクトへ逐次追加している．

この値に対する型変換の規則の記述手法には，関数呼び出し時にユーザーが柔軟に型の指定を行うことができるという利点がある．REPL を用いた開発時など，実行時に型変換規則を指定したい際に便利である．一方で，豊富な型を有する言語間の FFI を考えている場合，型変換規則の記述量が多くなるという問題がある．ソースコード 3.1 の例をとっても一つの他言語のデータを作るだけでもこれだけ記述が必要になっている．一つの他言語関数に渡す値は複数あることが多い上，何度も他言語関数を呼び出すコードを書くことを考えると，一つ一つのホスト言語のデータに対して型変換規則を記述するのは大変であると言える．

3.2 関数に対する型変換の規則の記述

関数に対する型変換規則の記述とは，使用する全関数について型変換の規則をユーザーがあらかじめ記述しておくという手法である．関数ごとに引数として渡されたホスト言語のデータをライブラリ言語のどの型へと変換するかについて，ユーザーが静的ないし動的に指定する．ほとんどの関数では，同じ引数なら異なる値が渡されたとしても同じ変換規則で変換を行ってよい．この観察に基づくと，関数の引数ごとに変換規則を指定し，呼び出しごとに自動的に変換を行うようにすることで，ある程度は型変換規則の記述を減らすことが可能になる．依然として，使用することになる全他言語関数に対し型変換規則の記述をユーザーが全て手動で行う必要があり，記述量が多く大変である．

関数に対する型変換規則の記述を行う FFI の一種として，OMG（Object Management Group）により仕様が策定された CORBA（Common Object Request Broker Architecture）[14] という技術がある．CORBA は実際には様々な言語の組み合わせの FFI の構築を容易にするため，ホスト言語の型とライブラリ言語の型を直接変換するのではなく，ある共通の中間言語 IDL（Interface definition language）を介して型変換を行う．その結果ホスト言語の型と IDL の型，IDL の型とライブラリ言語の型という形でデータの型変換が行われ，ホスト言語から他言語関数を使用することができる．CORBA を使用する際，ユーザーは使用する他言語関数ごとに C 言語のプロトタイプ宣言のようなものを静的に記述する必要がある．ここで記述される型は IDL の型である．他言語関数の型は記述されているため，このプロトタイプ宣言をみれば CORBA のシステムが引数，戻り値について型変換を行えるというものであ

ソースコード 3.1. Python の 3×3 list を用いて，C++ 言語の double 型 3×3 行列を要素にもつような vector を生成する例

```
translate_1 = [[1.0, 2.0, 3.0],
               [4.0, 5.0, 6.0],
               [7.0, 8.0, 9.0]]
translate_2 = [[1.0, 0.0, 0.0],
               [0.0, 1.0, 0.0],
               [0.0, 0.0, 1.0]]
translate_3 = [[0.866, -0.5, 1.73],
               [0.5, 0.866, 1.0],
               [0.0, 0.0, 1.0]]

def to_cpp(matrix):
  n, m = len(matrix), len(matrix[0])
  result = ((cpp_double * m) * n)()
  for i in range(n):
    for j in range(m):
      result[i][j] = matrix[i][j]
  return result

cpp_matrices = cpp_vector[POINTER((cpp_double * 3) * 3)]()
cpp_matrices.push_back(to_cpp(translate_1))
cpp_matrices.push_back(to_cpp(translate_2))
cpp_matrices.push_back(to_cpp(translate_3))
```

る．これは関数に対して型変換規則の記述を行っているとみなせる．

図 3.1 に CORBA のアーキテクチャーの模式図を示す．クライアントコード，サーヴァントコードは任意の CORBA 対応言語（C, C++, Java など）で書かれたアプリケーションコードである．各アプリケーションコードでは IDL とのインターフェースが記述されている必要がある．各関数に対する型変換規則が記述された IDL ファイルをもとに IDL コンパイラが，ホスト言語側，ライブラリ言語側で型変換および通信を担うコード（Proxy, Skelton）を生成する．重要な点は関数に対して型変換規則を記述しさえすれば，ホスト言語から他言語関数にデータを送れるようになるということである．

以下，具体的に C++ で書かれたクライアントコードから，Java で書かれたサーヴァントコードを使用する例を見ていく．ソースコード 3.2 に OMG IDL を用いたリモートインターフェースの記述の例を，ソースコード 3.3 に Java による IDL インターフェースの実装の例を示す．ソースコード 3.2 ではソースコード 3.3 中の Java の各関数に対して，プロトタイプ宣言のような形で変換されるべき IDL の型を記述している．Java の package, signature interface, java.lang.String といった型がそれぞれ IDL の module, interface, string といった

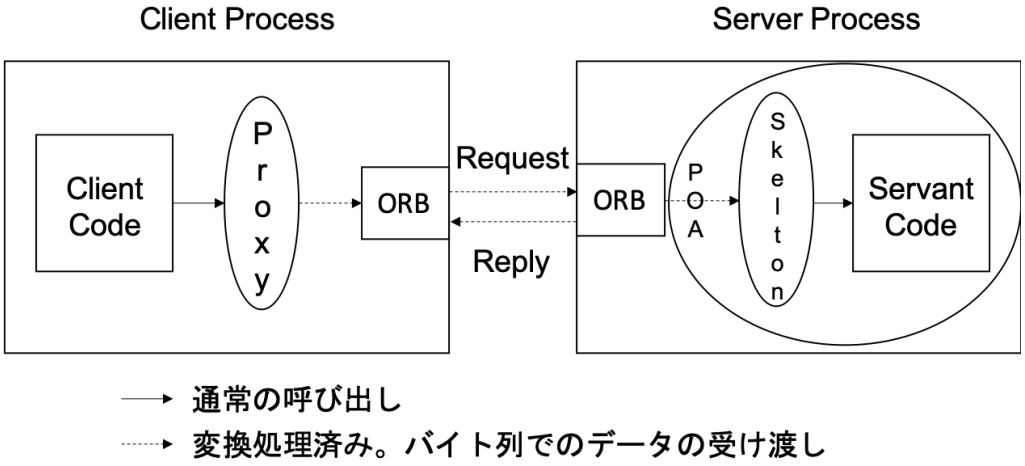

図 3.1. CORBA のアーキテクチャー

ソースコード 3.2. Java オブジェクトに対する IDL を用いたインターフェースの記述

```
1 module HelloApp {
2     interface Hello {
3         string sayHello(string);
4         oneway void shutdown();
5     };
6 };
```

型に対応していることがわかる．このように記述することで，同様にしてインターフェースが記述された CORBA オブジェクトが sayHello メソッドを呼び出して結果を受け取ったり，shutdown メソッドを呼び出して手続きを終了させたりすることができるようになる．

この手法には一度宣言時に記述しさえすれば他言語関数の使用の際に型変換規則の記述の必要がなくなり，値に対する型変換の記述よりは記述量が減るという利点がある．一般に使用する他言語関数の数は他言語関数に送り込む値の数よりは少ないと言える．他言語関数を複数回呼ぶ際や，宣言的に型の記述をしたい際に便利である．しかしながら，内部で使用することになる全他言語関数に対し型変換規則の記述をユーザーが全て手動で行う必要があり，やはり記述量が多く大変である．

3.3 Python/EusLisp FFI 及び従来の型変換規則の記述法の実装

本研究では既存手法の課題の確認および提案手法の有用性の確認の土台として，一から Python/EusLisp FFI の構築を行った．本節では，これまでの節で述べた 2 つの既存手法を用いて型変換の規則の記述を行う場合の実装を，Python/EusLisp FFI を例として具体的に見ていく．続く第 3.4 節では，この実装を用いて，既存の型変換規則の記述法ではやはり型変換

ソースコード 3.3. Servant プログラムの抜粋 (Java)

```
1  // https://docs.oracle.com/javase/jp/1.5.0/guide/idl/
       tutorial/GSserver.html
2  import HelloApp.*;
3  // 本来org.omg.CosNamingやorg.omg.
       CORBAなどのモジュールもimportする
4
5  class HelloImpl extends HelloPOA {
6    private ORB orb;
7    public void setORB(ORB orb_val) {
8      orb = orb_val;
9    }
10   public String sayHello(String name) {
11     return "\nHello, " + name + "!\n";
12   }
13   public void shutdown() {
14     orb.shutdown(false);
15   }
16 }
17
18 public class HelloServer {
19   public static void main(String args[]) {
20   // 初期化，参照の取得などを本来行う
21   // クライアントを待つ処理を行う
22   }
23 }
```

規則の記述量が多くなってしまうという課題があることを確認することになる．なお，全実装の詳細を記載することは分量の都合上不可能なため，大まかな機能及び実装の流れに限定して説明を行う．

3.3.1　Python/EusLisp FFI の概観とデザイン

第 2.1 節で見たように FFI の形態はさまざまである．今回はホスト言語とライブラリ言語側でそれぞれプロセスを立ち上げ，ソケットを介して他言語関数呼び出しを行うという手法をとった．ソケットを介して通信する手法には，容易にネットワークを介した他言語関数呼び出しを行えるように拡張可能であるという利点がある．

第 1 章および第 2 章で述べたように FFI は両言語間で有するデータ型の差異を吸収する．特定のクラスのインターフェースを持たない汎用のプロキシーオブジェクトを生成する場合を除いて，型変換規則が必要なのであった．今回定めた Python から EusLisp へのデフォルト

ソースコード 3.4. Client プログラムの抜粋 (C++)

```
1  #include "Hello.h"
2  // 本来 OB/CORBA.h や OB/Cosnaming.h などのヘッダーも
       include する
3  int main(int argc, char** argv)
4  {
5    CORBA::ORB_var orb;
6    try {
7      // 初期化，参照の取得などを本来行う
8      // "manager" へのメソッドコールは Java プロセス中の対
           応するオブジェクトへのメソッドコールとなる
9      CORBA::String_var response_string;
10     CORBA::Char* name;
11     name = CORBA::string_alloc(10);
12     strcpy(name, "PRO2019-5")
13     response_string = manager->sayHello(name);
14     cout << response_string.in() << endl;
15     // "Hello, PRO2019-5 !" と表示される
16
17     manager->shutdown();
18   } catch(const CORBA::Exception& e) {
19     cerr << e << endl;
20   }
21   // 終了処理を本来行う
22   return 0;
23 }
```

の型変換の規則を 3.1 に，EusLisp から Python へのデフォルトの型変換の規則を表 3.2 に示す．今回作成する FFI では引数に関してはコピー渡しを，戻り値に関しては参照渡しを基本とした．

他言語関数呼び出し時の引数については，EusLisp のオブジェクトを指すプロキシーオブジェクト以外は全て，対応する EusLisp のオブジェクトにコピー変換される．ここで型変換規則が必要となる．引数に関してコピー渡しを基本とした理由は，ライブラリ言語である EusLisp には行列やベクトルを引数にとる関数が多く，ライブラリ言語側に実体を生成することが実行速度の都合上好ましいと考えられるからである．

一方, 他言語関数呼び出しの戻り値については, 数値および真偽値以外は全て, その EusLisp オブジェクトを指す，特定のクラスのインターフェースを持たない汎用のプロキシーオブジェクトに変換される．このプロキシーオブジェクトに対しても Python の型を指定し，その型のもつインターフェースを継承させることは可能であるが，今回は行っていない．この場合，戻

り値に対する型変換規則は与えないでよいが，一般にはプロキシーオブジェクトを使用する場合も型変換規則を与える必要がある．なお，この EusLisp のオブジェクトへのプロキシーオブジェクトを，デフォルトの型変換規則に従って対応する Python のオブジェクトへと変換する場合，to_python メソッドを呼ぶことで変換が行われるように定めた．

また，FFI を通して他言語ライブラリを使用する際，できるだけホスト言語の場合と同様に扱えるようにしたい．ライブラリ言語には関数や変数，クラスといったオブジェクトが存在するが，それを指すプロキシーオブジェクトがホスト言語の環境に自動的に作成されるようにする．他言語関数を指すプロキシー関数に対し関数呼び出しを行えば，内部で引数結合や型変換を行い他言語関数呼び出しがなされ，適切に評価結果が返るようにする．他言語環境の変数を指すプロキシー変数に対し評価を行えば，内部で他言語環境での変数参照が行われ，適切に評価結果が返るようにする．

実装の節に移る前に，より Python/EusLispFFI を具体的に捉えるため，実装された FFI の使用例を図 3.2 に示す．上に示されているものが EusLisp から EusLisp オブジェクトである robot に対しメソッドコールを行う例で，下に示されているものが Python からその機能を使用する場合の例である．上のコードでは robot オブジェクトの持つ rarm メソッドの引数として:inverse-kinematics なる keyword と，#f(100.0 0.0 50.0) なる float-vector を渡している．しかし Python にはこのような型が存在しないため，型の変換が必要である．この例では Python の str を EusLisp の keyword へ，Python の list を EusLisp の float-vector へと変換している．戻り値として EusLisp の float-vector が返るが，それを Python の list へと変換して受け取っている．

3.3.2　最低限の通信層とブートコードの実装

まずはホスト言語である Python 環境とライブラリ言語である EusLisp 環境の間でやりとりを行えるよう，通信層を整備する．両言語間でソケット通信を行う環境を整えるため，Python 側には TCP クライアント起動関数 run_python_client，接続関数 connect_to_eus，切断関数 close_socket，EusLisp 側には TCP サーバー起動関数 run-eus-server が必要になるだろう．そして低レベルの通信関数として Python 側，EusLisp 側に write_to_socket，read_socket 関数を用意し，ソケットストリームに対し読み書きできるようにする．この時点で EusLisp にサーバープログラムとして，ソケットから S 式を読み込み，eval を用いて評価を行い，結果を書き込みループするようなコードを実行させれば，超簡易的な他言語環境評価システムが完成する．Python から EusLisp で評価可能な S 式をバイト列の形で送り込めば評価結果を文字列の形で得ることができる．

現段階では手動で Python のクライアントプログラムと EusLisp のサーバープログラムを立ち上げる必要がある．それでは不便なため，ホスト側からライブラリ側をブートする関数があるとよい．Python 側に，EusLisp をサブプロセスとして起動する関数 run_eus_process，サブプロセスを終了する関数 kill_eus_process 関数などを用意すればよい．EusLisp 側には run_eus_process 関数の実行ターゲットとなりエントリーポイントとなるファイルを用意し，

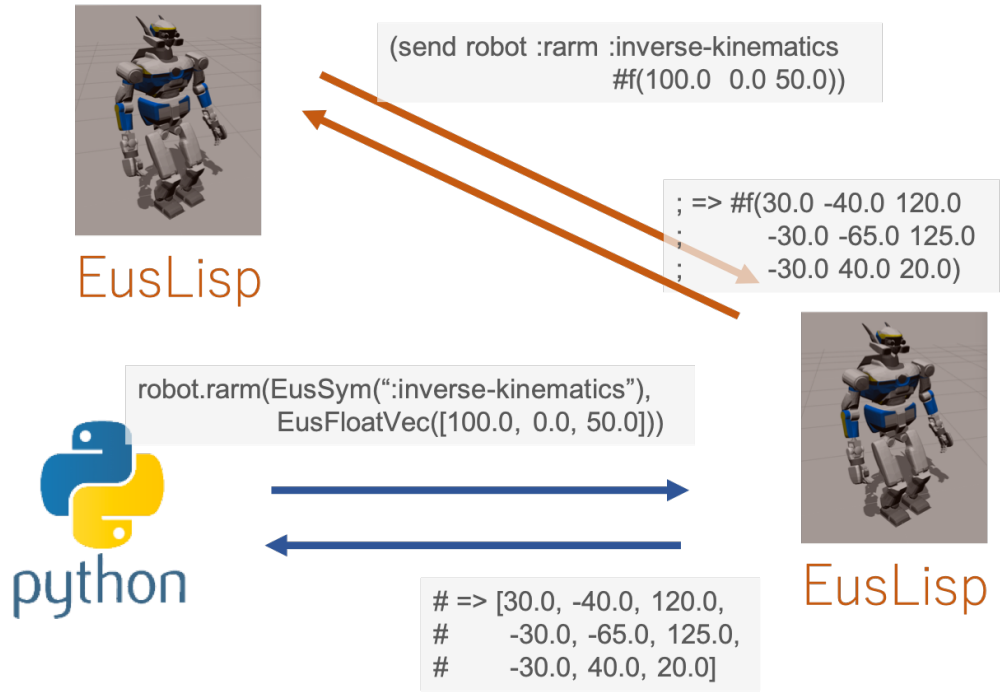

図 3.2. 実装した Python/EusLisp FFI における型変換の様子

そこでは run-eus-server の実行及び前述のループプログラムの実行を行えばよい．これで Python 側から EusLisp 環境をフックして起動し，S 式を評価させて結果を受け取ることが可能になった．

3.3.3 通信層の洗練

込み入った実装に取り掛かる前に，この段階で両言語間でのプロトコルを決め，通信層を洗練させる．どのような形でコマンドを送って，受け取る側でどれだけの処理を担当するかは自由なのであるが，今回は相手が本体データをそのまま eval すればことが済むように，送り手側で成形・コマンドの埋め込みをするという規約を設けた．これにより両言語の構築するべきコマンド列が明確になる．この通信層を介して送るデータについて，以下のような要望が生じると考えられる．

- ただ相手に評価させるのみでエラーや戻り値を受け付けない（発生したエラーは握り潰させる）(no-request, 便宜上モード n とする)
- 相手に評価させ，結果を自分が評価可能な形の文字列表現でコピー返しさせる（copy-request, モード c とする）
- 相手に評価させ，結果を保存させプロキシー生成コマンドの形で返させる（proxy-request, モード p とする）
- 相手にエラーを知らせる．エラーメッセージの文字列表現を評価させる (error-request,

モード e とする)

これらに対応できるよう，通信層を介して送るデータ表現のプロトコルを決定した．最初の 4byte は全体のデータサイズを示すヘッダー領域，次の 2byte はモードを知らせるヘッダー領域，続く任意長がデータ領域であると定めた．Python, EusLisp 両環境にこれらのプロトコルを満たすようなソケットの読み書きの関数 read_with_mode，write_with_mode を作成して通信層の実装は終了となる．次小節以降はこれを利用して機能を組み立てていく．

3.3.4　コピー渡しコピー返し機構 eval_foreign_copy の実装

前節までは低レイヤの通信層の実装だったが，ここからは一段階レイヤが上がる．この小節では表 3.1，3.2 に示したデフォルトの型変換規則に応じた，データをコピー渡しコピー返しのみでやり取りする， Python → EusLisp → Python の流れを実現する評価システムを作る．

まずコピー渡し，コピー返しのために必要となる評価結果を相手言語がそのまま評価できるような文字列データに変換する関数 eusobj_from_obj, pyobj-from-obj を作成する．内部で型によるディスパッチを行い特定の型に対する変換関数を書いていくことで実装が完了する．

また，通信層ではプロトコルを決定したが，この 4 つのモードに対応した処理を行う関数 process_packet を Python, EusLisp 両環境に作成する．これは read_with_mode を用いてモード文字列，評価するべきデータを取得し，評価を行う．モード文字列及び評価時に生じたエラーに応じて以下の対応をとる．

- エラーが生じて，モードが n でないとき．相手が待っているため，エラー文字列として評価されるようなデータを e モードで送る．モードと None を返す．
- エラーが生じて，モードが n のとき．相手は待っていないため，エラーを握り潰す．モードと None を返す．
- エラーが生じず，モードが n のとき．モードと評価値を返す．
- エラーが生じず，モードが e のとき．評価結果は相手方から送られてきたエラー文字列である．適切なエラーを挙げる．
- エラーが生じず，モードが c のとき．評価結果を相手言語 がそのまま評価できるような文字列データに変換構築し eval_foreign_no_return で送り返す．モードと評価値を返す．

以上により通信層より一段階上のデータの送受信関数を両言語の環境に書くことが可能となる．エラーを知らせるための関数 eval_foreign_error は，単純にモード e でデータを送信する関数である．ただ相手にコマンドを実行させるための関数 eval_foreign_no_return は，単純にモード n でデータを送信する関数である．相手にデータを評価させ，戻り値デフォルトの型変換規則に従いコピー返しさせる関数 eval_foreign_copy はモード c でデータを送信し，process_packet をループする．n で返ってきたら (コールバックなどではなく) 最初の問いかけへの応答であり，値を戻り値として返す．次小節以降はこれを利用して機能を組み立てていく．

3.3.5 コピー渡しコピー返しによる FFI

前節まででは他言語関数呼び出しはまだ行えなかった．なぜならば関数呼び出し時に解消すべき差異のうち，データ型以外の差異が解消されていなかったからである．この節ではそれらを解消し，コピー渡しコピー返しによる他言語関数呼び出しを行えるようにする．

Python 側から自然な EusLisp 他言語関数呼び出しを実現するには，他に解決するべき主要な差異として

- 関数引数の与え方の違い
- 関数呼び出しの方法の違い

が挙げられる．図 3.3 にどのように差異を吸収するかを示す．EusLisp のパッケージ EUS_LIB にある他言語関数 F を使用したいとする．EUS_LIB.F([100, 0, 50]) と関数呼び出しをしたら，(EUS_LIB::F (list 100 0 50)) という S 式が EusLisp の環境に送られて欲しい．既に Python の list を EusLisp の cons に評価されるような文字列へ変換する機能はできているが，このように両言語で関数の引数の与え方，関数の呼び出し方法が全く異なり，さらなる実装が必要であるとわかる．上記のような他言語関数呼び出しの実現には

1. EusLisp の存在するパッケージを探索し，これと対応するプロキシーモジュールを Python 側に作成する
2. そのパッケージに存在する関数を探索し，これと対応するプロキシー関数を Python 側に作成する
3. そのプロキシー関数は呼び出された時に，パッケージ名，関数名，引数部を適切に変換し埋め込んだコマンドを作成し，eval_foreign_copy を実行するようにする

という手順を踏む必要がある．上二つについては既に作成した eval_foreign_copy 及びリフレクション [2] を用いて EusLisp 側のオブジェクトの情報を取得できるため容易である．一例として，eval_foreign_copy("(mapcar #'package-name (list-all-packages))") を実行すればただちに存在するパッケージの大文字表記の Python リストを得ることができる．最後の項目について，引数部の変換がまだ達成されていない．Python の場合関数引数は通常引数についてはタプルにまとめられ，キーワード引数については辞書にまとめられて関数へ渡される．これを解析し，適切に eusobj_from_obj 関数を適用した結果を結合した文字列を生成する引数処理関数 translate_tuple, translate_dict を作成する．これで晴れてコピー渡しコピー返しの FFI が構築された．次小節以降はこれを利用して機能を組み立てていく．なお、以降は簡単のため他言語パッケージについては適宜省略し、他言語関数についてのみ注目する．

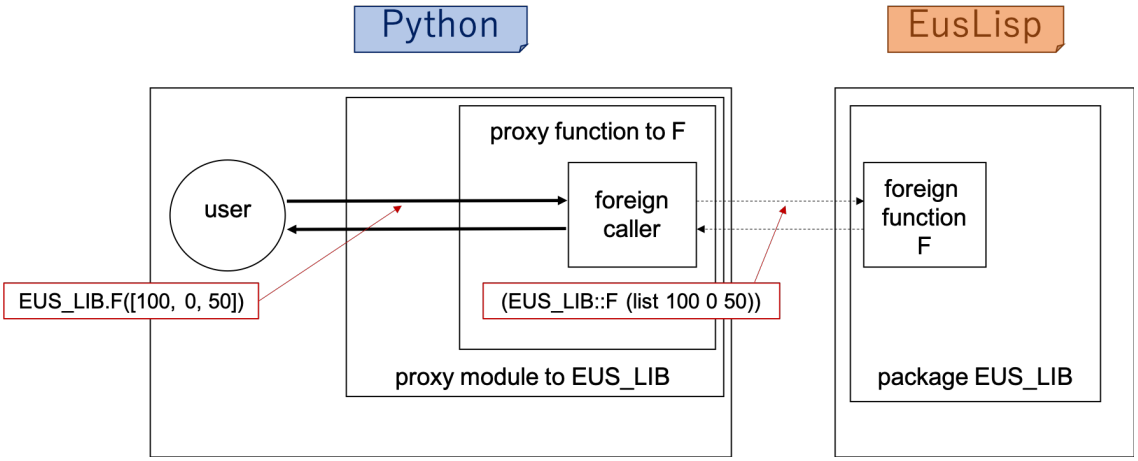

図 3.3.

3.3.6　コピー渡し参照返し機構 eval_foreign_proxy の実装

本論文で必要となる型変換の話はコピー渡しコピー返しの FFI で出切っているため，この小節及び次節は飛ばして構わない．まずはモード p であることのみが eval_foreign_copy と異なる関数 eval_foreign_proxy を両言語に作成する．そして両言の process_packet において，モード p のときの挙動を追記する．両言語に参照オブジェクトを管理するテーブルをあらかじめ用意しておく．モード p で送り込まれたデータの評価結果に固有の番号を与え，テーブルに束縛する．そして固有番号を属性として持つようなプロキシーオブジェクト生成コードを埋め込み eval_foreign_no_return で送りこみ，モードと評価値を返せばよい．他言語関数呼び出し時に eusobj_from_obj, pyobj-from-obj 関数が引数を解析するが，このようなプロキシーオブジェクトが存在した場合，その属性を見て固有番号を特定し，相手言語環境でテーブルを検索して参照先の実体を取得するようなコードを埋め込めば良い．

なお，このままではテーブルに登録されたオブジェクトはそれを指すプロキシーが消滅しても登録され続けてしまう．これを解決するには，プロキシーオブジェクトのファイナライザとして，相手言語のテーブルから実体を削除するようなコマンドを eval_foreign_no_return を利用して送りこむコードを記述しておけばよい．現在 EusLisp には弱参照の機構やファイナライザの機構は存在しないが，*gc-hook*を用いて模擬的なファイナライザを実装することによりこの問題の解決をした．

3.3.7　コピー渡し参照返しによる FFI

第 3.3.5 小節と同様にプロキシー関数を作成していけばよい．この際，プロキシー関数は eval_foreign_proxy でもってコマンドを送り込むような関数として生成する．関数以外にも他言語パッケージ内の変数のプロキシーや（コンストラクタのプロキシーが定義された）クラス

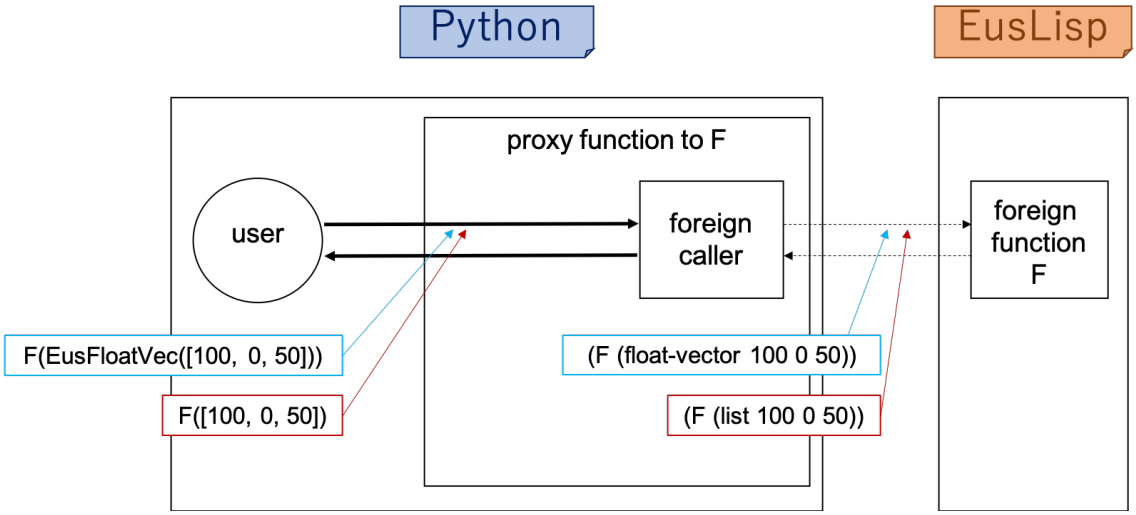

図 3.4.

のプロキシーも生成したのだがそれらの機能は本論文では割愛する．

3.3.8 値に対する型変換規則の記述の実装

作成した Python/EusLisp FFI に対して，値に対する型変換の規則の記述の実装を行った．API を表 3.3 に示す．値に対する型変換規則の記述を行うには希望の型の変換先と対応する型コンストラクタ（第一列）を Python のオブジェクト（第二列）に適用する．すると，その変換先の型を示すタグを属性として持つオブジェクト（第三列）が生成される．他言語呼び出しの際にはその型タグを見て変換先の型を FFI が特定し，EusLisp 側にて対応するオブジェクト（第四列）をコピー生成するようなコードを埋め込んで他言語関数呼び出しがなされるようにした．一連の流れの模式図を図 3.4 に示す．図は他言語関数 F に対し，値に対する型変換規則の記述がある引数で呼び出した場合と，ない引数で呼び出した場合を示している．値に対する型変換規則の記述の実装のためには，他言語関数呼び出し時に eusobj_from_obj, pyobj-from-obj 関数が引数を解析するが，この時に型タグを指定されたオブジェクトがあった場合にデフォルトの型変換規則ではなく型タグを見て変換先の型を決定し文字列表現を構築するようにすればよい．

3.3.9 関数に対する型変換規則の記述の実装

作成した Python/EusLisp FFI に対して，関数に対する型変換の規則の記述の実装を行った．関数に対する型変換記述を行うには，set_params という関数に対し，記述のターゲットとなる他言語関数と，希望の引数の変換先の型に対応する型タグを列挙する．結果，他言語関数のプロキシー内のマッピングテーブルに引数の変換規則を記録される．他言語呼び出しの際にはまずマッピングテーブルを見て型タグが登録されている場合それを適用するようにする．以

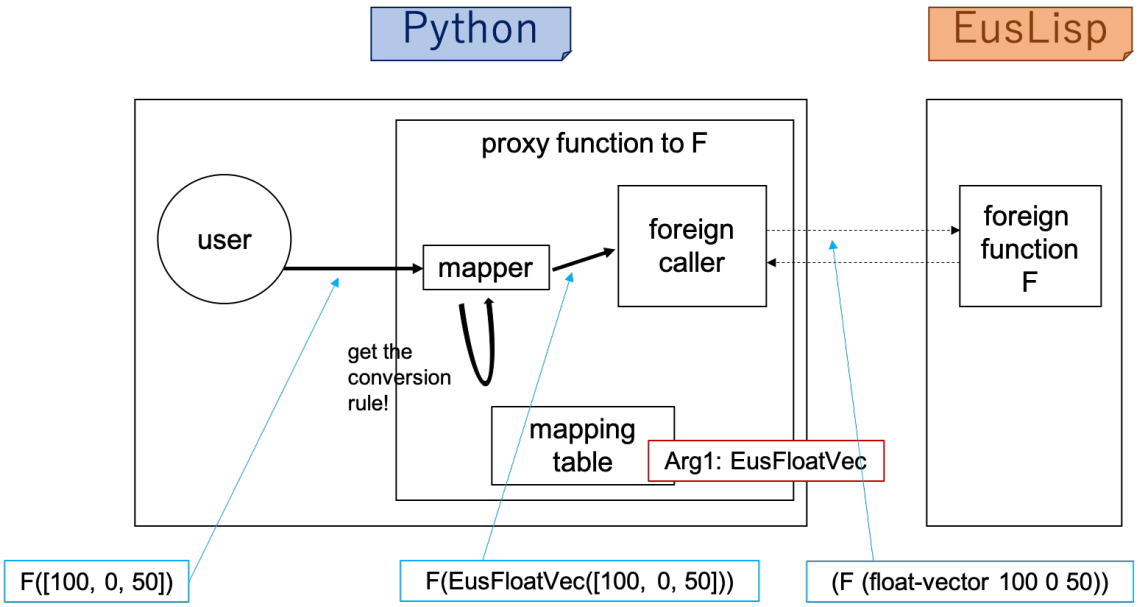

図 3.5.

降の流れは以前同様である．一連の流れの模式図を図 3.5 に示す．図は他言語関数 F に対し，関数に対してあらかじめ第一引数を float-vector へ変換するよう設定してから呼び出した場合を示している．以上のように，関数に対する型変換規則の記述の実装のためには，プロキシー関数内部に引数の変換先を登録するためのマッピングテーブルを用意し，他言語関数呼び出し時にはそれを検索して適切に型タグを設定するようにすればよい．

3.4　従来の型変換規則の記述法の限界

この節では，既存手法（値に対する型変換規則の記述法及び関数に対する型変換規則の記述法）は共に記述量が多くなってしまうという課題を抱えていることを，実装した Python/EusLisp FFI を用いて確認する．

まず初めに，型変換規則を記述する必要がない例として，リストをうけとり逆順にして返す EusLisp の reverse 関数を Python から使用することを考える．Python 側のコードはソースコード 3.5 のようになる．第 2 行で他言語関数 LISP:reverse の Python バインドである LISP.reverse が Python の list を引数として受け取っている．この他言語関数の Python バインドである LISP.reverse は，引数として Python オブジェクトを受け取った時，それに対応する EusLisp オブジェクトをコピー生成し，関数実行のための適切な S 式を EusLisp 側に送り込むことを内部で行なっている．デフォルトの型変換の規則によれば Python の list はコピーされ EusLisp の nil-terminated cons へと自動的に変換されるため，このような場合は引数として Python の list を渡すだけで EusLisp 側で正しく引数を受け取り，型エラーを起こすことなく関数の処理が完了する．LISP.reverse 関数コールの戻り値は EusLisp の cons オブジェ

クトのプロキシーである．ここでは Python の list に変換するために to_python なるメソッドを呼んでいる．このような場合はそもそも型変換規則をユーザーが FFI に与える必要がなく，例外的に問題が起きない．

しかしながら，大半の例においてはユーザーが型変換規則を何らかの形で FFI に与える必要があり，記述量が多くなってしまうという問題が浮上する．ソースコード 3.6 のような 2 つの行列を引数にとり，それらの 0, 0 成分の和を求める EusLisp の関数を Python から使用することを考える．デフォルトの型変換の規則からでは EusLisp の array は生成できないため，ユーザーが明示的な型変換の規則の記述を行う必要がある．第 3.1 節や第 3.2 節で示したように，型変換の規則の記述には値に対する記述と関数に対する記述がある．

前者の方法で Python から EusLisp の sum-of-0-0-element-of-arr 関数を呼び出すコードはソースコード 3.7 のようになる．引数部の Python の list に対し EusArray なる型タグの記述を与えることで型変換を行なっている（第 3.3.8 小節参照）．この型変換コンストラクタは引数として第二列のような Python オブジェクトを受け取った時，そのオブジェクトの変換先の型を型タグとして記録する．FFI は他言語関数呼び出しの際，引数のオブジェクトの型タグをを見て変換先の型を決定し，適切に対応するオブジェクトを生成するコードを引数部に埋め込む．その結果 EusLisp 側では正しく array として引数を受け取り，型エラーを起こすことなく関数の処理が終了する．値に対する型変換規則の記述を行う手法では型コンストラクタを何度も呼ぶ必要が出てしまう．他言語関数の引数は複数あることもあるし，他言語関数を複数回呼ぶことを考えると型変換規則のユーザー記述量は多い．

後者の方法で Python から EusLisp の sum-of-0-0-element-of-arr 関数を呼び出すコードはソースコード 3.8 のようになる．冒頭の set_params という関数は他言語関数に対して型変換規則の登録を行う関数である（第 3.3.9 小節参照）．このようにユーザーが関数呼び出し前に，関数に対し型変換の規則を一度記述すれば，以降はその通りに引数が型変換されて，結果の受け取りができるようになる．第 3 行以降で他言語関数の引数として Python の list が渡されているが，内部的に array へと型変換が行われてから EusLisp 側へ渡されるため，この場合も型エラーを起こすことなく関数の処理が終了する．この例において値に対する型変換規則の記述のときよりも記述量が減っていることが見てとれる．一方，ソースコード 3.9 に示したような，内部で他言語関数を複数使用するような関数を使うことを考える．使用する他言語関数が増えれば増えるほど型変換規則の記述が必要となるため，ユーザーが記述する必要のある型変換規則の量は依然として多いと言える．

表 3.1. Python から EusLisp へのデフォルトの型変換規則

Python	EusLisp	Description
int	integer	
float	float	
None, False	nil	
True	t	
string	string	
list, tuple, xrange	list	
set, frozenset, dict	hash-table	
any other object	N/A	TypeError

ソースコード 3.5. EusLisp の reverse 関数を Python から呼び出す例

```
1 # LISP は EusLisp の標準関数が入ったパッケージ
2 LISP.reverse([1,2,3]).to_python()
3 # => [3,2,1]
4
5 type(LISP.reverse([1,2,3]))
6 # => <class 'pyeus.cons'>
7 # proxy object that points to cons in EusLisp
```

ソースコード 3.6. 2 つの配列の (0, 0) 成分の和を求める EusLisp 関数の例

```
1 (in-package  "EUS_LIB")
2
3 (defun sum-of-0-0-elment (arr1 arr2)
4   (+ (aref arr1 0 0) (aref arr2 0 0)))
```

ソースコード 3.7. EusLisp の sum-of-0-0-element 関数を，Python から値に対する型変換規則の記述とともに呼び出す例

```
1 result1 = EUS_LIB.sum_of_0_0_element(EusArray
    ([[1,2],[3,4]]), EusArray([[5,6],[7,8]]))
2
3 result2 = EUS_LIB.sum_of_0_0_element(EusArray
    ([[9,10],[11,12]]), EusArray([[13,14],[15,16]]))
4
5 result3 = EUS_LIB.sum_of_0_0_element(EusArray
    ([[17,18],[19,20]]), EusArray([[21,22],[23,24]]))
```

表 3.2. EusLisp から Python へのデフォルトの型変換規則

Class of the object (EusLisp)	Class of the object (Python)	after to_python conversion
integer	int	N/A
float	float	N/A
nil	None	N/A
t	True	N/A
symbol	class 'pyeus.symbol'	string
function symbol	class 'pyeus.compiled_code'	string
string	class 'pyeus.string'	string
(non-nil-terminated) list	class 'pyeus.cons'	list
property list	class 'pyeus.cons'	list
list	class 'pyeus.cons'	list
integer-vector	class 'pyeus.integer_vector'	list
float-vector	class 'pyeus.float_vector'	list
bit-vector	class 'pyeus.bit_vector'	list
vector	class 'pyeus.vector'	list
array	class 'pyeus.array'	list
hash-table	class 'pyeus.hash_table'	dict
pathname object	class 'pyeus.pathname'	str
any other instance of 'MyClass'	class 'pyeus.MyClass'	N/A
class object (not an instance)	class 'pyeus.metaclass'	N/A

表 3.3. Python から EusLisp への明示的な型変換に用いるコンストラクタ

Constructor (Python)	Args (Python)	Class (Python)	Class (EusLisp)
EusSym	string	pyeus.symbol	symbol
EusFuncSym	string	pyeus.compiled_code	compiled-code
EusStr	string	pyeus.string	string
EusCons	list, tuple, xrange	pyeus.cons	(non-nil-terminated) list
EusList	list, tuple, xrange	pyeus.cons	(nil-terminated) list
EusPlist	list, tuple	pyeus.cons	property list
EusIntVec	list, tuple, xrange	pyeus.integer_vector	integer-vector
EusFloatVec	list, tuple	pyeus.float_vector	float-vector
EusBitVec	list, tuple	pyeus.bit_vector	bit-vector
EusVec	list, tuple, xrange	pyeus.vector	vector
EusArray	list, tuple, xrange	pyeus.array	array
EusHash	dict	class hash_table	hash-table
EusPath	str	pyeus.pathname	pathname object

ソースコード 3.8. EusLisp の sum-of-0-0-element 関数を，Python から関数に対する型変換規則の記述とともに呼び出す例

```
1 set_params(EUS_LIB.sum_of_0_0_element, EusArray, EusArray)
2
3 result1 = EUS_LIB.sum_of_0_0_element([[1,2],[3,4]],
      [[5,6],[7,8]])
4
5 result2 = EUS_LIB.sum_of_0_0_element([[9,10],[11,12]],
      [[13,14],[15,16]])
6
7 result3 = EUS_LIB.sum_of_0_0_element([[17,18],[19,20]],
      [[21,22],[23,24]])
```

ソースコード 3.9. 内部で複数の EusLisp 関数を使用するような関数を，Python から関数に対する型変換の指定とともに呼び出す例

```
# LISP は EusLisp の標準関数が入ったパッケージ
# index が複数入った2次元 vector と array の2引数をうけとり，その index の指す array の要素の和を求める
def sum_indirect(arr, ind_vec):
  sum = 0
  dims = LISP.array_dimensions(arr)
  dim_x = LISP.car(dims)
  dim_y = LISP.car(LISP.cdr(dims))
  num = LISP.length(ind_vec)
  for i in range(num):
    indices = LISP.svref(ind_vec, i)
    ind_x = LISP.car(indices)
    ind_y = LISP.car(LISP.cdr(indices))
    if ind_x < dim_x and ind_y < dim_y:
      sum += LISP.aref(arr, ind_x, ind_y)
    else:
      raise IndexError
  return sum

# sum_indirectを使用
set_params(LISP.array_dimensions, EusArray)
set_params(LISP.svref, EusVec)
set_params(LISP.aref, EusArray)

sum_indirect([[1,2,3], [4,5,6], [7,8,9]], [[0,0], [1,1], [2,2]])
# => (1+5+9=) 15
sum_indirect([[1,2,3], [1,2,3], [1,2,3]], [[0,0], [1,0], [2,0]])
# => (1+1+1=) 3
```

第 4 章

Type Description Helper の提案

本章では型変換規則の半自動的な導出器である Type Description Helper を提案する．これまでに FFI は型変換を行うこと，その規則は自明でないためユーザーが与える必要があること，その記述量が多くなってしまい問題であることを確認してきた．この Type Description Helper は一部の型変換規則を半自動的に導出するため，ユーザーによる記述が必要な型変換規則の量を減らすことができる．Type Description Helper は値に対する型変換規則を導出する Log-based 法と，関数に対する型変換規則を導出する Type-inference-based 法の 2 種類を組み合わせて導出を行う．これらは既存手法に対し表 4.1 のような関係にある．第 3.1 節で説明した値に対する型変換規則の記述を半自動化したものが Log-based 法であり，第 3.2 節で説明した関数に対する型変換規則の記述を半自動化したものが Type-inference-based 法である．どちらから得られる型変換規則も完全なものではないため，一部の導出できない型変換規則はユーザーが手動で記述する必要がある．

4.1 Log-based な型変換規則の導出

Log-based 法は他言語関数呼び出し時のログから動的に型変換規則を導出する．これは型の変換先が定まったオブジェクトを引数に他言語関数呼び出しがなされた場合それを記録し，以降は特に記述がない限りその記録の通りに型が変換されて他言語関数が呼び出され，結果の受け取りができるというものである．

このシステムの模式図を図 4.1 に示す．図中左側がホスト言語の環境で右側がライブラリ言語の環境である．右側の F がターゲットとなる他言語関数であり，それに対応するプロキシ関数が示されている．このプロキシ関数は FFI ライブラリにより自動生成される．プロキシ関数内にあるマッピングテーブルは引数に適用する型変換規則を登録するためのフィールドである．これは既存手法である関数に対する型変換規則の記述で出てきたものと同じものであり，このシステムではそのテーブルに追記がなされる．プロキシ関数内にあるロガーは関数呼び出し時に適宜マッピングテーブルへと型変換規則を登録することを担う．図は使用したい他言語関数のプロキシに対しユーザーが関数呼び出しを行なった時の挙動を表している．点線矢印で示されているものが型指定済みのオブジェクトを引数に関数呼び出しを行った時の振る舞いを

表 4.1. 型変換規則の記述の自動性とその型変換規則の記述先

型変換規則の記述の自動性 ＼ 記述先	値	関数
手動	3.1 節で示した手法	3.2 節で示した手法
自動	Log-based 法	Type-inference-based 法

示し，実線矢印で示されているものがその他のオブジェクトを引数に関数呼び出しを行った時の振る舞いを示す．型指定ずみのオブジェクトとは，ライブラリ言語のプロキシオブジェクト，値に対する型変換がなされ変換先の型のタグが記載されたオブジェクトのことである．

まず型指定ずみのオブジェクトを引数に関数呼び出しが行われた時を見ていく．この場合，図中のロガーがその型情報たちを識別し，マッピングテーブルに記録する．そしてフォーリンコーラールーチンに処理が渡される．引数がライブラリ言語のオブジェクトのプロキシであった場合は，ライブラリ言語側でそのオブジェクトを検索するようなコードが引数部に埋め込まれ，他言語関数呼び出しがなされる．型のタグが記載されたオブジェクトの場合は型のタグに対応するオブジェクトを生成するようなコードが引数部に埋め込まれ，他言語関数呼び出しがなされる．次に型指定済みでない，その他のオブジェクトを引数に関数呼び出しが行われた場合を見ていく．この場合，図中のマッパーがマッピングテーブルを検索する．型変換規則が登録されていなかったならば既存手法と同じくデフォルトの型変換規則を適用し，型のタグを設定する．登録されていたならばその型変換規則を各引数に対して適用し，型のタグを設定する．前述の場合と同様にフォーリンコーラールーチンに処理が渡され，型のタグに対応するオブジェクトを生成するようなコードが引数部に埋め込まれ，他言語関数呼び出しがなされる．

これにより一度ユーザーが型指定済みのオブジェクトを引数に関数呼び出しを行ったら，以降は型変換はその規則に従い自動的に行われる．ユーザーが記述する必要のある型変換規則の量は削減される．REPL 駆動で開発やデバッグを行い，動的に値に対しての型変換規則の記述を行いたい場合に特に便利である．

4.2 Type-inference-based な型変換規則の導出

Type-inference-based 法はソースコードを静的に解析することで型変換規則を導出する．この方法では FFI ライブラリが型推論（正確には第 6 章で説明するように，包括的な型検査はせず型の上界を求めるアルゴリズムである）をもとに，考えられる型変換規則の候補を自動的に絞る．絞られた候補を元に型変換をするほか，ユーザーが型指定済みのオブジェクトを引数に他言語関数を呼び出した際にその引数の型に明らかな誤りがないか検証する．このシステムの模式図を図 4.2 に示す．基本的な構成は当然図 4.1 と共通だが，ロガーの代わりにチェッカーが，そして新たに型の上界を記録しておくフィールド possible arg-types が存在すること，推論器が存在することが大きく異なる．そして推論器が動作するタイミングは他言語関数呼び出し時ではなく，FFI ライブラリの機能により他言語ライブラリをロードする時である．この型推論器は他言語ライブラリに対し型推論を行い，各関数の引数の型の上界を取得する．

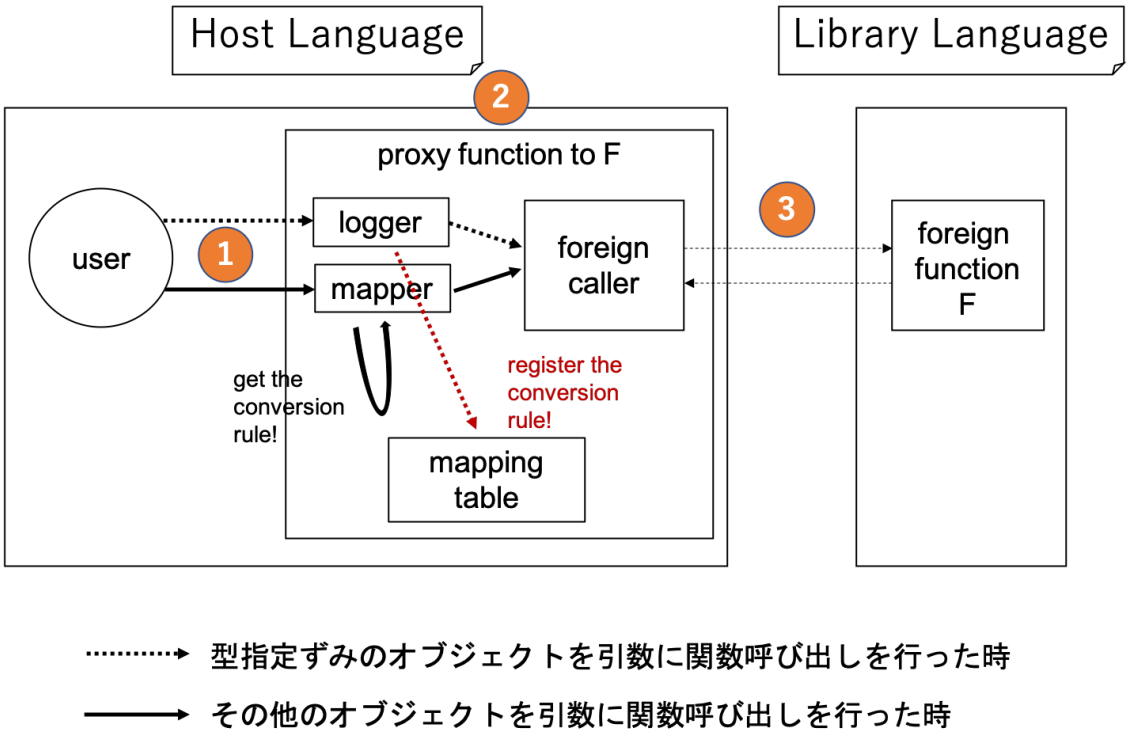

図 4.1. Log-based な型変換規則の導出の模式図

取得した結果は対応するプロキシ関数の型の上界のフィールである possible arg-types に記録する．そして上界がユニオンタイプなどになり変換先が複数考えられる場合，そのうちの一つを選択し，マッピングテーブルに引数の型変換規則として登録する．

関数呼び出し時の基本的な挙動は Log-based 法の場合と似ている．型指定済みでない，その他のオブジェクトを引数に関数呼び出しが行われた場合の挙動は Log-based 法の時と等しい．型指定ずみのオブジェクトを引数に関数呼び出しが行われた場合の挙動は少し異なるため見ていく．この場合，図中のチェッカーがその型情報たちを識別し，型の上界を示す possible arg-types と比較し検証を行う．これにより，ライブラリ言語で実行をしたら型エラーを起こしてしまうようなユーザーによる無効な型変換規則の記述に対し，ホスト言語側で事前に型エラーとして弾くことができる．以降の流れ Log-based 法のときとは等しい．フォーリンコーラールーチンに処理が渡され，引数がライブラリ言語のプロキシであった場合はライブラリ言語側でそのオブジェクトを検索するようなコードが引数部に埋め込まれ，型のタグが記載されたオブジェクトの場合は型のタグに対応するオブジェクトを生成するようなコードが引数部に埋め込まれ，他言語関数呼び出しがなされる．

これにより引数の型が明確な他言語関数に関しては型変換が自動的に行われる．ユーザーが記述する必要のある型変換規則の量は削減される．前節の手法は型変換規則の記述量を削減することはできるが，やはりその型変換規則自体の信頼性に関してはユーザーに任せていた．対しこの手法は記述量の削減に加えて型変換規則の信頼性の問題に対しても部分的に取り組むこ

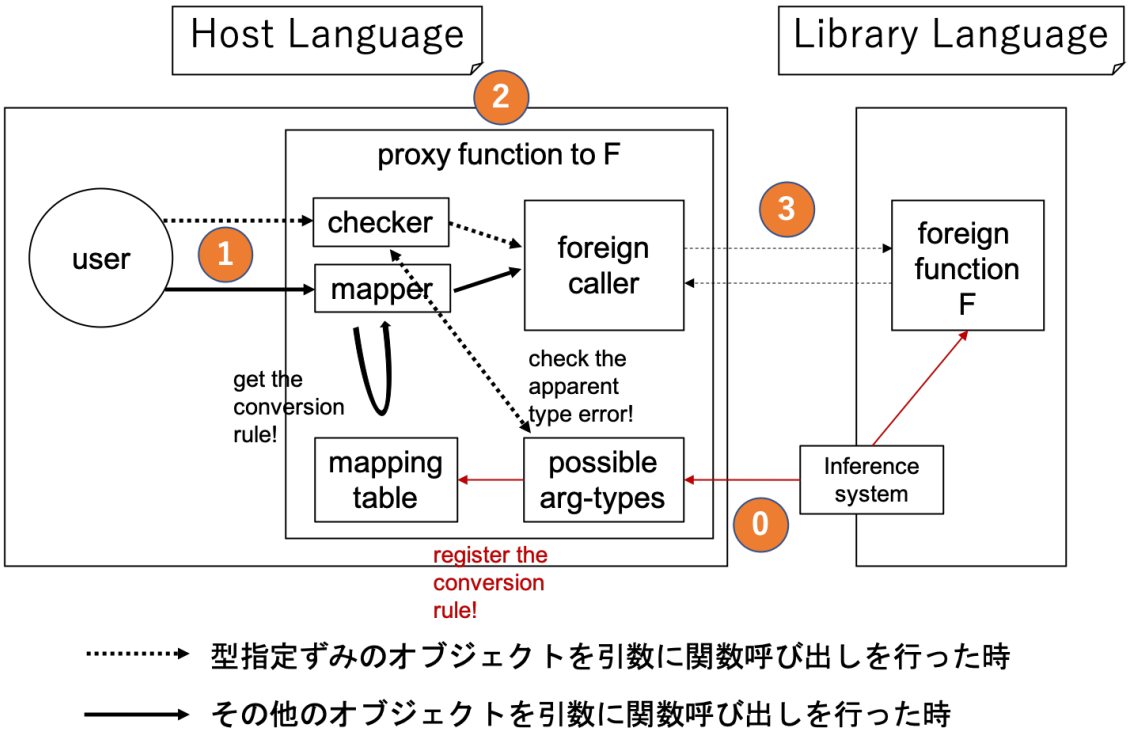

図 4.2. Type-inference-based な型変換規則の導出の模式図

とができる.

第 5 章

Python/EusLisp FFI における提案手法の実装

本研究では，第 4 章で提案した Type Description Helper の有用性を実際に確認するため，第 3.3 節で製作した Python/EusLisp FFI へと実装を行った．特に Type-inference-based 法については実装の内容が，これが導出する型変換規則の量に直結する．そのため本章では提案手法の実装の詳細[*1]について説明を行う．

5.1 Log-based 法の実装の詳細

Log-based 法は他言語関数呼び出し時のログから動的に型変換規則を導出することを第 4 章で述べた．Log-based 法の実装は明快である．単純には他言語関数呼び出し時に，引数に型タグを持つオブジェクトがきたかどうかを判定する機構及びその型タグを登録する機構があればよい．ただし型タグを持つコンテナオブジェクトが他言語関数呼び出しの引数としてきた場合に，その要素が型タグを持っていないか再帰的に調べ，記録する必要があることに注意する（この仕組みがないと，float-vector を要素に持つ vector といった型の，型変換規則の記述を記録することができない）．「ユーザーは前回指定した型変換規則の通りに変換されることを望んでいる」という前提が崩れぬ限り，Log-based 法において型変換規則の導出の精度が落ちることはない．これ以上の詳細は省略し，実装の詳細が型変換規則の導出の精度に影響する Type-inference-based 法の記述に文面を割くこととする．

5.2 Type-inference-based 法の実装の詳細

Type-inference-based 法はソースコードを静的に解析することで型変換規則を導出することを第 4 章で述べた．この手法では FFI のもつ型推論器が，ライブラリ言語である EusLisp のソースコードに対し型推論を行い，関数型を決定する．絞られた引数型の候補をもとに型変

[*1] Type-inference-based 法について，実際の完全な実装は https://github.com/ikeshou/InferenceEus から見ることができる．主要となる部分については抜粋して付録に記載した．

換規則の導出がなされる．主要な機構は型推論器がソースコードに対し型推論を行う部分であるため，本節では型推論の内容について述べる．

5.2.1 実装した型推論器の性質

この型推論器は正確には，各項に対しできるだけタイトな型の上界を求めるというものである（便宜上型推論という言葉を使用している）．完全に型の制約式を集めることは諦め，ビルトイン関数の型を頼りに，集められる範囲で型に対する正確な制約式を集め，型の候補を Top 型である object から絞り込んでいく．注意するべき事柄として，一般的な型推論器と異なり，型安全性は保障されない．また，処理の過程で明らかな型エラーを検出することは可能であるが，包括的な型検査も行われない．このような設計であるのは以下の理由からである．

- 動的型付け言語のソースコードから完全に型の制約式を集めることは困難である．同時に完全な型検査も困難である
- FFI の用途上，使用歴の長いライブラリに対して推論を行うことが多いため，型エラーはないと仮定してもよいと考えられる
- 型変換規則を導出するという目的を考えると，完全に保守的な推論を行う意味は薄い．集められる範囲で情報を集め型の候補を絞り込んでくれるだけで有用であると考えられる（これは第 7 章で実証される）

この推論器は，以下の幾つかの仮定が満たされる限りにおいて，本来の型よりも真に小さい型を求めることはなく，常に同じか大きな型を求める．以降はこの仮定をもとに考察および実装を行う．

- 推論対象のライブラリに型エラーは含まれない
- 一度変数に値がセットされたら，その型の部分型となる型の値しかセットされ得ない．（実用上，初期値として nil が設定された場合は変数の初期の型は object，初期値として t が設定された場合は変数の初期の型は bool，int/float/rational が初期値として設定された場合は変数の初期値の型は number と定める）
- 5.2.6 小節で導入する，条件分岐における動的ディスパッチの判定は常に成功する

5.2.2 実装した型推論器で表現可能な型

この型推論器で表現可能な型の説明を行う．特徴的なものは部分型（サブタイプ多相），パラメトリックな型（有界量化した let 多相），不特定の型の上界表現である．

部分型

第 2.3.4 小節で扱ったものと基本的に等しい．推論の対象とする EusLisp では単一継承のみが許され，複数インターフェースの継承という概念がないため，単純なサブクラス関係のみを

判定してオブジェクトの部分型関係を決定することとする．なお，数値型を示す number クラスや，int, float, rational クラス，真偽値を示す bool クラスや nil-class, t-class といったクラスを，組み込みのクラスを補う形で導入した．

また，合併型を表現するために，クラスの集合を表現するクラスである group クラスを導入した．これは内部のスロット class-list に，要素となるクラスのリストを持つだけのクラスである．なお，この合併はクラスの直和の形で直ちに表現し直すことが可能である．また，交差型もクラスの直和の形で直ちに表現し直すことが可能である．

グループ G_1，G_2 の合併型及び交差型をクラスの直和で表すアルゴリズムを簡単に述べる．EusLisp では単一継承のみが許されるため，全クラスの継承関係を木の形で表現可能である．初めに定義済みの全クラスのリストをリフレクションにより調べ，クラスとインデックスの対応を決定する（このインデックス番号をノード番号とするようなグラフを作成していくことになる）．そして親クラスから子クラスへ有向辺を張ったときの，クラスツリーの隣接リスト表現を得る．これは全クラスについて親クラスを調べることで簡単に実現できる．グループからその全てのサブクラスを求める関数 reachable-list-by-group-or-class も，グループを構成するクラスについて対応するクラスツリーのノードから探索を行い，巡回したノードの和集合をとることにより実現できる．G_1，G_2 について reachable-list-by-group-or-class を実行し，その和集合ないし積集合をとれば，あとは直和表現に直すだけで合併型および交差型をクラスの直和で表現できた．直和表現への変換は，クラスツリーのルートを起点に以下の 3 つの場合わけをして再帰的に探索を行えばよい．現在地から到達可能な全てのノードが問題とするノードリストに含まれている場合は現在地に対応するクラスを追加し，そのノードでの探索を終了する．全てではないが一部が含まれているなら，そのサブクラスのいずれかが class-list に入るべきである．子ノードを再帰的に探索する．全く含まれていないなら，そのノードでの探索を終了する．

パラメトリックな型

第 2.3.6，2.3.7 小節で扱ったものと等しい．一般列を受け取り，それのコピーを作って返す EusLisp の copy-seq 関数などは $\forall\tau \in Union(cons, array, vector).\ \tau \to \tau$ と型を付ける．

不特定の型の上界表現

不特定の型の上界表現は，型の上界を求めるアルゴリズムそのものと深く関係する．第 5.2.1 小節で述べたように，ソースコードから完全に型の制約式を集めることは困難であるため，集められる範囲で型に対する正確な制約式を集め，型の候補を絞り込んでいく方針をとっている．ここで，「集めることができた制約から絞り込むことができた型の候補」，つまり「上界が何らかの型で定められた不特定の型」を示す表現が必要となる．この型はひとまず "?" をプレフィクスとして持つ型変数で表し，その型変数に対する上界を記録しておくことで取り扱えるようにする．

f という一引数一戻り値の関数に対して推論を行う場合を例にとり，具体的に説明する．推論にあたり，引数型に型変数 α，戻り値型に型変数 β をあてたとする（$f : \alpha \to \beta$）．静的に強

く型付けされた言語で α に対する制約を集め切り，整理した結果 $\alpha <: T$ となった場合，引数型は T であると言える．仮引数に要請される「全ての」条件は，T の部分型であることで満たされているため，実引数としていかなる T の部分型であるような型の値がきても型エラーを起こさない．一方，今回の推論で集められる範囲で集めた制約を整理した結果 $\alpha <: T$ となった場合を考える．引数型は T であると言うことはできない．仮引数に要請される「一部の」条件は，T の部分型であることで満たされるだけである．引数型を $?\alpha$ とし，それに対する求まった上界 T を記録するにとどめる．実引数にどんな型の値がきたとしても，型エラーを起こさないことの判定はできない．ただ分かるのは実引数の型が T の部分型でない場合，明らかな型エラーであるということのみである．

不特定の型の上界表現は，有界量化した全称型とは異なることを強調しておく．どちらも関数型の表現に型変数が残るということは共通するが，意味するところは異なる．比較のため，前者の例として $f_1 :?\alpha \rightarrow ?\beta$（但し $?\alpha$ の上界 S，$?\beta$ の上界 T），$f_2 :?\alpha \rightarrow ?\alpha$（但し $?\alpha$ の上界 S）を，後者の例として $\forall \alpha <: S.\, g : \alpha \rightarrow \alpha$ を取り上げる．f_1，f_2，g のいずれも，実引数の型が S の部分型でない場合は明らかな型エラーとわかる．しかし，実引数の型が γ （$\gamma <: S$）の場合，後者は型エラーが起きず戻り値は γ になると保障される．対し前者は型エラーが起きないかどうかは判定不能である．ソースコードに型エラーを起こすコードは含まれていないと仮定したときに初めて，f_1 の戻り値は β'（但し β' の上界 T），f_2 の戻り値は γ になると分かるのである．（f_1 の戻り値が β ではなく β' なのはパラメトリック多相のためである）

5.2.3 推論の流れ

本小節では全体の推論の流れを説明する．推論の流れは以下である．

0. 予めビルトイン関数全てに対し，手動で関数型の注釈をつけておく
1. ターゲットのライブラリに対し必要な前処理を行う
2. 型環境の作成を行う
3. ターゲットのライブラリに対し推論を行う

ビルトイン関数に対する関数型の注釈の作成

推論の前に，予めビルトイン関数に対する関数型の注釈を作成しておく．これをもとに未知の型に対する制約式が集まり，ユーザー関数の引数や戻り値の型の上界が定まっていくことになる．関数型の注釈を作成する際は，パラメトリックな型や不特定の型の上界表現を用いて正確に型を記述するようにする．特にパラメトリックな関数やアドホックな関数の戻り値型について型を広くつけてしまわないように注意が必要である．前者の例として，一般列を受け取り，それのコピーを作って返す EusLisp の copy-seq 関数を取り上げる．これは copy-seq: $\forall \tau \in Union(cons, array, vector).\, \tau \rightarrow \tau$ と型を付け，copy-seq: $Union(cons, array, vector) \rightarrow Union(cons, array, vector)$ と型をつけてしまわないようにする．後者の例として，数値を受け取り，引数に float が含まれていたら float を，そうで

なく rational が含まれていたら rational を，全て int であったら int を返す EusLisp の + 関数を取り上げる．本来は任意個の引数をとるが，ここでは簡単のため 2 引数関数であるとする．これは $+ : number \rightarrow number \rightarrow ?\alpha$（但し $?\alpha$ の上界 $number$）と型をつけ，$+ : number \rightarrow number \rightarrow number$ と型をつけることがないようにする．

ビルトイン関数の中には推論の実行時に各引数の情報を解析することで，戻り値により詳細な型を付けることが可能であるものが存在する（アドホックな関数についてもそうであるが，対応する量が膨大になるため省略している．集める型の制約式が多少弱くなりうるのみである）．例えば，一般列と一般列クラスを受け取り，第一引数の列を第二引数で与えられたクラスのオブジェクトに変換する EusLisp の関数 coerce などである．これまで同様，予め型を付けるとするなら coerce: $Union(cons, array, vector) \rightarrow metaclass \rightarrow ?\alpha$（但し $?\alpha$ の上界 $Union(cons, array, vector)$）なのだが，第一引数にクラスの名前が与えられている場合は直ちに戻り値のクラスが定まるだろう．このように推論の実行時に各引数の情報を解析することで，戻り値により詳細な型を付けることが可能であるものについては，個別に対応することにしている．

ターゲットライブラリに対する必要な前処理

第 2.3.9 小節で let 多相の単純型システムの型再構築を行う方法として型方程式の収集と単一化を交互に繰り返すアルゴリズムがあることを述べた．単一化で明らかにされなかった型変数は型スキーム化され，多相性が伝播することになる．部分型が導入された場合，型方程式が型不等式となるが基本的に同じである．関数適用部分を見る際，その関数の型は明らかになっている必要がある．

しかし，今回の推論のターゲットとする EusLisp では Common Lisp 同様に関数定義は実際の関数の使用までに終わっていれば問題ないため，上から順に推論を行った場合，関数定義で出現する関数の型が明らかにされていない可能性がある．ワンパスで推論を行うためには，前処理として大域関数定義および大域変数定義の並び替えを行う必要がある．

ここでは並び替えを行うアルゴリズムを簡単に説明する．基本的には関数参照関係，変数参照関係を読み解き，定義部分で参照している関数及び変数の型が明らかになってから，その関数定義ないし変数定義が出現するようにすればよいだけである．まずはターゲットライブラリを S 式の羅列として捉え，S 式とインデックスの対応をとる（このインデックス番号をノード番号とするようなグラフを作成していくことになる）．次に，各 S 式について，内部でユーザー定義大域関数やユーザー定義大域変数を参照している場合，その定義部分を示すノードから自身の S 式を示すノードへ有向辺をはるようにする（なお，局所関数や局所変数と名前の衝突が起きているケースや，関数と変数の間で名前の衝突が起きているケースに注意する．再帰関数については自己ループをはらないようにする）．作成されたグラフに対し，トポロジカルソート [25] を行うことにより有向非巡回グラフ（DAG）へと変換することができる．なお，この前処理を行いワンパスで推論していく手法では，相互再帰をする関数が存在する場合に変換に失敗し，推論ができないという欠点がある．

型環境の作成

スコープチェーンを持つ型環境を作成する．EusLisp は Common Lisp 同様，変数名前空間と関数名前空間が別となっているため，ビルトイン大域変数型環境とビルトイン大域関数型環境を別に用意する必要がある．大域変数型環境の作成には，リフレクションにより全大域変数を集め，それらのクラスを調べ登録するのみでよい．大域関数型環境の作成には，第 5.2.3 小節で作成していた型注釈を読み込めばよい．この際，型変数に対する上界もセットで保持するようにする．

ターゲットライブラリに対する推論

S 式と関数型環境と変数型環境を入力にとり，S 式に対し推論を行い，戻り値の型を返す推論関数 infer をターゲットライブラリに対し再帰的に適用していく．この際，型環境は自身への追記がなされたり，その環境を親環境にもつ型環境が作成されたりする．作成された型変数ついては上界の情報が適宜更新されていくことになる．

5.2.4 関数適用部分における型不等式の立式及び整理と，型の上界を求めることの保障

関数 $func : X \to Y$ に対し引数として $arg : Z$ が渡されていたとする（X，Y はクラスのグループか，有界量化した全称型か，不特定の型の上界表現．Z はクラスのグループか，不特定の型の上界表現）．この関数適用部分では，第 2.3.4 小節で述べたように，$X :> Z$ なる制約式が立つ．X と Z のとりうる 6 つの組み合わせについてどのように制約式を立式可能であるか見ていく．以下，X を G_1 もしくは $\forall\alpha <: G_3$ もしくは $?\beta$（但し $?\beta$ の上界 G_4）とする．Z を G_2 もしくは $?\gamma$（但し $?\gamma$ の上界 G_5）であるとする．なお，型変数はパラメトリック多相が反映され，関数型決定時の型変数から新たな型変数が適切にコピー生成されているものを考える．

$X = G_1$ かつ $Z = G_2$ のとき
この場合，$G_1 :> G_2$ を検査する．検査を通ったら戻り値 Y であり，通らなかったら明らかな型エラーである．

$X = \forall\alpha <: G_3$ かつ $Z = G_2$ のとき
この場合，$G_3 :> G_2$ を検査する．検査を通ったら戻り値 $[\alpha \mapsto G_2]Y$ であり，通らなかったら明らかな型エラーである．

$X = ?\beta$（但し $?\beta$ の上界 G_4）かつ $Z = G_2$ のとき
この場合，まず $G_4 :> G_2$ を検査する．検査を通った場合も安全性はわからない．ソースコードに型エラーが含まれないという仮定を用いて初めて戻り値を $[?\beta \mapsto G_2]Y$ と定めることが

できる．通らなかった場合は明らかな型エラーである．

$X = G_1$ かつ $Z =?\gamma$（但し $?\gamma$ の上界 G_5）のとき
この場合，まず $G_1 :> G_5$ を検査する．検査を通ったら戻り値 Y である．通らなかった場合安全性はわからない．ソースコードに型エラーが含まれないという仮定を用いて初めて戻り値を Y と定めることができる．

$X = \forall\alpha <: G_3$ かつ $Z =?\gamma$（但し $?\gamma$ の上界 G_5）のとき
この場合常に安全性はわからない．ソースコードに型エラーが含まれないという仮定を用いて初めて戻り値を $[\alpha \mapsto ?\gamma]Y$ と定めることができる．また，この仮定のもと $?\gamma$ に対し，$G_3 \wedge G_5 :>?\gamma$ という型不等式を立式可能である．

$X =?\beta$（但し $?\beta$ の上界 G_4）かつ $Z =?\gamma$（但し $?\gamma$ の上界 G_5）のとき
この場合常に安全性はわからない．ソースコードに型エラーが含まれないという仮定を用いて初めて戻り値を $[?\beta \mapsto ?\gamma]Y$ と定めることができる．また，この仮定のもと $?\gamma$ に対し，$G_4 \wedge G_5 :>?\gamma$ という型不等式を立式可能である．

最後の二つの例では不特定の型を表す型変数に対し，上界が更新された．ここでは型の連立不等式から，（自明に成立すると分かる）一方にのみ型変数が現れる型不等式を抽出している．両辺に型変数が現れる不等式のまま保持し，適切なタイミングでまとめて不等式の整理を行った場合と比べて，立式した不等式が多少弱くなる可能性がある．しかし，求めた型の上界がタイトでなくなる可能性があるのみであり，上界を求めるという性質が失われることはない．そして，上界にしか興味がないため，下界については立式していない（下界について立式を行っても結果的に求まる上界に影響しない）．

5.2.5　ユーザー関数定義部分での推論

引数部を見て，引数と戻り値に未知の型変数をあてる．なお，この型推論器において「制約を集め切ることができたか」を判定することはできないため，ユーザー関数の引数型については常に不特定の型の上界表現の形で定まる．戻り値については具体的な型で定まる場合と，不特定の型の上界表現の形で定まる場合がある．どのような場合に戻り値が不特定な型の上界表現となるかについては，続く第 5.2.6 小節で説明を行う．

引数と戻り値について型変数をあてたら，現在の型環境を親に持つ，新たな変数型環境を作成し，それらの型を登録する．関数型については，defun による大域関数定義の場合は，大域関数型環境に登録するのみである．flet や labels による局所関数定義の場合は，適切に関数型環境を作成し，そこへ登録を行う．関数ボディの部分について推論を行うことで，あてた型変数に対する制約及び戻り値の型を得ることができる．得た戻り値の型で戻り値にあてた型変数を置き換えれば推論が終了する．

5.2.6 推論が困難な部分に対する取り組み

これまでに関数定義部分での推論，関数適用部分での推論について述べた．実装を行った型推論器では EusLisp の全ての特殊書式を含め，さまざまな構文要素に対応している．ここでは推論が困難な部分であるメソッドコール部分，条件分岐部分についての取り組みについて述べる．

メソッドコール部分での推論

メソッドコール部分での推論は，メソッドがどのオブジェクトのメソッドであるかを絞り込み，戻り値の候補を決定する必要があるため，難しい．今回は型推論器のモードを二つ用意し，メソッドコール部分についてどの程度の推論を行えるかを切り替えられるようにした．一つは粗い推論を行う RC (Receiver-inference-Coarse)，もう一つは詳細な推論を行う RP (Receiver-inference-Precise) である．

RC では単純にメソッドコール部分の S 式での推論をスキップする．戻り値は any のようなクラスを返し，特別扱いをする．RP では改善を行い，レシーバータイプの推論のみを行う．予め存在する全てのクラスに対し，有するメソッドを調べておくことで，メソッド名からレシーバーのとりうるクラスの集合を決定できるようになる．戻り値は RC 同様に any とする．第 7 章で実証実験を行う際には，この両モードについて行うことにする．

条件分岐部分での推論

条件分岐部分での推論は，条件分岐のテスト部分で型判定を伴うような動的ディスパッチのあるコードにおいて困難となる．条件分岐の実行部分を常に実行されるものとして推論し，制約を立式することはできない．今回は型推論器のモードを二つ用意し，条件分岐部分についてどの程度の推論を行えるかを切り替えられるようにした．一つは粗い推論を行う FC (Flow-inference-Coarse)，もう一つは詳細な推論を行う FP (Flow-inference-Precise) である．

FC では常に評価される，条件分岐のテスト部分についてのみ推論を行う．戻り値は any とする．FP では改善を行い，まずテスト部分において型判定が行われているかを調べる．型判定が行われているかどうかは，型を調べるための組み込み関数群が使用されているかのみで判断することとする．行われていないと判断された場合，全ての実行部に対して推論を行い型の制約を集める．戻り値は不特定の型の上界表現を用い，型変数の上界として全実行部の合併型を設定する．行われていると判断された場合は FC 同様の処理を行う．

なお，上記の型判定を伴う動的ディスパッチの判定は，ユーザーの作成した型判定関数などに対応していない．テスト部分において型判定が行われていることを見抜くことができなかった場合，型推論に失敗し，型の上界を求めるという性質が失われる可能性がある．

第 6 章

Python/EusLisp FFI における提案手法のケーススタディ

これまでに FFI は型変換を行うこと，その型変換規則は自明ではないのでユーザーが与える必要があること，その記述量が多くなってしまい問題であること，提案手法である Type Description Helper は型変換規則を半自動的に導出するためユーザーによる型変換規則の記述量を減らすことができると考えられることを述べた．本章では第 5 章で実装を行った Type Description Helper が実際に型変換規則を導出することをコードとともに見ていく．そしてユーザーの記述する必要のある型変換規則の量が削減されることを確認する．

6.1 Log-based 法のケーススタディ

実際に, Log-based 法は型変換規則を半自動的に導出することで, ユーザーによる型変換規則の記述の量を減らす．ソースコード 3.6 に挙げた EusLisp の sum-of-0-0-element 関数を, Log-based 法と共に使用した場合のコードをソースコード 6.1 に示す．すでに見たように 1,2 行目の EusArray といった型コンストラクタは変換先の型を示す型タグの情報を登録し，他言語関数呼び出し時にはその型タグに示された型のオブジェクトをライブラリ言語である EusLisp の環境でコピー生成する（この例でいうと#a((1 2) (3 4)), #a((5 6) (7 8)) といったオブジェクトを生成する）．1, 2 行目で他言語関数のプロキシーである sum_of_0_0_element は型タグをもったオブジェクトを引数に呼び出されるため，図 4.1 のロガーが EUS_LIB.sum_of_0_0_element 関数のマッピングテーブルに型コンストラクタ EusArray， EusArray を登録する．以降の 5, 6 行目，8，9 行目では型タグを持たない，ホスト言語である Python のオブジェクトを引数に関数呼び出しが行われている．既にこの関数は適用すべき型変換規則がマッピングテーブルに登録されているため，4.1 のマッパーがマッピングテーブルを元に EusArray なる型コンストラクタを二つの引数に対して自動的に適用する．本来ならソースコード 3.7 のようにして毎回 EusArray なる値に対する型変換規則の記述を行う必要があった．しかしこのように一度値に対して型変換情報を付加した状態で関数を呼べば，以降は型変換規則の記述なしに，変換可能な Python オブジェクトを渡すのみで自動的に変換されるようになる．なお，ジェネリックな

ソースコード 6.1. EusLisp の sum-of-0-0-element 関数を，Python から Log-based な型変換規則の導出と共に呼び出す例

```
1 result1 = EUS_LIB.sum_of_0_0_element(EusArray
      ([[1,2],[3,4]]), EusArray([[5,6],[7,8]]))
2 # type mapping is registered automatically
3
4 result2 = EUS_LIB.sum_of_0_0_element([[9,10],[11,12]],
      [[13,14],[15,16]])
5
6 result3 = EUS_LIB.sum_of_0_0_element([[17,18],[19,20]],
      [[21,22],[23,24]])
```

関数を使用するケースなどでは同一関数を様々な引数の型で呼び出すことが考えられるが，そのような場合では再び手動で明示的な型変換規則の記述が必要となる．

6.2 Type-inference-based 法のケーススタディ

実際に，Type-inference-based 法は型変換規則を半自動的に導出することで，ユーザーによる型変換規則の記述の量を減らす．ソースコード 3.6 に挙げた EusLisp の sum-of-0-0-element 関数を，Type-inference-based 法と共に使用した場合のコードをソースコード 6.2 に示す．FFI ライブラリがライブラリ言語である EusLisp のファイルをロードした時点で図 4.1 の型推論器が一連の他言語関数を対象として型推論を行う．sum-of-0-0-element 関数の引数 arr1, arr2 は共に aref 関数の引数となっており，aref 関数は array のみを引数にとる関数であるため，今回引数の型は共に array であると特定される．EUS_LIB.sum_of_0_0_element 関数の possible arg-types 及びマッピングテーブルに型コンストラクタ EusArray，EusArray が自動的に登録される．以降の 5，6 行目では型タグを持たない，ホスト言語である Python オブジェクトを引数に関数呼び出しが行われているので，図 4.2 のマッパーがマッピングテーブルを元に，二つの引数に対して EusArray なる型コンストラクタを自動的に適用する．8，9 行目では EusLisp の（nil terminated）cons を型タグにもつ Python の list を引数に関数呼び出しが行われているが．この引数は possible arg-types 属性を見るに正当でないため，他言語関数に処理を渡す前に，ホスト言語側のみで TypeError であると判断できる．本来ならソースコード 3.7 のようにして毎回 EusArray なる型変換を書くところが，型推論がうまくいくと，型変換規則の記述なしに，変換可能な Python オブジェクトを渡せば最初から自動的に型変換されるようになる．また，明らかに適用不可能な型の引数とともに他言語関数が呼ばれた場合，型エラーをホスト言語側のみで検知できるようになっている．なお型推論で型を絞りきれず，かつデフォルトの型変換規則から外れた型指定を行いたい場合は，手動で明示的な型変換規則の記述が必要となる．

ソースコード 6.2. EusLisp の sum-of-0-0-element 関数を，Python から Type-inference-based な型変換規則の導出と共に呼び出す例

```
1 # the type conversion rule has already been registered
    automatically!
2 result1 = EUS_LIB.sum_of_0_0_element([[1,2],[3,4]]),
    [[5,6],[7,8]]))
3
4 result2 = EUS_LIB.sum_of_0_0_element([[9,10],[11,12]],
    [[13,14],[15,16]])
5
6 # result3 = EUS_LIB.sum_of_0_0_element(EusList
    ([[17,18],[19,20]]), EusList([[21,22],[23,24]]))
7 # => raises TypeError
```

第 7 章

既存のライブラリに対する実証実験

本章では実際に使用されているライブラリを通して Type Description Helper がどの程度型変換規則を導出できるのか定量的な評価を行い，その有用性を確認する．Log-based 法に関しては定量的な評価が難しい（ユーザーのログを用いれば必ず型変換規則の導出に成功する）．そのため本章では基本的に Type-inference-based 法に対して定量的評価を行う．第 7.1 節では実験方法を説明し，続く第 7.2 ではその実験結果を示す．第 7.3 節では実験結果に対する考察を行う．80-85% の精度で引数型の変換先を確定できることを確認し，それでなお現行の Type-inference-based 法では型変換規則を導出しきれない例について分析を行う．そしてそのような例については Log-based 法により確かに導出を行えることを確認する．

7.1 実験方法

実験の対象とした EusLisp ライブラリは，それぞれ作者の異なる以下の 3 つである．EusLisp では Common Lisp 同様にコンパイラの効率的なコード生成のためユーザーが型を記述することができるが，以下のライブラリにはいずれもそのような記述は含まれていない．

- Python/EusLisp FFI（FFI における EusLisp 側のサーバープログラム及びそのテスト）
- RCB4LISP（EusLisp プログラムから，近藤科学製コントロールボード RCB4 のアセンブリ言語を表現する EusLisp データへと変換するコンパイラ）
- RCB4ASM（上記アセンブリ言語のデータから，近藤科学製コントロールボード RCB4 のマシン語命令を表現する EusLisp データへと変換を行うアセンブラ）

これらのライブラリに存在する 125 個の関数について，表 7.1 に示す 4 つの推論条件のもと関数の引数および戻り値の型の上界を推論する．RC は Receiver-inference-Coarse の，RP は Receiver-inference-Precise の，FC は Flow-inference-Coarse の，FP は Flow-inference-Precise の略記であり，単に 4 つの実験条件のラベルを表す（これらの推論条件は第 5.2.6 小節で見た取り組みと一対一対応している）．その後これら 125 個の関数に対して，ソースコードを見て引数型と戻り値型を手動で特定し，推論で得た型の上界と結果を比較し集計を行う．

表 7.1. ライブラリ関数の型推論の推論条件

レシーバータイプの推論＼条件分岐の詳細な推論	なし	あり
なし	RC-FC	RC-FP
あり	RP-FC	RP-FP

表 7.2. ライブラリ関数の型推論結果

推論条件＼集計指標	T-funcs	T-args	T-rets	W-args	W-rets
RC-FC	97 (77.6%)	160 (80.0%)	84 (67.2%)	0 (0%)	0 (0%)
RP-FC	102 (81.6%)	167 (83.5%)	84 (67.2%)	0 (0%)	0 (0%)
RC-FP	100 (80.0%)	168 (84.0%)	98 (78.4%)	0 (0%)	0 (0%)
RP-FP	106 (84.8%)	175 (87.5%)	98 (78.4%)	0 (0%)	0 (0%)

全引数の個数は 200 個であり，全戻り値の個数は 125 個である（EusLisp は Common Lisp とは異なり多値返却を許していないため，全戻り値の個数は関数の個数と一致する）．集計する指標は，引数型について全てタイトな型の上界を特定できた関数の個数（T-funcs），全引数の中でタイトな型の上界を特定できたものの個数（T-args），上界を求められなかったものの個数 （W-args），全戻り値の中でタイトな型の上界を特定できたものの個数（T-rets），上界を求められなかったものの個数（W-rets）である．なお，上界を求められないとは，型を集合として捉えたときに推論結果の型が手動でつけた型の全てを包含しているわけではないということである．

7.2　実験結果

表 7.1 で示した 4 つの推論条件のもとライブラリの関数群を推論し，手動でつけた型との比較を行った結果を表 7.2 に示す．全引数に対しタイトな型の上界を求めることのできた関数の割合は最良で 84.8% となった．個別に引数を見て行った時，タイトな型の上界を求めることのできた割合は最良で 87.5% となった．戻り値については最良で 78.4% となった．なお，上界を求められなかったものは引数戻り値共に確認されなかった．レシーバータイプの推論機能を入れることで引数の推論結果は 3.5% 改善し，条件分岐の詳細な推論機能を入れることで引数の推論結果は 4.0% 改善した．両方を入れた場合には 7.5% の改善が見られた．（これが上記の値の和である必然性はない．単にそれぞれの機能により型の特定が進んだ引数の集合が排反だっただけであり，偶然である．）

7.3　評価と考察

理論通りであれば，FC については常に型の上界を返すため，W-args と W-rets は 0 となるはずである．FP については動的ディスパッチが起きているかどうかの判別に失敗した場合に

限り，W-args 及び W-rets が正の値となる可能性があった．結果を見ると，これらの条件においても W-args 及び W-rets は 0 となっており，今回の簡易的な動的ディスパッチの判定は型を推論するのに十分なものであったということが示唆された．

FFI における型変換規則の導出という用途において，大切な指標は T-funcs 及び T-args である（結果的に戻り値についても型の上界を求めることができるが，利用することはない．他言語関数の戻り値の型は実行時に確定することができるためである）．これらの指標は同じ傾向を示すため T-args について着目する．推論機構としては最も弱い RC-FC においても 80.0% という高い精度で引数の型を推定できている．この理由としては以下の 2 つが考えられる．

- そもそも組み込みクラスに対してはそのクラス専用の関数が多く存在し，そういった関数を適用している部分から型を特定していくことが容易である．ジェネリックな関数が定義されている場合でも速度の観点から特化した関数を使用していることも多い．
- 特にキーワード引数において，初期値が補助的に与えられていることも多く，型を関数定義の引数部分だけで特定することが可能な場合も多い．

メソッドコールの部分と条件分岐の実行部を除外したとしてもこれだけ引数の型を絞ることができているのである．

レシーバータイプの推論，条件分岐の詳細な推論を入れた場合，共に引数の型の推論結果を向上させた．どちらの機能を利用しても型の制約をより多く立式できるようになる．しかしこれまでタイトな型の上界を求めることができなかったような関数の引数に対し，タイトな型の上界を決定することができるようになるのは興味深い．これらの機能を利用し制約を立式するも，依然として求めた上界がタイトでない場合も考えられる．RC-FC ではタイトな型の上界を求められなかったが，これらによりタイトな型の上界を求めることに成功したユーザー関数の例を分析すると以下の二つの傾向がある．

- 関数引数に対してそのままメソッドコールを一度行い，以降はその引数を使わない場合
- 関数全体が条件分岐で覆われており，条件分岐のテスト節のみでは使用されぬ引数が存在する場合

これらに対して，それぞれ以下の理由のもと，タイトな型の上界を求めることに成功したと考えられる．

- ユーザーがメソッドコールを行うとき，メソッド名が全クラスで固有のものであることも多い．特にユーザーが作成したメソッドには．多くの場合において目的に沿った固有の名称をユーザーが付けているため，メソッド名のみから型を特定できる場合が十分にある．
- 関数全体が条件分岐で覆われているような関数についても動的ディスパッチが行われていない場合も多い．そのような場合は通常通り全ての式を型の制約立式に活かせるため，RC-FC では飛ばされて一切制約が立たなかった変数についても型を絞り込んでい

くことができる．

レシーバータイプの推論と条件分岐の詳細な推論を両方組み込むことで，より導出される型変換規則の量を増やすことができた．

次に，RP-FP でも引数についてタイトな型の上界を求めることができなかった場合について分析を行った結果を述べる．これは上記考察と対応するが，レシーバータイプを特定しきれなかった場合と動的ディスパッチがあり制約を立式できなかった場合が主である．前者についてはメソッドの引数の個数や，引数の型の情報を用いた推論機構を使用することで特定できる可能性がある．後者については flow-sensitive typing[26] などの手法をもちいることで制約を立ていくことができる可能性がある．しかしながら現行のシステムでも，これらのいずれの場合についても，型を絞りきれなかった引数に対し，たった一度値について変換規則を記述することで，Log-based 法により型変換規則を導出することが可能である．

最後に，この実験結果には現れていない現行の推論機構の限界を述べる．この推論において, 型を組み合わせた型については推論していない．具体例を挙げると, float-vector の vector のみ引数として受け付ける関数があったとして，この推論機構は引数型を vector としか推論することができない．実験結果には現れないが，このような場合に適切な型変換規則を導出することに失敗する可能性がある．しかしながら，もしそういったものが引数で要求されたとしても，やはりたった一度値について変換規則を記述することで，Log-based 法により型変換規則を導出することが可能である．

第 8 章

関連研究

本研究では動的型付け言語間の FFI における，型変換規則の半自動的な導出器である Type Description Helper を提案した．Type Description Helper によりユーザーが記述する必要のある型変換規則の量をおさえることができること，明らかに適用不可能な型の引数の検出が可能であることを確認した．このような動的型付け言語間の FFI に関する論文はごく僅かであるため，本章では静的型付け言語との FFI も含めて関連研究として述べることにする．

FFI を利用する際のユーザーの記述量削減に関する研究としては，静的型付け言語との FFI におけるグルーコードの自動生成が挙げられる．SWIG（Simplified Wrapper and Interface Generator）[27] は，C/C++ コードと Tcl, Python, Perl といったスクリプト言語のバインディングの自動生成を行う．具体的には，ヘッダーファイルの情報とユーザーが与えるインターフェースファイルのみを元に，各 C/C++ 関数に対してアクセスするためのラッパー関数を自動生成する．FIG[28] は C コードと Moby[29] 言語のバインディングの自動生成を行う．FIG は SWIG と似ているが，こちらは C のラッパー関数を生成するのではなくユーザー定義の型変換規則に基づき Moby コードを生成するため，データレベルでの互換性があるという特徴がある．このように型変換規則の記述は必要ではあるものの，使用する個々の関数生成に関しては自動化がなされている．

FFI を利用する際のユーザーコードの検証に関する研究としては，静的型付け言語との FFI における静的解析・動的解析が存在する．前者の一例として OCaml/C FFI において型の検証を行うシステムである multi-lingual type inference system[30] が挙げられる．これは C の型で表現された OCaml の型と，OCaml の型で表現された C の型を有する型言語を定義することで，両言語間で完全な型情報の追跡を可能にしている．後者の一例として Java/C FFI において JVM の状態に関する制約・型に関する制約・リソースに関する制約が守られているか検証を行うプログラムである Jinn[31] が挙げられる．他言語関数呼び出しを行う Java の関数を，上記制約を適宜検証するラッパー関数に自動で置き換えることで網羅的な動的解析を可能にしている．

Chiba[32] は高水準言語間インターフェースに適したデザインとして，FFI に替わりコードマイグレーションを提案している．標準の処理系を用いた Ruby/Python FFI である PyCall[33] と埋め込みドメイン特化言語を組み合わせることで，Ruby コード中に Python ラ

イクなコードブロックを記述して使用できるようになるとともに，コードブロックのシンタックスチェックも行うことが可能となる．両言語のシンタックスを混同することに由来する他言語インターフェース部分でのエラーはつきものだが，そのようなユーザーコストを削減する試みである．本研究と着眼点は異なるが関連研究として位置付けることができるだろう．

動的型付け言語間の FFI の実装自体に関する研究は，Ramos ら [34][35] の Racket/Python FFI，Barrett ら [36] の PHP/Python FFI などがある．前者は Python のセマンティクス，組み込みデータタイプを Racket 上で全て実装することで，Racket プラットフォーム上での両言語の実行が可能となっている．後者は Python インタプリタの RPython 実装である PyPy[37] と PHP インタプリタの RPython 実装である HippyVM[38] を組み合わせることで，ソースコードエディター Eco 上での両言語の実行が可能となっている．どちらの研究も高速化を図るため独自の処理系，環境を作り FFI の構築を行なっているが，本研究では標準の処理系および環境での FFI の構築を対象としたため，このような構築手法はとらなかった．また，いずれもユーザーの記述コストや検証という観点からの取り組みはなされていない．

提案した Type Description Helper の機能のうちの一つである Log-based 法に関連する研究としては，プログラムヒストリを用いたコードの自動補完 [39] の例が挙げられる．これは従来型のパターンマッチングに基づく補完時の候補の提案に代わり，記録されたユーザーの使用履歴を用いて候補を提案するというものである．本研究での応用先が FFI であるのに対しこの研究の応用先はエディタであり，対象は異なるものの，使用のログをとって推論に活かすという文脈のもと関連研究として挙げることができるだろう．

Type Description Helper のもう一つの機能である Type-inference-based 法の関連研究としては，動的型付け言語に対する型推論や型検査があり，数多くの先行研究が存在する． Ma ら [40] の作成した TICL は Common Lisp のサブセットを対象とした部分的な型推論器である．これはユーザーの施した最小限の型の記述をもとに，導けるだけ項の型を導き，コンパイラによる効率的なコード生成の一助とすることを目的としている．Robert[41] は Lisp のサブセットを対象に静的な型検査器を構築した．関数定義に Guard なる引数型の制約を与えるものを含める必要があるが，その前提のもと，アドホック多相となっている関数についても型推論が可能であることを示した．また，提案した型システムの定式化および健全性の証明も行っている．Pluquet ら [42] は Smalltalk を対象としたヒューリスティックな型推論器を構築した．変数に送られたメッセージのみを見てインターフェース型を抽出し，変数代入の右辺値の型を推論している．この推論は理論に裏付けられたものではないが，ミリ秒という高速な動作速度のもと 75% の精度で変数の型を再構築が可能であることを示した．Python プログラムを対象とした型推論や型検査の研究のうち関連度の高いものは以下である．Vitousek ら [43] は Python3 コードに変換可能な，漸近的型付けシステムをもつ ReticulatedPython というシステムを提案した．また，Python には型アノテーションの機能がありユーザーが関数に対し型の情報を記述することが可能である．Hassan ら [44] は型アノテーション情報を用いた健全な型推論器 TYPPETE を提案している．サブタイプ関係から型の制約を集め，SMT ソルバーを用いて制約を解くという過程が特徴的である．型アノテーション情報を用いて静的型検査を行うサードパーティーライブラリとして PyType[45] や MyPy[46] も広く利用さ

れている（PyType に関しては型アノテーション情報なしでの限定的な検査も可能である）．Rak-amnouykit[47] らは実際のオープンソースプロジェクトのリポジトリを対象として，どの種の型エラーを引き起こしやすいのかを調査し，PyType と MyPy で検出されたエラーの比較を行った．

第 9 章

まとめと今後の展望

本研究では型が豊富な動的型付け言語間 FFI における型変換において，型変換規則の半自動的な導出器である Type Description Helper の提案を行った．対象言語の一例として Python, EusLisp を選定し，FFI および Type Description Helper の実装を行い，ユーザーによる型変換規則の記述量の削減が可能であることを確認した．

今後の展望としては，本稿執筆時点では型推論機構を用いた半自動的な型変換規則の導出はまだ改善の余地があるため，型推論機構をより充実させることで広範な型の推定が可能となるだろう．具体的に条件分岐の部分において Flow-sensitive Typing などの手法を用いれば，型変数についての制約式をより多く立てることが可能になるだろう．メソッドコールの部分において引数の個数や引数の型といった情報も利用したレシーバータイプの推論を行えば，これまた型変数についての制約式をより多く立てることが可能になるだろう．結果的に関数の引数型をより良い制度で推論できるようになり，FFI における型変換規則の導出がさらに自動化されると考えられる．

発表文献と研究活動

(1) Presented a paper at PRO2019-5 organized by IPSJ SPecial Interest Group on Programming in March 2020.

参考文献

[1] E. Gamma, R. Helm, R. Johnson, & J. Vlissides. (1994). Design Patterns. Addison-Wesley

[2] Smith, B. C. (1984, January). Reflection and semantics in Lisp. In Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 23-35). ACM.

[3] Travis Oliphant. (2006). NumPy. https://numpy.org/.

[4] Travis Oliphant, Pearu Peterson & Eric Jones. (2001). SciPy. https://www.scipy.org/.

[5] Adam Paszke, Sam Gross & Soumith Chintala and Gregory Chanan. (2016). PyTorch. https://pytorch.org/.

[6] Google Brain. (2015). TensorFlow. https://www.tensorflow.org/.

[7] Willow Garage and Stanford Artificial Intelligence Laboratory. (2007). https://www.ros.org/.

[8] Matsui, T., & Inaba, M. (1990). Euslisp: An object-based implementation of lisp. Journal of Information Processing, 13(3), 327-338.

[9] Okada, K., Ogura, T., Haneda, A., Kousaka, D., Nakai, H., Inaba, M., & Inoue, H. (2004, April). Integrated system software for HRP2 humanoid. In IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 (Vol. 4, pp. 3207-3212). IEEE.

[10] Okada, K., Ogura, T., Haneda, A., Fujimoto, J., Gravot, F., & Inaba, M. (2005, July). Humanoid motion generation system on hrp2-jsk for daily life environment. In IEEE International Conference Mechatronics and Automation, 2005 (Vol. 4, pp. 1772-1777). IEEE.

[11] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., ... & Wolczko, M. (2013, October). One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (pp. 187-204).

[12] Python/C API. https://docs.python.org/ja/3/c-api/index.html

[13] ErlPort. http://erlport.org/docs/python.html

[14] Vinoski, S. (1997). CORBA: integrating diverse applications within distributed het-

erogeneous environments. IEEE Communications magazine, 35(2), 46-55.

[15] 前原昭二. (1967). 記号論理入門. 日本評論社.

[16] 五十嵐 淳. (2011). プログラミング言語の基礎概念. サイエンス社.

[17] Pierce, B. C. (2013). 型システム入門 プログラミング言語と型の理論. 株式会社 オーム社.

[18] 横内 寛文. (1994). プログラム意味論. 共立出版.

[19] 林 晋. (1995). プログラム検証論. 共立出版.

[20] Cardelli, L. (1984, June). A semantics of multiple inheritance. In International symposium on semantics of data types (pp. 51-67). Springer, Berlin, Heidelberg.

[21] Hall, C. V., Hammond, K., Peyton Jones, S. L., & Wadler, P. L. (1996). Type classes in Haskell. ACM Transactions on Programming Languages and Systems (TOPLAS), 18(2), 109-138.

[22] Robinson, J. A. (1971). Computational logic: The unification computation. Machine intelligence, 6, 63-72.

[23] Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society, 146, 29-60.

[24] Milner, R. (1978). A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3), 348-375.

[25] Kahn, A. B. (1962). Topological sorting of large networks. Communications of the ACM, 5(11), 558-562.

[26] Foster, J. S., Terauchi, T., & Aiken, A. (2002, May). Flow-sensitive type qualifiers. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (pp. 1-12).

[27] Beazley, D. M. (1996, July). SWIG: An easy to use tool for integrating scripting languages with C and C++. In TCL'96,

[28] Reppy, J., & Song, C. (2006, October). Application-specific foreign-interface generation. In Proceedings of the 5th international conference on Generative programming and component engineering (pp. 49-58). ACM.

[29] Fisher, K., & Reppy, J. (1999, May). The design of a class mechanism for Moby. In ACM SIGPLAN Notices (Vol. 34, No. 5, pp. 37-49). ACM.

[30] Furr, M., & Foster, J. S. (2005, June). Checking type safety of foreign function calls. In ACM SIGPLAN Notices (Vol. 40, No. 6, pp. 62-72). ACM.

[31] Lee, B., Wiedermann, B., Hirzel, M., Grimm, R., & McKinley, K. S. (2010). Jinn: synthesizing dynamic bug detectors for foreign language interfaces. ACM Sigplan Notices, 45(6), 36-49.

[32] Chiba, S. (2019, October). Foreign language interfaces by code migration. In Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (pp. 1-13). ACM.

[33] Kenta Murata. (2016). PyCall: Calling Python functions from the Ruby language. https://github.com/mrkn/pycall.rb.

[34] Ramos, P. P., & Leitão, A. M. (2014). Implementing Python for DrRacket. In 3rd Symposium on Languages, Applications and Technologies. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[35] Ramos, P. P., & Leitão, A. M. (2014, August). Reaching Python from Racket. In Proceedings of ILC 2014 on 8th International Lisp Conference (p. 32). ACM.

[36] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, & Laurence Tratt. (2016). Fine-grained Language Composition: A Case Study. In 30th European Conf. on Object-Oriented Programming (ECOOP 2016), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 3:1–3:27.

[37] (2007). PyPy. http://pypy.org/

[38] (2014). HippyVM. http://hippyvm.baroquesoftware.com/

[39] Robbes, R., & Lanza, M. (2010). Improving code completion with program history. Automated Software Engineering, 17(2), 181-212.

[40] Ma, K. L., & Kessler, R. R. (1990). TICL—a type inference system for common Lisp. Software: Practice and Experience, 20(6), 593-623.

[41] Akers, R. L. (1994). Strong static type checking for functional Common Lisp (Doctoral dissertation, University of Texas at Austin).

[42] Pluquet, F., Marot, A., & Wuyts, R. (2009, October). Fast type reconstruction for dynamically typed programming languages. In Proceedings of the 5th symposium on Dynamic languages (pp. 69-78).

[43] Vitousek, M. M., Kent, A. M., Siek, J. G., & Baker, J. (2014, October). Design and evaluation of gradual typing for Python. In Proceedings of the 10th ACM Symposium on Dynamic languages (pp. 45-56).

[44] Hassan, M., Urban, C., Eilers, M., & Müller, P. (2018, July). Maxsmt-based type inference for python 3. In International Conference on Computer Aided Verification (pp. 12-19). Springer, Cham.

[45] PyType. https://github.com/google/pytype

[46] MyPy. https://github.com/python/mypy

[47] Rak-amnouykit, I., McCrevan, D., Milanova, A., Hirzel, M., & Dolby, J. (2020, November). Python 3 types in the wild: a tale of two type systems. In Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (pp. 57-70).

謝辞

本論文の作成にあたり，終始適切な助言を賜り，また丁寧に指導して下さった千葉滋教授，鵜川始陽准教授，穐山空道助教に深く感謝いたします。毎週定期的に実装の進捗や論文執筆の進捗を確認していただき，非常に励みになりました。実装でつまった点があれば進むべき方向性や知見を示していただき，研究が大きく進みました。

東京大学博士後期課程の山崎徹郎さんには Lisp プログラミングの初歩から型システムのさまざまなトピックまで非常に多くのことを教えていただきました。また，発表論文作成の際には論の通った文章構成の仕方からコントリビューションの整理まで幅広くお世話になりました。最後は FFI という同じテーマを扱い，FFI という題材そのものについても考えを深めることができました。そして東京大学博士後期課程の李森曦さんは自分同様動的型付け言語の型推論という共通トピックを研究の一部で扱っており，モチベーションになるとともに得られた興味深い知見なども聞かせていただきました。同窓生の皆さん，先輩方など研究室のメンバーには常に刺激的な議論を頂き，精神的にも支えられました。ありがとうございます。

また, 東京大学情報システム工学研究室の稲葉雅幸教授, 同研究室博士後期課程の Guilherme Affonso さんには，EusLisp でわからないことがあった際に，たびたび助言をいただきました。快くプライベートなライブラリも使用させていただき，研究を進める上で助けになりました。また，ともに処理系のバグを直すなどの貴重な経験もさせていただきました。この場を借りて感謝の意を示させていただきます。

最後になりますが，困難な状況下でも支えていただいた父親と母親と兄と弟と彼女に心から感謝いたします。大変なときも精神的な支えとなり最後まで研究を進めることができました。本当にありがとうございました。

付録 A

型の上界を求めるアルゴリズムの抜粋

```
;;; this is an extracted program from type_inference.l
(require :FIX "fixed_hash.l")
(require "env.l")
(require "type_util.l")
(require "type_class_tree.l")
(require "infer_receiver_helper.l")
(require "infer_flow_helper.l")
(require "type_data.l")

;;; used in experiment
;;; *receiver-mode*: nil => RC / t => RP
;;; *flow-mode*: nil => FC / t => FP
(defconstant *receiver-mode* t)
(defconstant *flow-mode* t)

;;; <: 型不等式. ?a <: T のとき ((a T) ...) という alist
(defparameter *type-ineq-lt* nil)
;;; :> 型不等式. ?a :> T のとき ((a T) ...) という alist
(defparameter *type-ineq-gt* nil)


;;;;;;;;;;;;;;
;;; macros ;;;
;;;;;;;;;;;;;;
```

```
;;; dot pair を alist 先頭へ破壊的に追加する
;;; (但し nil を push した場合変化しない)
;;; (let ((x '(a . b))
;;;       (y '((c . d))))
;;;   (push-cons-al x y))
;;; => ((a . b) (c . d))
(defmacro push-cons-al (from-cons to-alist)
  `(unless (null ,from-cons)
     (setf ,to-alist (cons ,from-cons ,to-alist))))

;;; alist を alist 先頭へ破壊的に追加する
;;; (但し nil を push した場合変化しない)
;;; (let ((x '((a . b) (c . d))
;;;       (y '((e . f))))
;;;   (push-al-al x y))
;;; => ((a . b) (c . d) (e . f))
(defmacro push-al-al (from-alist to-alist)
  `(setf ,to-alist (append ,from-alist ,to-alist)))

;;; 要素を list 末尾へ破壊的に追加する
;;; (let ((p (list 1 2)))
;;;    (push-end 3 p))
;;; => (1 2 3)
(defmacro push-end (i lst)
  `(setf ,lst (append ,lst (list ,i))))



;;;;;;;;;;;;;;;;;;;;;;
;;; user interface ;;;
;;;;;;;;;;;;;;;;;;;;;;

(defun infer-file (fname)
  "
  Args:
      fname (str)
  Returns:
      fix:hash-table
          各ユーザー関数シンボルをキーに、関数型をバリューにもった辞書
```

```
        導入された型変数は上界の型で置換されている。
        型はクラスオブジェクトまたは group のインスタンス
  "
  (load fname)
  ;;; 変数環境も変わりクラスも増えるので強制的に再読み込み
  (load "type_class_tree.l")
  (load "infer_receiver_helper.l")
  (load "type_data.l")
  (let* ((sexp-vec (source-to-sexp-vec fname))
         (user-vars (collect-user-vars sexp-vec))
         (user-funcs (collect-user-funcs sexp-vec))
         (arranged-vec (convert-to-dag sexp-vec user-vars user-funcs))
         (sexp `(progn ,@(coerce arranged-vec cons)))
         (global-varenv (build-varenv)) ; env
         (dot-pair (build-fenv))
         (global-fenv (car dot-pair)))  ; env
    ;; 型不等式を反映
    (setq *type-ineq-lt* (cdr dot-pair))
    ;; S 式を推論
    (infer sexp global-varenv global-fenv)
    ;; ユーザー関数の型を引き、各引数と戻り値について具象型ならそれを、
    ;;; 型変数なら上界を記録した辞書を作成
    (upper-bound-typing user-funcs (env-ht global-fenv))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 以降 infer-file の補助関数 ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun upper-bound-typing (user-funcs ftype-dict)
  "f: ?a -> ?b, ?a <: S, ?b <: T のとき、f: S -> T で型を登録した辞書を返す"
  (let ((ht (copy-object user-funcs)))
    (maphash
      #'(lambda (fn v)  ; fn: user func symbol, v: t
          (let* ((type-pair (fix:gethash fn ftype-dict))
                 (arg-lst (car type-pair))
                 (ret (cdr type-pair))
                 (arg-subst nil)  ; arg-lst を上界変換したもの
```

```
                (ret-subst (sym-subst ret))) ; ret を上界変換したもの
            ; arg-lst は (S a0 (repeat . a1) (rest) ...)
            (dolist (elm arg-lst)
              (if (atom elm)
                (push (sym-subst elm) arg-subst)
                (progn
                  (push
                    (if (equal elm '(rest))
                      elm
                      (cons (car elm) (sym-subst (cdr elm))))
                    arg-subst)
                  )))
            (setq arg-subst (reverse arg-subst))
            (fix:sethash fn ht (cons arg-subst ret-subst))))
      user-funcs)
    ht))


;;; upper-bound-typing の補助関数
(defun sym-subst (target)
  "
  引数が型変数のシンボルなら *type-ineq-lt* を検索し、上界で変換して返す
  target は int や ?a0 などの型を表す単項
  (optional . T) などの形ではこないとする
  "
  (assert (atom target) "sym-subst(): arg should be an atom")
  (cond
    ; 既にクラスに類する表現になっている
    ((group-or-classp target)
      (if (equal target any-class)
        object
        target))
    ; 型変数
    ((typevarp target)
      (if (assoc target *type-ineq-lt*)
        (cadr (assoc target *type-ineq-lt*))
        object))
    (t
```

```
      (error "unknown type symbol?"))))


;;; main
(defun infer (sexp varenv fenv)
  "
  対象の S 式の型を返す
  型変数に対する型不等式は *type-ineq-lt*, *type-ineq-gt* に記録される
  関数、変数定義がなされた場合は varenv, fenv に適宜登録される
  "
  ;; (format t "infer -> ~A~%" sexp)
  (assert (and (env-p varenv) (env-p fenv)) "infer(): not an env")
  (if (atom sexp)
    (infer-unary sexp varenv fenv)
    (let ((head (car sexp)))
      (cond
        ; EusLisp Core
        ((getenv head fenv)
          (infer-call sexp varenv fenv))
        ((member head '(prog1 progn))
          (infer-prog sexp varenv fenv))
        ((eq head 'quote)
          (infer-quote sexp varenv fenv))
        ((member head '(setq setf defvar defparameter defconstant))
          (infer-set sexp varenv fenv))
        ((member head '(let let*))
          (infer-let sexp varenv fenv))
        ((eq head 'defun)
          (infer-defun sexp varenv fenv))
        ((member head '(flet labels))
          (infer-flet sexp varenv fenv))
        ((eq head 'function)
          (infer-function sexp varenv fenv))
        ; for and while loop
        ((member head '(while until))
          (infer-while sexp varenv fenv))
        ((eq head 'loop)
          (infer-loop sexp varenv fenv))
```

```
((member head '(dotimes dolist))
  (infer-do sexp varenv fenv))
; block and exit
((member head '(block catch))
  (infer-block sexp varenv fenv))
((member head '(return-from throw))
  (infer-return-from sexp varenv fenv))
((eq head 'return)
  (infer-return sexp varenv fenv))
; other common macros
((member head '(push pushnew))
  (infer-push sexp varenv fenv))
((eq head 'pop)
  (infer-pop sexp varenv fenv))
((member head '(incf decf))
  (infer-incf sexp varenv fenv))
((eq head 'with-output-to-string)
  (infer-output-string sexp varenv fenv))
; 個々に対応する必要のある関数たち
; with or without using #'
((member head '(funcall apply))
  (infer-funcall sexp varenv fenv))
; instances
((member head '(instance make-instance))
  (infer-make-instance sexp varenv fenv))
((eq head 'instantiate)
  (infer-instantiate sexp varenv fenv))
; type specification
((eq head 'coerce)
  (infer-coerce sexp varenv fenv))
((eq head 'concatenate)
  (infer-concatenate sexp varenv fenv))
((eq head 'map)
  (infer-map sexp varenv fenv))
; conditional branch
((member head '(and or if when unless cond case))
  (infer-flow sexp varenv fenv))
; receiver
```

```
        ((member head '(send send*))
          (infer-receiver sexp varenv fenv))
        (t
          any-class)))))


;;; 単項
(defun infer-unary (x varenv fenv &key (is-func nil))
  (assert (atom x) "infer-unary(): sexp should be an atom")
  (cond
    ((null x) nil-class)
    ((eq x 't) t-class)
    ; 関数
    ((and is-func (symbolp x))
      (if (getenv x fenv) (getenv x fenv) any-class))
    ; 変数
    ((symbolp x)
      (if (getenv x varenv) (getenv x varenv) any-class))
    ; リテラルのうちオブジェクトでないもの
    ((integerp x) int)
    ((floatp x) float)
    ((numberp x) rational)
    ; リテラルの残り
    (t (class x))))


;;; formulate-actual の補助関数
(defun unify-ineq (ineq-pair ineq-alist &key (lt t))
  "
  1 本の型不等式 ineq-pair と型不等式の集合 ineq-alist を受け取り、
  型不等式をまとめて返す（非破壊）
  型不等式の要素は常に car に型変数が、cadr に具象型が入っており、
  戻り値もその条件を満足する
  型不等式の要素に同一型変数のエントリが複数登録されていることはない
  ineq-pair: 新しく立った型不等式. (?a0 S)
  ineq-alist: これまでに立った型不等式たちの alist. ((?a0 T) (?a1 U) ...)
  lt: type-ineq が <: かどうか
  "
```

```
  (let ((typevar (car ineq-pair))
        (from-cls (cadr ineq-pair))
        (unified-ineq (copy-tree ineq-alist)))
    (if (assoc typevar ineq-alist)
      ; 既に存在する型変数に対する制約
      (let ((to-cls (cadr (assoc typevar ineq-alist))))
        (if lt
          (setf (cadr (assoc typevar unified-ineq))
                (class-intersec from-cls to-cls))
          (setf (cadr (assoc typevar unified-ineq))
                (class-lca from-cls to-cls))))
      ; 新しい型変数に対する制約
      (push-cons-al ineq-pair unified-ineq))
    unified-ineq))


;;; infer-call の補助関数
(defun formulate-actual (formal-type actual-type)
  "
  formal-type :> actual-type（型変数ないし具象型）という式がたったとき、
  actual-type に関する不等式を *type-ineq-lt* に追記する
  (formal-type の上界で actual-type を上から抑えられる)
  "
  (when (and (typevarp actual-type)
             (or (group-or-classp formal-type)
                 (assoc formal-type *type-ineq-lt*)))
    (let* ((upper
             (if (group-or-classp formal-type)
               formal-type
               (cadr (assoc formal-type *type-ineq-lt*))))
           (ineq-pair
            (list actual-type upper)))
      (setq *type-ineq-lt* (unify-ineq ineq-pair *type-ineq-lt* :lt t)))))


(defun formulate-formal (formal-type actual-type)
  "
  formal-type :> actual-type（型変数ないし具象型）という式がたったとき、
```

```
  formal-type に関する不等式を *type-ineq-lt* に追記する
  (actual-type の下界で formal-type を下から抑えられる)
  "
  (when (and (typevarp formal-type)
             (or (group-or-classp actual-type)
                 (assoc actual-type *type-ineq-gt*)))
    (let* ((lower
             (if (group-or-classp actual-type)
               actual-type
               (cadr (assoc actual-type *type-ineq-gt*))))
           (ineq-pair
             (list formal-type lower)))
      (setq *type-ineq-gt* (unify-ineq ineq-pair *type-ineq-gt* :lt nil)))))


;;; infer-call の補助関数
(defun keysym-eq (x y)
  "(keysym-eq :sym 'sym) => t"
  (and (symbolp x) (symbolp y) (equal (string x) (string y))))


;;; 関数呼び出し
(defun infer-call (sexp varenv fenv)
  (let* ((fn (car sexp))
         (body (cdr sexp))
         (ftype (getenv fn fenv))
         (formal-params-list (car ftype))  ; 登録された関数仮引数の型のリスト
         (formal-ret (cdr ftype))          ; 登録された関数戻り値の型
         (ret-memo nil))  ; 呼び出し結果の型
    (while body
      (let* ((actual-sexp (pop body))
             (formal-top (car formal-params-list))
             ; 注目している実引数の S 式がキーワード引数のためのものか
             (is-key-arg
                (and (keywordp actual-sexp)
                     (consp formal-top)
                     ; optional が残っていた場合はキーワード引数として扱わぬ
                     (not (eq (car formal-top) 'optional))
```

```
                (position-if
                   #'(lambda (x)
                     (and (consp x)
                          (keysym-eq (car x) actual-sexp)))
                   formal-params-list)))
        ; 注目している実引数が rest 分に対応するものか
        (is-rest
           (and (not is-key-arg)
                (consp formal-top)
                (eq (car formal-top) 'rest))))
    ;; rest に吸われる場合は不等式がたたない
    ;; それ以外の場合、引数結合させるときの仮引数の型、実引数の型を求める。
    (unless is-rest
        (let ((formal-type
                (cond
                    ; (keysym . a)
                    (is-key-arg
                      (cdr (elt formal-params-list
                                (position-if
                                  #'(lambda (x)
                                    (and (consp x)
                                         (keysym-eq (car x) actual-sexp)))
                                  formal-params-list))))
                    ; (repeat . a)
                    ((and (consp formal-top) (eq (car formal-top) 'repeat))
                      (cdr formal-top))
                    ; (optional . a)
                    ((and (consp formal-top) (eq (car formal-top) 'optional))
                      (cdr (pop formal-params-list)))
                    ; class or group or typevar
                    (t
                      (pop formal-params-list))))
              (actual-type
                (if is-key-arg
                  (infer (pop body) varenv fenv)
                  (infer actual-sexp varenv fenv))))
          ; 制約立式
          (formulate-actual formal-type actual-type)
```

```
                ; 注目している仮引数の型変数と戻り値の型変数が等しい場合、
                ; 実引数の型を戻り値の型としてセット（全称型）
                (when (and (typevarp formal-type)
                           (typevarp formal-ret)
                           (eq formal-type formal-ret))
                  (setq ret-memo actual-type))))))
    ; 関数が全称型でなく、formal-ret が型変数だった場合、
    ; 同様な制約を持つ型変数を生成して返す
    (cond ((not (null ret-memo)) ret-memo)
          ((not (typevarp formal-ret)) formal-ret)
          (t
            (let ((upper (cadr (assoc formal-ret *type-ineq-lt*)))
                  (lower (cadr (assoc formal-ret *type-ineq-gt*)))
                  (fresh-variable (sym)))
              (when (not (null upper))
                (setq *type-ineq-lt*
                  (unify-ineq (list fresh-variable upper)
                              *type-ineq-lt*
                              :lt t)))
              (when (not (null lower))
                (setq *type-ineq-gt*
                  (unify-ineq (list fresh-variable lower)
                              *type-ineq-gt*
                              :lt nil)))
              fresh-variable)))))


;;; prog1, progn
(defun infer-prog (sexp varenv fenv)
  (let ((ret nil)
        (i 0)
        (head (car sexp))
        (body (cdr sexp)))
    (dolist (elm body)
      (let ((inner-ret (infer elm varenv fenv)))
        (cond
          ((and (eq head 'prog1) (= i 0))
            (setq ret inner-ret))
```

```
          ((and (eq head 'progn) (= i (1- (length body))))
            (setq ret inner-ret))))
      (incf i))
    ret))


;;; quote
(defun infer-quote (sexp varenv fenv)
  (let ((tail (cadr sexp)))
    (if (consp tail)
      ; '(1 2) など。(quote (1 2 3)) => cons
      cons
      ; 'x => symbol, 'nil => nil-class, ...
      (cond
        ((null tail) nil-class)
        ((eq tail 't) t-class)
        ((integerp tail) int)
        ((floatp tail) float)
        ((numberp tail) rational)
        ((symbolp tail) symbol)
        (t (class tail))))))


;;; setq, setf, defvar, defparameter, defconstant
(defun infer-set (sexp varenv fenv)
  (let* ((head (car sexp))
         (mid (cadr sexp))
         (tail (caddr sexp))
         (ret (infer tail varenv fenv)))
    (if (consp mid)
      (infer mid varenv fenv)
      (let ((prev-type (getenv mid varenv)))
        (cond ((group-or-classp prev-type)
                 nil)
              ((and (typevarp prev-type)
                    (typevarp ret)
                    (null (assoc ret *type-ineq-lt*)))
                 nil)
```

```
            ((and (typevarp prev-type)
                  (typevarp ret))
               (setq *type-ineq-lt*
                 (unify-ineq
                   (list prev-type (cadr (assoc ret *type-ineq-lt*)))
                   *type-ineq-lt*
                   :lt t)))
            ((typevarp prev-type)
               (setq *type-ineq-lt*
                 (unify-ineq (list prev-type ret)
                             *type-ineq-lt*
                             :lt t)))
            ((equal ret nil-class) (setenv mid varenv (sym)))
            (t (setenv mid varenv ret)))))
    ret))
```

```
;;; infer-let, argparse でヘビーに使用する補助関数
(defun init-typing (sexp varenv fenv &key var target-env)
  "
  sexp は「新しく導入された」変数 var の初期値となる S 式であるとする
  (setq などではすでに型情報があることがあるため、使用できない)
  環境下で sexp の型を評価し、nil-class となった場合、var に型変数を
  割りあて target-env にその深さで登録する。
  それ以外の型ないし型変数で定まった場合、その型を target-env にその深さで
  登録する。登録した初期値の型を返す。
  (例外)
  int, float, rational で初期化していたら number なら OK とする
  t で初期化しているたら bool なら OK とする
  "
  (let* ((init-type
            (infer sexp varenv fenv))
         (register-type
            (cond
              ((equal init-type nil-class) (sym))
              ((equal init-type t-class) bool)
              ((member init-type (list int float rational)) number)
```

```
              (t init-type))
            ))
    (setenv-here var target-env register-type)
    register-type))


;;; let, let*
(defun infer-let (sexp varenv fenv)
  (let ((head (car sexp))
        (bind (cadr sexp))
        (body (cddr sexp))
        (new-varenv (make-env :outer varenv)))
    ;; 各ローカル変数と型のバインドを型環境へ追加
    (dolist (elm bind)
      (if (consp elm)
        ; 初期値あり
        (init-typing
          (cadr elm)
          (if (eq head 'let) varenv new-varenv)
          fenv
          :var (car elm) :target-env new-varenv)
        ; 初期値なし
        (setenv-here elm new-varenv (sym))))
    (infer `(progn ,@body) new-varenv fenv)))


;;; infer-defun, infer-flet の補助関数
(defun argparse (arg-lst varenv fenv parent-varenv)
  "
  1. 関数定義の引数部分を解析し、関数型を構築する
  (a:a0 b:a1 &optional c:a2 d:a3 &rest ls:cons &key e:a4 f:a5 &aux g:a6 h:a7)
  --> (a0 a1 (optional . a2) (optional . a3) (rest) (e . a3) (f . a4))
  (&aux は関数内 let の略記であり、関数型には現れない)

  2. 初期値がない or あっても nil ならその引数には型変数をあてる。
  初期値が定まっているならその型をあてる。
  上記のようにして仮引数と型が登録されたローカル変数環境を、
  parent-env を親にして構築する
```

3．上記関数の引数型を表すリスト、ローカル変数環境を dot pair にして返す
（関数の戻り値については担当しない。あくまでも引数のみ）
"

```
(let ((arg-sym-lst nil)
      (child-env (make-env :outer parent-varenv)))
  (if (null arg-lst)
    ; 無引数関数の場合
    (push nil-class arg-sym-lst)
    ; 引数があるなら解析、同時にバインドも型環境へ追加。
    ; (a b &optional c d &rest ls &key e f &aux g h) の順。後ろから処理
    ; [prev:] を処理済みであるとして経過を記録していく
    (let ((prev (length arg-lst)))
      ; &aux g (h 1) => 環境にのみ追加
      (when (find '&aux arg-lst)
        (let* ((&aux-pos (position '&aux arg-lst))
               (auxarg-lst (subseq arg-lst (1+ &aux-pos) prev)))
          (dolist (elm auxarg-lst)
            (let* ((var (if (consp elm) (car elm) elm))
                   (init-sexp (if (consp elm) (cadr elm) nil))
                   (init-type
                     (init-typing
                       init-sexp varenv fenv
                       :var var :target-env child-env)))))
          (setq prev &aux-pos))
        )
      ; &key e (f 1) => (e . ?a0) (f . int) が追加
      (when (find '&key arg-lst)
        (let* ((&key-pos (position '&key arg-lst))
               (keyarg-lst (subseq arg-lst (1+ &key-pos) prev))
               (keyarg-sym-lst nil))  ; buffer
          (dolist (elm keyarg-lst)
            (let* ((var (if (consp elm) (car elm) elm))
                   (init-sexp (if (consp elm) (cadr elm) nil))
                   (init-type
                     (init-typing
                       init-sexp varenv fenv
                       :var var :target-env child-env)))
```

```
          (push-cons-al (cons var init-type) keyarg-sym-lst)))
      (push-al-al (reverse keyarg-sym-lst) arg-sym-lst)
      (setq prev &key-pos))
    )
  ; &rest ls => (rest) が追加される
  (when (find '&rest arg-lst)
    (let* ((&rest-pos (position '&rest arg-lst))
           (restvar (elt arg-lst (1+ &rest-pos))))  ; &rest ls の ls
      (setenv-here restvar child-env cons)
      (push-cons-al '(rest) arg-sym-lst)
      (setq prev &rest-pos))
    )
  ; &optional c (d 1) => (optional . ?a1) (optional . int) が追加
  (when (find '&optional arg-lst)
    (let* ((&op-pos (position '&optional arg-lst))
           (oparg-lst (subseq arg-lst (1+ &op-pos) prev))
           (oparg-sym-lst nil)) ; buffer
      (dolist (elm oparg-lst)
        (let* ((var (if (consp elm) (car elm) elm))
               (init-sexp (if (consp elm) (cadr elm) nil))
               (init-type
                 (init-typing
                   init-sexp varenv fenv
                   :var var :target-env child-env)))
          (push-cons-al (cons 'optional init-type) oparg-sym-lst)))
      (push-al-al (reverse oparg-sym-lst) arg-sym-lst)
      (setq prev &op-pos))
    )
  ; 残りの通常引数
  (let ((normalarg-lst (subseq arg-lst 0 prev))  ; (a b) など
        (normalarg-sym-lst nil))  ; buffer
    (dolist (elm normalarg-lst)
      (let ((init-type (sym)))
        (setenv-here elm child-env init-type)
        (push init-type normalarg-sym-lst)))
    ; (b a) になっているので反転
    (push-cons-cons (reverse normalarg-sym-lst) arg-sym-lst)
    )
```

```
        ))
    ; return
    (cons arg-sym-lst child-env)))


;;; defun
(defun infer-defun (sexp varenv fenv)
  (let* ((fn (cadr sexp))
         (arg-lst (caddr sexp))
         (body (cdddr sexp))
         (dot-pair (argparse arg-lst varenv fenv varenv))
         (arg-type-lst (car dot-pair))
         (new-varenv (cdr dot-pair))
         (temporal-ret-type (sym))
         (fn-type (cons arg-type-lst temporal-ret-type)))
    (setenv fn fenv fn-type)
    (let ((ret-type (infer `(progn ,@body) new-varenv fenv)))
      ; 実際の型で戻り値型を置換
      (setf (cdr (getenv fn fenv)) ret-type)
      symbol)))



;;; flet, labels
;; (flet ((f (arg) sexp sexp ...)
;;        (g (arg) sexp sexp ...))
;;   sexp sexp ...)
(defun infer-flet (sexp varenv fenv)
  (let* ((head (car sexp))
         (bind (cadr sexp))
         (body (cddr sexp))
         (new-fenv (make-env :outer fenv)))
    (dolist (elm bind)
      (let* ((fn (car elm))
             (arg-lst (cadr elm))
             (fn-body (cddr elm))
             (dot-pair (argparse arg-lst varenv fenv varenv))
             (arg-type-lst (car dot-pair))
```

```
              (new-varenv (cdr dot-pair))
              (temporal-ret-type (sym))
              (fn-type (cons arg-type-lst temporal-ret-type)))
         (setenv-here fn new-fenv fn-type)
         ; labels のときのみ再帰が可能
         (let ((ret-type
                 (infer `(progn ,@fn-body)
                        new-varenv
                        (if (eq head 'flet) fenv new-fenv))))
           ; 実際の形で戻り値型を置換
           (setf (cdr (getenv fn new-fenv)) ret-type))
         ))
    (infer `(progn ,@body) varenv new-fenv)))


;; function, #'
;; #'f or #'(lambda (arg) sexp)
;; こいつが compiled-code を返すのが重要
;; 関数型情報を活かすことでより絞れる apply などの高階関数は個々に対応する
(defun infer-function (sexp varenv fenv)
  compiled-code)


;; while, unless
(defun infer-while (sexp varenv fenv)
  (let ((test (cadr sexp))
        (body (cddr sexp)))
    (infer test varenv fenv)
    (infer `(progn ,@body) varenv fenv)))


;; loop
(defun infer-loop (sexp varenv fenv)
  (let ((body (cdr sexp)))
    (infer `(progn ,@body) varenv fenv)))


;; dotimes, dolist
```

```
(defun infer-do (sexp varenv fenv)
  (let* ((head (car sexp))
         (args (cadr sexp))
         (left (car args))  ; 新たに導入される変数
         (right (cadr args))  ; こちらは評価される
         (body (cddr sexp))
         (new-varenv (make-env :outer varenv)))
    ; 辞書の構築と制約の立式
    (if (eq head 'dotimes)
      (progn
        (setenv-here left new-varenv int)
        (formulate-actual int (infer right varenv fenv)))
      (progn
        (setenv-here left new-varenv (sym)) ; 型は未知
        (formulate-actual cons (infer right varenv fenv))))
    (infer `(progn ,@body) new-varenv fenv)))


;; block, catch
;; (block tag form*)
(defun infer-block (sexp varenv fenv)
  (let ((head (car sexp))
        (tag (cadr sexp))
        (body (cddr sexp)))
    (when (eq head 'catch) (infer tag varenv fenv)) ; catch は tag を評価
    (infer `(progn ,@body) varenv fenv)))


;; return-from, throw
;; (return-from tag val)
(defun infer-return-from (sexp varenv fenv)
  (let ((head (car sexp))
        (tag (cadr sexp))
        (val (caddr sexp)))
    (when (eq head 'throw) (infer tag varenv fenv))  ; throw は tag を評価
    (infer val varenv fenv)))
```

```
;; return
(defun infer-return (sexp varenv fenv)
  (infer-return-from `(return-from nil ,(cadr sexp)) varenv fenv))


;; push, pushnew
;; (push item place) or (pushenw item place :test :test-not :key)
(defun infer-push (sexp varenv fenv)
  (let ((item (cadr sexp))
        (place (caddr sexp)))
    (infer item varenv fenv)
    (if (symbolp place)
      (formulate-actual cons (getenv place varenv))
      (infer place varenv fenv))
    ; 戻り値型は cons
    cons))


;; pop
(defun infer-pop (sexp varenv fenv)
  (let ((tail (cadr sexp)))
    (if (symbolp tail)
      (formulate-actual cons (getenv tail varenv))
      (infer tail varenv fenv))
    ; 戻り値型は不明
    any-class))


;; incf, decf
(defun infer-incf (sexp varenv fenv)
  (let ((tail (cadr sexp)))
    (if (symbolp tail)
      (formulate-actual int (getenv tail varenv))
      (infer tail varenv fenv))
    ; 戻り値型は int
    int))
```

```
;; with-output-to-string
;; (with-output-to-string (str) sexp sexp...)
(defun infer-output-string (sexp varenv fenv)
  (let* ((var (caadr sexp))
         (body (cddr sexp))
         (new-varenv (make-env :outer varenv)))
    ; 引数の型を stream として束縛
    (setenv-here var new-varenv stream)
    (infer `(progn ,@body) new-varenv fenv)
    ; 戻り値型は string
    string))


;; funcall, apply
;; (set-ftype funcall any-class compiled-code (rest)) としてもいいのが、
;; 頑張るともう少し情報が得られる
(defun infer-funcall (sexp varenv fenv)
  (let ((fn-section (cadr sexp))
        (res (cddr sexp)))
    (dolist (elm res) (infer elm varenv fenv))
    ; 立式できる場合は立式
    (when (symbolp fn-section)
      (formulate-actual compiled-code (getenv fn-section varenv)))
    ; 戻り値型
    (if (or (symbolp fn-section) (consp (cadr fn-section)))
      ; lambda or ローカル変数
      any-class
      (cdr (getenv (cadr fn-section) fenv)))))


;; instance, make-instance
;; (instance class &rest)
(defun infer-make-instance (sexp varenv fenv)
  (let ((mid (cadr sexp))
        (res (cddr sexp)))
    (dolist (elm res) (infer elm varenv fenv))
    (cond
      ; mid はクラスオブジェクトを表すシンボルのとき
```

```
      ((and (symbolp mid) (classp (eval mid)))
        (eval mid))
      ((or (symbolp mid) (consp mid))
        (formulate-actual metaclass (infer mid varenv fenv))
        any-class)  ; 判定不能
      (t (error "infer-make-instance(): unknown class parameter")))))


;; instantiate
;; (instantiate class &optional size)
(defun infer-instantiate (sexp varenv fenv)
  (when (= (length sexp) 3)
    (let ((tail (infer (caddr sexp) varenv fenv)))
      (when (typevarp tail) (formulate-actual int tail))))
  (let ((mid (cadr sexp)))
    (cond
      ((and (symbolp mid) (classp (eval mid)))
        (eval mid))
      ((or (symbolp mid) (consp mid))
        (formulate-actual metaclass (infer mid varenv fenv))
        any-class)  ; 判定不能
      (t (error "infer-instantiate(): unknown class parameter")))))


;; coerce
;; (coerce seq result-type)
(defun infer-coerce (sexp varenv fenv)
  (let ((mid (cadr sexp))
        (tail (caddr sexp)))
    (formulate-actual seq (infer mid varenv fenv))
    (cond
      ((and (symbolp tail) (classp (eval tail)))
        (eval tail))
      ((or (symbolp tail) (consp tail))
        (formulate-actual metaclass (infer tail varenv fenv))
        any-class)  ; 判定不能
      (t (error "infer-coerce(): unknown class parameter")))))
```

```
;; concatenate
;; (concatenate result-type seq*)
(defun infer-concatenate (sexp varenv fenv)
  (let ((mid (cadr sexp))
        (res (cddr sexp)))
    (dolist (elm res)
      (formulate-actual seq (infer elm varenv fenv)))
    (cond
      ((and (symbolp mid) (classp (eval mid)))
        (eval mid))
      ((consp mid)
        (formulate-actual metaclass (infer mid varenv fenv))
        any-class)  ; 判定不能
      (t (error "infer-concatenate(): unknown class parameter")))))


;; map
;; (map result-type function seq*)
(defun infer-map (sexp varenv fenv)
  (let ((two (cadr sexp))
        (three (caddr sexp))
        (res (cdddr sexp)))
    (formulate-actual compiled-code three)
    (dolist (elm res)
      (formulate-actual seq elm))
    (cond
      ((and (symbolp two) (classp (eval two)))
        (eval two))
      ((consp two)
        (formulate-actual metaclass (infer two varenv fenv))
        any-class)  ; 判定不能
      (t (error "infer-map(): unknown class parameter")))))


;; and, or, if, when, unless, cond, case
(defun infer-flow (sexp varenv fenv)
```

```
  "
  条件分岐の部分の推論を行う。実験のため *flow-mode* で挙動の制御を行う。
  nil のとき
  確実に評価される最初の test 部分のみ見て、戻り値の型は any とする
  t のとき
  最初の test は必ず見る。test 部分全てにクラス判定ディスパッチがないと
  予測できる場合、全ての test と condition body を見る。
  戻り値の型は型変数をあて、上界を全ての condition body の Union とする
  "
  (if *flow-mode*
    (infer-flow-precise sexp varenv fenv)
    (infer-flow-coarse sexp varenv fenv)))


(defun infer-flow-coarse (sexp varenv fenv)
  (let* ((head (car sexp))
         (test (if (eq head 'cond) (caadr sexp) (cadr sexp))))
    (infer test varenv fenv)
    any-class))


;;; infer_flow_helper.l の補助関数に頼る
(defun infer-flow-precise (sexp varenv fenv)
  (let ((head (car sexp)))
    (cond
      ((member head '(and or))
         (infer-and sexp varenv fenv))
      ((and (eq head 'if) (= (length sexp) 4))
         (infer-if-full sexp varenv fenv))
      ((member head '(if when unless))
         (infer-when sexp varenv fenv))
      ((eq head 'cond)
         (infer-cond sexp varenv fenv))
      ((eq head 'case)
         (infer-case sexp varenv fenv))
      (t (error "infer-flow-precise(): not a conditional sexp")))))
```

```
(defun calc-return-union (ret-type-lst)
  "
  condition body の戻り値を集めてその Union を返す際の補助関数
  ret-type-lst は各条件分岐での戻り値の型を集めたものである
  それらが全て具象型の場合その Union をとった結果を返す
  型変数が一つでも含まれていたら、それぞれの上界を集め、それらの Union を上界とする
  型変数を生成して返す
  "
  (if (some #'typevarp ret-type-lst)
    ; 型変数があった
    (let ((upper
            ; 型変数を全て上界で置換したリストを作成して reduce
            (reduce #'class-union
                    (mapcar #'(lambda (x) (if (typevarp x)
                                              (if (assoc x *type-ineq-lt*)
                                                (cadr (assoc x *type-ineq-lt*))
                                                object)
                                              x))
                            ret-type-lst)))
          (typevar (sym)))
      (push-cons-al (list typevar upper) *type-ineq-lt*)
      typevar)
    ; 全て具象型
    (reduce #'class-union ret-type-lst)))


;; and, or
(defun infer-and (sexp varenv fenv)
  (let ((body (cdr sexp)))
    (if (dynamic-dispatch-p body) ; and, or は全てが test である
      (progn
        (infer (car body) varenv fenv)
        any-class)
      (let ((ret-type-lst nil))
        (dolist (elm body)
          (push (infer elm varenv fenv) ret-type-lst))
        (calc-return-union ret-type-lst)))))
```

```
;; if (else までありり)
(defun infer-if-full (sexp varenv fenv)
  (let ((test (cadr sexp))
        (then (caddr sexp))
        (else (cadddr sexp)))
    (infer test varenv fenv)
    (if (dynamic-dispatch-p test)
      any-class
      (calc-return-union
        (list (infer then varenv fenv) (infer else varenv fenv))))))


;; if (then のみ), when, unless
(defun infer-when (sexp varenv fenv)
  (let ((test (cadr sexp))
        (body (cddr sexp)))
    (infer test varenv fenv)
    (if (dynamic-dispatch-p test)
      any-class
      (infer `(progn ,@body) varenv fenv))))


;; cond
;; (cond (test sexp) (...) (...))
(defun infer-cond (sexp varenv fenv)
  (let ((test-body-pairs (cdr sexp)))
    (if (dynamic-dispatch-p (mapcar #'car test-body-pairs))
      (progn
        (infer (caar test-body-pairs) varenv fenv)
        any-class)
      (let ((ret-type-lst nil))
        (dolist (pair test-body-pairs)
          (infer (car pair) varenv fenv)
          (push (infer (cadr pair) varenv fenv) ret-type-lst))
        (calc-return-union ret-type-lst)))))
```

```
;; case
;; (case val ((not eval-ed list) sexp) (val-is-also-ok sexp) (...))
(defun infer-case (sexp varenv fenv)
  (let ((val (cadr sexp))
        (label-body-pairs (cddr sexp)))
    (infer val varenv fenv)
    (let ((ret-type-lst nil))
      (dolist (body (mapcar #'cdr label-body-pairs))
        (push (infer `(progn ,@body) varenv fenv) ret-type-lst))
      (calc-return-union ret-type-lst))))


;; send, send*
(defun infer-receiver (sexp varenv fenv)
  "
  レシーバータイプの推論を行う。実験のため *receiver-mode* で挙動の制御を行う。
  nil のとき、何もせず any を返す
  t のとき、(send x :method arg) の :method から x を絞り込み any を返す
  "
  (if *receiver-mode*
    (infer-receiver-precise sexp varenv fenv)
    (infer-receiver-coarse sexp varenv fenv)))


(defun infer-receiver-coarse (sexp varenv fenv)
  any-class)


;;; infer_receiver_helper.l の補助関数に頼る
(defun infer-receiver-precise (sexp varenv fenv)
  (let* ((receiver (cadr sexp))
         (actual-type (infer receiver varenv fenv))
         (selector (caddr sexp))
         (formal-type (method-to-class-or-group selector)))
    (formulate-actual formal-type actual-type)
    any-class))
```

付録 B

合併型および交差型を求めるアルゴリズム

```
;;; this is an extracted program from type_class_tree.l
(require :fix "fixed_hash.l")
(require "type_util.l")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 型を表現するためのクラス定義 ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass number :super object)
(defclass int :super number)
(defclass float :super number)
(defclass rational :super number)
;;; 型を細分化
(defclass mat :super array)
(defclass bool :super symbol)
(defclass nil-class :super bool)
(defclass t-class :super bool)
;;; 独自クラス
(defclass original :super object)
(defclass any-class :super original)
(defclass never-class :super original)
;;; Union を表現するためのクラス
(defclass group :super object :slots (class-list))
```

```
;;;;;;;;;;;;;;;
;;; 操作関数 ;;;
;;;;;;;;;;;;;;;


;;; 定義済みのクラスを列挙した list
(defparameter *class-list* (system:list-all-classes))
;;; 上記の vector
(defparameter *class-vec* (apply #'vector *class-list*))
;;; クラス数
(defparameter *class-num* (length *class-vec*))
;;; {クラスオブジェクト: インデックス} なる辞書
(defparameter *class-index-dict*
  (let ((ht (fix:make-hash-table :size (* *class-num* 2))))
    (dotimes (i *class-num*)
      (fix:sethash (svref *class-vec* i) ht i))
    ht))


(defun index-to-class (num) (svref *class-vec* num))


(defun class-to-index (cls)
  (assert (fix:gethash cls *class-index-dict*) "class-to-index(): no class found")
  (fix:gethash cls *class-index-dict*))


;;; クラスツリーの隣接リスト表現（親クラス -> 子クラスへ有向辺を張っている）
(defparameter *class-tree*
  (let* ((size *class-num*)
         (adj (instantiate vector size)))
    ;; u, v はクラスのインデックス表記 (int)
    ;; u (parent) -> v (child) へ辺を張るよう adj へ追記
    (flet ((add-edges (u v)
             (push v (svref adj u))))
        (dolist (child *class-list*)
          (unless (equal child object)
            (add-edges (class-to-index (send child :super))
```

```
                        (class-to-index child)))))
    adj))


;; reachable-list-by-index の補助関数
(defun group-or-classp (x) (or (derivedp x group) (classp x)))


;; reachable-list-by-group-or-class の補助関数
(defun reachable-list-by-index (num)
  "
  *class-tree* において num ノードから到達可能なノードのリストを返す
  "
  (let ((reachable nil)
        (visited (instantiate vector *class-num*)))
    (labels ((dfs (u)
               (setf (svref visited u) t)
               (push u reachable)
               (dolist (v (svref *class-tree* u))
               (if (null (svref visited v))
                   (dfs v)
                   (format t "warn: *class-tree* is not tree?~%")))
             ))
        ; num を開始点として探索
        (dfs num))
    reachable))


;; class-intersec の補助関数
(defun reachable-list-by-group-or-class (cls-exp)
  "
  クラスオブジェクトまたは group のインスタンスを受け取り、
  そのクラス集合の全サブクラスのインデックスをリストにまとめて返す
  "
  (assert (group-or-classp cls-exp) "group or class expected")
  (if (classp cls-exp)
    (reachable-list-by-index (class-to-index cls-exp))
    (let ((buf nil)
```

```
          (lst nil))
      (dolist (cls (group-class-list cls-exp))  ; slot access
        (setq lst (reachable-list-by-index (class-to-index cls)))
        (push-cons-cons lst buf))
      buf)))


;; class-intersec の補助関数
(defun index-list-to-group-or-class (ind-lst)
  "
  クラスの集合を表すインデックスのリストを受け取り、
  クラスオブジェクトないし group インスタンスを返す
  ind-lst が nil の場合、never-class を適切に返す
  "
  (if (null ind-lst)
    never-class
    (let ((cls-ind-lst nil)
          (visited (instantiate vector *class-num*)))
      ;; 探索は Top である object を示すルートノードから始める
      (labels ((dfs-coloring (u)
                 (setf (svref visited u) t)
                 (let ((reachable (reachable-list-by-index u)))
                   (cond
                     ((subsetp reachable ind-lst)
                       (push u cls-ind-lst))
                     ((not (null (intersection reachable ind-lst)))
                       (dolist (v (svref *class-tree* u))
                         (if (null (svref visited v))
                           (dfs-coloring v)
                           (format t "warn: *class-tree* is not tree?~%")))
                       )
                     (t nil))))
               )
        ; ルートを開始点として探索
        (dfs-coloring (class-to-index object)))
      (if (= (length cls-ind-lst) 1)
        (index-to-class (car cls-ind-lst))
        (make-instance group :class-list
```

```
            (mapcar #'index-to-class cls-ind-lst)))
    )))


(defun class-intersec (a b)
  "
  a, b はクラスオブジェクトないし group のインスタンス
  サブクラス関係において a と b の交わりを求める
  (a, b の積集合を表すクラスオブジェクトないし group のインスタンスを返す)
  空集合となった場合 never-class を適切に返す
  "
  (cond
    ((or (equal a never-class) (equal b never-class)) never-class)
    ((equal a any-class) b)
    ((equal b any-class) a)
    (t (index-list-to-group-or-class
         (intersection (reachable-list-by-group-or-class a)
                       (reachable-list-by-group-or-class b))))
  ))


(defun class-union (a b)
  "
  a, b はクラスオブジェクトないし group のインスタンス
  サブクラス関係において、 a と b の結びを求める
  (a, b の和集合を表すクラスオブジェクトないし group のインスタンスを返す)
  "
  (cond
    ((or (equal a never-class) (equal b never-class)) never-class)
    ((equal a any-class) b)
    ((equal b any-class) a)
    (t (index-list-to-group-or-class
         (union (reachable-list-by-group-or-class a)
                (reachable-list-by-group-or-class b))))
  ))
```