関数型プログラミング言語における
通信記述の型付け

今井 敬吾

概要

本論文は，信頼性の高い分散システムを構築するための通信記述の基盤を与えることを目的として，プロセス計算の 1 つである π 計算に基づくプログラミング技法を提案する．具体的には，関数型プログラミング言語 Haskell において，非同期 π 計算に基づく通信コンビネータを実装する手法と，π 計算のセッション型システムを Haskell の型システム上で実現する手法を示す．さらに，セッション型システムの理論的な基礎付けとして，主部簡約を定式化し証明する．

今日において，分散システムは極めて一般的である．代表的なものは，インターネットとそれを活用したアプリケーションであり，様々な産業に深く根ざし現在でもその利用は拡大し続けている．今日における自動車は複数の ECU が連携して動作する，高い信頼性が要求される分散システムの一つである．分散システムは，その複雑さにも関わらず様々なレベルにおいて産業と生活の基盤を成しているため，信頼性を備えているだけでなく，効率よく構築されることが求められている．

分散システムでは，複数のノードそれぞれにおいて計算が自律的に進行する．個別のノードが役割を果たし分散システムが全体として正しく動作するためには，各ノードの内部動作が正しく進行するだけでなく，必要なメッセージが適切なタイミングで交換される必要がある．しかしながら，分散システムの連携動作は並行性をもち非決定的であるため，通信のみに着目した場合でもシステムが取り得る状態の数は膨大な数に上る．

分散システムの通信動作を網羅的に解析するためには，個別の通信記述が全体として整合していることを確認しなければならない．ここで通信記述とは，ノードの振る舞いに関する記述のうち，通信に関する部分である．通信記述の整合性を確認するためには，陽に，ないし暗黙に定められた通信手順との適合性の検査が必要となる．インターネットの中心的存在である HTTP や，SMTP などは代表的な通信手順である．分散システムの開発においては，厳密な計算モデルに基づきソフトウェアを構成し，通信記述の整合性を解析できることが望ましい．

プロセス計算は，プロセスと呼ばれる計算主体の間で発生する通信に着目した形式的な計算体系である．これらの体系においては，計算は各々のプロセスが保持

する通信路を媒体とした同期的な通信により進行する．Milner による CCS，Hoare による CSP，Milner, Parrow, Walker による π 計算などが知られている．形式的には，名前が通信路の端点として扱われ，計算は各々のプロセスが保持する名前を介した同期的な通信により進行する．

π 計算は通信路を介して名前それ自体を送受する機構を備えたプロセス計算である．これにより，プロセス間のリンク構造の動的な変化を表現できる．π 計算では，名前の送受や生成を行う並行プロセスを柔軟に記述でき，型理論やプロセスの等価性に関する多くの解析技法が開発されている．さらに，名前によりポインタや参照といった計算資源をも表現できる．このため，π 計算は様々な抽象度において並行分散ソフトウェアの形式的な基礎として多く用いられる．

Haskell は強力な抽象化能力をもつ関数型プログラミング言語の一つである．Haskell は強く型付けされた言語であり，厳密な型検査の能力をもつ．このため，Haskell で記述されたソフトウェアは基本的に信頼性が高いとされている．さらに抽象化の技法として，関数型プログラミング言語ではコンビネータや埋め込み言語などと呼ばれる技法が知られている．例えば構文解析やリレーショナルデータベース等の特定分野における語彙を，言語処理系を拡張することなく実現できるため，実装者と利用者の双方に利点がある．

本論文で提案する枠組みにより，静的型付けの信頼性のもとで通信記述の整合性を確保でき，通信について信頼性をもつ分散システムを構築できる．さらに，既存の柔軟かつ信頼性の高い静的検査をもつ Haskell を利用するため，利便性と通信の信頼性を両立できる．

まず，この構想の基礎として非同期局所化 π 計算 (ALπ) を基礎としたネットワークプログラミングの枠組みを提案する．この枠組みの特徴は，Haskell の型システムにより，ALπ の通信路が型付けされ，内部動作と通信の両面において実行時エラーが発生せず信頼性が高いことである．さらに，フレームワークの実装において ALπ の動作意味を直接的に模倣しており，π 計算における既存の技法を適用して振る舞いを解析できる．通信記述においては，π 計算がもつ表現能力をそのまま利用できることが望ましいが，計算資源の多くは計算機に局所化されており，名前とアドレスやポートといった計算資源を一対一に対応づけられない．ALπ は，通信プリミティブを非同期に制限し，さらに通信路の入力能力の移動を局所的に制限した体系である．名前の局所性により，通信路と計算資源の対応が明白である利点がある．このため，通信記述の基本的な道具として，π 計算の名前渡しの能力を使った記述ができる．名前の局所性を Haskell の型システムで表現するために，

π 計算のサブタイプ関係を表現する型クラスを導入する．

この枠組みにおいては，ALπ のプロセスは Haskell のモナドを表現したコンビネータとして提供される．Haskell ではモナドはプログラミングの基礎的な道具として極めて頻繁に用いられるため，プロセス計算の構文をそのままプログラムコードに記述するよりも利用しやすいためである．この枠組みの有用性を示すため，インスタントメッセンジャーの例を与える．

一方，通信記述においては，単一の通信路において異種のメッセージを送受信する通信手順に従わなければならない場合も多い．ALπ が基礎におく型システムは通信路について同種のメッセージしか扱えず制限が強いため，そのような通信手順に関する静的な解析ができない．

π 計算のセッション型システムは，異種のメッセージの系列（セッション）の通信を表現できる型システムであり，そのような通信手順を伴う通信記述の信頼性を静的解析により保証できる．セッション型は通信路に割り当てられた通信手順を表現し，通信手順の違反は型エラーとして検出できる．しかしながら，通常のプログラミング言語の型システムでは，セッション型を直接に表現できない．まず，ほとんどの型システムでは，識別子にただ一つの型を割り当てるため，通信路の使用順序を型で表現できない．さらに，線形型のような型機能が無いため，スレッド間で通信が干渉しないことも保証できない．

このような型を Haskell の型にエンコードするために，様々な挑戦がなされてきた．Sackman と Eisenbach による実装は，豊富な機能を持つものの，セッション型が推論されず，型をソースコードに陽にする必要があり，通信の記述が煩雑になりがちだった．近年の Pucella と Tov による実装は，型推論によりセッション型が推論されるものの，複数の通信路を扱うために本来不要な辻褄合わせの操作をしなければならず，多くの通信路を扱う場合にやや問題があった．

そこで本論文では，従来手法の問題を解決し，型推論を伴うセッション型システムを Haskell の型システム上に実現する手法を提案する．Pucella と Tov による既存の実装 [PT08] では，型推論によりセッション型が推論される．しかしながら複数の通信路を扱うために通信路のスタックを操作しなければならず，多くの通信路を扱う場合にやや問題があった．我々の実装では，de Bruijn レベルによるエンコーディングを用いて，複数の通信路を用いた場合においてもスタック操作などが必要ない，ほぼ全自動のセッション型推論を実現する．枠組みの有用性を示すため，セッション型で型付けされた複数の通信路を用いた SMTP クライアントの実装を示す．

さらに，我々のセッション型の実装の理論的基盤を堅固にするために，本論文ではセッション型システムの主部簡約について異なる定式化を行い，その型安全性を確かめる．本田らにより提案されたオリジナルのセッション型システムは，型システムの基礎的な性質である主部簡約が成立しない場合があることが，吉田らにより指摘されている．我々は本田らのセッション型システムに対する修正を与え，主部簡約を定式化し，これを用いて型安全性が成立することを示す．鍵となるアイデアは，型がつかないプロセスに簡約される通信パターンにおいても，依然として安全性が保たれていることを示すために，型付けを拡張することである．具体的には，プロセスの部分項において型安全が成立しない場合においても，その部分には到達しないことを主部簡約として定式化し，その証明を用いて，型安全性を示した．

我々の技法により，Haskell の従来の静的型付けに加えて，セッション型システムによる通信手順の整合性検査が Haskell コンパイラにより提供される．一連の結論として，分散システムを，高い信頼性で，かつ効率よく記述できるようになる．

目 次

図 目 次

第1章　序論

1.1　背景

今日において，分散システム [Lyn96] は極めて一般的である．代表的なものは，インターネットとそれを活用したアプリケーションであり，様々な産業に深く根ざし現在でもその利用は拡大し続けている．今日における自動車は複数の ECU が連携して動作する，高い信頼性が要求される分散システムの一つである [EJ09]．分散システムは，その複雑さにも関わらず様々なレベルにおいて産業と生活の基盤を成しているため，信頼性を備えているだけでなく，効率よく構築されることが求められている．

分散システムでは，複数のノードそれぞれにおいて計算が自律的に進行する．個別のノードが役割を果たし分散システムが全体として正しく動作するためには，各ノードの内部動作が正しく進行するだけでなく，必要なメッセージが適切なタイミングで交換される必要がある．しかしながら，分散システムの連携動作は並行性をもち非決定的であるため，通信のみに着目した場合でもシステムが取り得る状態の数は膨大な数に上る．

分散システムの通信動作を網羅的に解析するためには，個別の通信記述が全体として整合していることを確認しなければならない．ここで通信記述とは，ノードの振る舞いに関する記述のうち，通信に関する部分である．通信記述の整合性を確認するためには，陽に，ないし暗黙に定められた通信手順との適合性の検査が必要となる．インターネットの中心的存在である HTTP[FGM+99] や，SMTP[Kle08] などは代表的な通信手順である．分散システムの開発においては，厳密な計算モデルに基づきソフトウェアを構成し，通信記述の整合性を解析できることが望ましい．

プロセス計算 [BPS01] は，プロセスと呼ばれる計算主体の間で発生する通信に着目した形式的な並行計算体系であり，厳密さをもつ．これらの体系においては，計算は各々のプロセスが保持する通信路を媒体とした同期的な通信により進行する．Milner による CCS[Mil80]，Hoare による CSP[Hoa85]，Milner, Parrow, Walker

による π 計算 [MPW92, Mil99, SW01] などが知られている．形式的には，名前が通信路の端点として扱われ，計算は各々のプロセスが保持する名前を介した同期的な通信により進行する．

π 計算は通信路を介して名前それ自体を送受する機構を備えたプロセス計算である．これにより，分散システムにおけるリンク構造の動的な変化を柔軟に表現できる．π 計算においては，型理論やプロセスの等価性に関する多くの解析技法が開発されている [SW01]．さらに，名前によりポインタや参照といった計算資源をも表現できる．このため，π 計算は並行分散プログラミングの形式的な基礎として多く用いられてきた [KMK01, PT00, SWU10, MMK+05, PH06].

型理論は，プログラムの実行の抽象的な側面を表す技法として研究されてきた．数理論理学とラムダ計算から発展した型理論は，プログラミング言語の静的検査において非常に大きな役割を果たしている [Pie02]．型理論における個々の型システムは，値，関数やオブジェクトといった計算資源に型を割り当て，その論理的な整合性を比較的少ない計算量で検査できる枠組みである．さらに，多相性により柔軟なプログラムの合成が可能になる．型推論 [DM82] により，プログラムにおける型の記述を省略できる．今日では，型理論は，ソフトウェアの信頼性を確認するための軽量かつ信頼性の高い手法の一つである．

型理論の恩恵を最も大きく活用している言語の一種に関数型プログラミング言語がある．静的に型付けされた関数型プログラミング言語は，安全性を考慮しつつ，多相性による柔軟かつ再利用性の高い記述が可能である．代表的な静的型付けの関数型プログラミング言語に ML[MTHM97] や Haskell[Has10] がある．関数型プログラミング言語においては，関数を第一級市民として扱えることを基礎として，高階関数によるリスト処理，遅延評価，Functional Reactive Programming[EH97] など様々なプログラミング技法が知られている．関数型プログラミング言語は，近年では学界のみならず産業界からも様々な観点で注目されている．

Haskell[Has10] は型機能に関して大きな柔軟さをもち，型によりプログラムの安全性を保証するための様々な技術が開発されている [Mcb02]．Haskell の型システムは関数依存 [Jon00] や型族 [SJCS08] といった枠組みによって拡張され，プログラムに関する多様な性質を型で表現できる．Haskell は関数型プログラミング言語のひとつであり，純粋関数的で遅延評価の機構をもつ．Moggi による圏論 [Lan05] のモナドを用いた様々な計算作用の定式化 [Mog91] は，現在において関数型プログラミングに欠かすことのできないプログラミング技法の一つである．Haskell における抽象化の技法として，関数型プログラミング言語ではコンビネータや埋め

込み言語などと呼ばれる技法が知られている．例えば構文解析 [LM01] やリレーショナルデータベース [BHA+04] 等の特定分野における語彙を，言語処理系を拡張することなく実現できるため，実装者と利用者の双方に利点がある．Haskell において，モナド，コンビネータや高階関数などを用いて抽象度の高いソフトウェアを構築する際にも，型システムが重要な役割を果たす．

π 計算のセッション型システム [HVK98] は，通信記述において，ある通信路の両端点における通信手順が互いに整合することを保証する型システムの一つである．セッション型システムに代表される振る舞い型システム [IK04, KSS00] は，プロセスの通信に関する振る舞いを抽象化した型を割り当てる．このため，通信に関してより多くの解析が可能となる．しかしながら，既存の型システムとは異なり，プログラムの進行に従い型が変化する性質をもつため，既存のプログラミング言語の型システムとはうまく整合せず，この分野における応用の妨げになっている．

1.2 目的

本論文は，プログラミング言語において高い信頼性のもとで通信を記述できるプログラミングの枠組みを与えることを目指し，プロセス計算の一種である π 計算の型システムと，その関数型プログラミング言語における実現の両方を提案する．Haskell における，次の２つの体系に基づくプログラミング技法を個別に提案し，分散システムの高信頼な実装手法を示す．

- 非同期局所化 π 計算
- セッション型システムにより型付けされた π 計算

さらに，型システム型の重要な性質である主部簡約をセッション型において確立して，型安全性の基礎づけを行う．

Haskell で記述されたソフトウェアは型安全性のために基本的に信頼性が高いとされている．Haskell において，π 計算の型付けを基礎とした通信記述のための枠組みが確立すれば，より信頼性の高い分散システムの構築が可能になる．

まず，この構想の基礎として，非同期局所化 π 計算 (ALπ)[Mer00] を基礎としたネットワークプログラミングの枠組みを提案する．この枠組みの特徴は，Haskell の型システムにより，ALπ の通信路が型付けされ，内部動作と通信の両面において実行時エラーが発生せず信頼性が高いことである．さらに，フレームワークの

実装において ALπ の動作意味を直接的に模倣しており，π 計算における既存の技法を適用して振る舞いを解析できる．ALπ は，通信プリミティブを非同期に制限し，さらに通信路の入力能力を局所的に制限した体系である．通信記述においては，π 計算がもつ表現能力をそのまま利用できることが望ましいが，計算資源の多くは計算機に局所化されており，π 計算の名前はそのままでは計算資源と一対一に対応づけられない．基本的なモデルとして ALπ を採用することで，名前の局所性により，通信路と計算資源の対応については明白な関係が保たれるという利点がある [Mer00]．これにより，通信記述の基本的な道具として，π 計算の名前渡しの能力を使った記述ができる．

一方，通信記述においては，単一の通信路において異種のメッセージ系列を入出力する通信手順に従わなければならない場合も多い．セッション型システム [HVK98] は，そのような通信手順の適合性を型検査により確認できる体系である．このような型を Haskell の型にエンコードするために，様々な挑戦がなされてきた [NT04, SE08, PT08]．そのなかでも，Pucella と Tov による実装 [PT08] は，型推論によりセッション型が推論される．しかしながら複数の通信路を扱うために通信路のスタックを操作しなければならず，多くの通信路を扱う場合にやや問題があった．我々の実装では，de Bruijn レベルによるエンコーディングを用いて，複数の通信路を用いた場合においてもスタック操作などが必要ない，ほぼ全自動のセッション型推論を実現する．この枠組みは，通信記述が互いに整合していることを型により確認できるため，分散システムの実装において非常に有用である．有用性を示すため，複数の通信路を用いた SMTP クライアントの例を与える．

さらに，我々のセッション型の実装の理論的基盤を堅固にするために，本論文ではセッション型システムの主部簡約を定式化し，その型安全性を確かめる．本田らにより提案されたオリジナルのセッション型システム [HVK98] は，型システムの基礎的な性質である主部簡約が成立しない場合があることが，吉田らにより指摘されている．我々は本田らのセッション型システムに対する修正を与え，主部簡約を定式化し，これを用いて型安全性が成立することを示す．鍵となるアイデアは，型がつかないプロセスに簡約される通信パターンにおいても，依然として安全性が保たれていることを示すために，型付けを拡張することである．具体的には，プロセスの部分項において型安全が成立しない場合においても，その部分には到達しないことを証明することで，型安全性を示した．

我々の技法により，Haskell の従来の静的型付けに加えて，π 計算の型システムによる通信手順の整合性検査が Haskell コンパイラに導入される．一連の結論とし

て，分散システムを，高い信頼性で，かつ効率よく記述できるようになる．

1.3 本論文の構成

本論文は次の順に構成されている．第 2 章では本論文の基礎となる π 計算の構文，動作意味と複数の型システムを導入する．第 3 章では，非同期 π 計算を基礎とした Haskell におけるネットワークプログラミングの枠組みを与える．第 4 章では，セッション型システムを Haskell に導入するプログラミング技法を与える．第 5 章では，Honda らのセッション型システムを修正した体系を提案し，この体系における主部簡約を定式化し証明する．最後に，第 6 章で本論文のまとめと今後の課題を述べる．本論文の付録には，第 4 章で与えたセッション型の実装の全体を示す．

第 2 章は Pierce と Sangiorgi[PS96], Honda[HVK98] らの文献を本論文のために修正して紹介している．第 3 章の内容は [IYA06] で公表ずみである．第 4 章の内容は [IYA07], [IYA08],[IYA09] に基づいており，[IYA11] で公表した内容の著者による和訳である．第 5 章の内容の一部は [IYA10b] で公表済みである．

第2章　π計算

プロセス計算は，プロセスと呼ばれる計算主体の間で発生する通信に着目した形式的な計算体系である．これらの体系において，可算無限個の**名前**が通信路の端点として扱われ，計算は各々のプロセスが保持する名前を介した同期的な通信により進行する．

π計算 [MPW92, Mil99, SW01] は通信路を介して名前それ自体を送受する機能を備えたプロセス計算である．これにより，リンク構造の動的な変化を伴うモバイルプロセスの抽象モデルを表現できるだけでなく，名前をオブジェクト [Wal95] やヒープ [PH06] といった具体的な計算資源に対応づけることもできる．それゆえ，π計算は今日では並行分散プログラミングの形式的な基礎として多く用いられる [KMK01, PT00, SWU10, MMK+05, PH06]．本項では多岐にわたる文献すべてを網羅しないが，本論文で扱う範囲のπ計算に関連する主要な結果を記す．π計算においては，それまでのプロセス計算よりも多くの振る舞い等価性のレベルが研究され [MPW92, HT92, San96]，並行プログラムの意味に関する理論的基礎を与えた．非同期π計算 [HT91, Bou92] は，プロセスが出力動作を行った後はただちに停止しなければならないという制限を加えた体系である．この体系においては，出力プロセスは単に通信路に向けて送られたメッセージそのものとみなせる．非同期π計算のように，能力を制限したπ計算も強力な表現能力をもつことが確かめられている．

特に本論文と密接に関わるのは型システムに関する体系群である．当初，複数の名前を同時に送受できるポリアディックπ計算において，送受される名前の個数を静的に確かめるために，ソートと呼ばれる仕組みが導入された [Mil93]．通信路の使用に関する入力と出力の向き付けを，サブタイプ機構を使い表現した体系が知られている [PS96]．セッション型システム [THK94, HVK98, YV07, CDCY07, CdY08] は，単一の種類のメッセージしか送受できない型システムと異なり，異種のメッセージの系列の送受について型を付ける型システムである．他にも線形型 [KPT99]，デッドロックフリー性 [KSS00, CDCY07, CdY08] などを特徴づける型システムがある．さらに，そのような振舞い型システムの包含する体系として，Generic Type

System [IK04] が提案されている．

本章では，π計算の構文および動作意味と，後の章で用いる型システムを導入する．型システムは Sangiorgi と Pierce の i/o 型と，本田らのセッション型を導入する．π計算の記法は様々な流儀があるが，本論文で主に取り扱うセッション型に関連する論文で近年使われている記法（[HYC08] などの記法）に統一している．i/o 型については，[PS96] を用いる．

2.1 構文

プロセスが用いる通信路として，名前の可算無限集合 $\mathcal{N}$ を仮定する．名前 $s, s', t, t', u, u', r, r' \cdots$ は $\mathcal{N}$ の上を動く．以後，名前と通信路を特に区別しない．

定義 2.1 (Processes) プロセス P の構文は次の文法で与えられる．

$$P \quad ::= \quad s!\langle t\rangle;P \;\big|\; s?(t);P \;\big|\; P|Q \;\big|\; (\nu s)P \;\big|\; *s?(t);P \;\big|\; \mathbf{0} \tag{2.1}$$

$s?(t);P$ と $(\nu t)P$ は P に出現する自由な t を束縛する．$(\nu s)(\nu s')\cdots P$ を省略して $(\nu \tilde{s})P$ と書く．P に出現する自由な名前の集合を $fn(P)$ と書く． □

各構文の直観的意味は次の通りである．

- （出力） $s!\langle t\rangle;P$ は s を介して名前 t を送信し，P に遷移する．これは 名前渡しあるいは通信路渡しと呼ばれる．
- （入力） $s?(t);P$ は s を介して名前を受信し，それを P の自由な t に束縛する．
- （並行） $P|Q$ は 並行に動作する P と Q を表す．
- （制限） $(\nu s)P$ は P に出現する自由な s を隠す．新しい通信路を生成して s に束縛する名前生成とみなすこともできる [河辺 02]．
- （入力複製） $*s?(t);P$ は プロセス $s?(t);P$ の無限の並行合成 $s?(t);P \mid s?(t);P \mid \cdots$ とみなされる．直観的には通信路 s で待ち受けるサーバーを表現している．
- （終了） $\mathbf{0}$ は停止したプロセスを表す．

$$P \equiv Q \text{ if } P \equiv_\alpha Q \qquad P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu s)\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)P \mid Q \equiv (\nu s)(P \mid Q) \text{ if } s \notin fn(Q)$$
$$*s?(t);P \equiv s?(t);P \mid *s?(t);P$$

図 2.1: 構造合同性

π 計算の通信プリミティブは同期的であり，受信側が通信可能になるまで送信側はブロックする．π 計算には，他に選択 $P+Q$，一般化された複製 $*P$，名前のマッチング $[s=t]P$，再帰などの構文が導入されることがあるが，本論文では用いない．選択とマッチングは殆どの場合，それを含まない形に変換できることが知られている [Nes00, NP00, Pal03]．再帰と一般化された複製は互いに等価なものに変換できる [Mil93]．同様に，複製は等価な入力複製に変換できる．

定義 2.2 (プレフィクス，サブジェクト，オブジェクト) プロセス P は，$s!\langle t\rangle;P$ あるいは $s?(t);P$ の形であるときプレフィクスあるいはプレフィクスされたプロセスと呼ばれる．これらのプロセスにおいて s をサブジェクトと呼ぶ．さらに，$s!\langle t\rangle;P$ における t をオブジェクトと呼ぶ． □

束縛された名前の名前替えは，慣例に従い α 変換と呼ぶ．Q が P の α 変換であるとき，$P \equiv_\alpha Q$ と書き，構文的に等価であるとみなす．さらに，$P|Q$ と $Q|P$ のような構造的に等価と見なせるプロセスを，等価関係である構造合同性を用いて同一視する．

定義 2.3 (構造合同性) 構造合同性は図 *2.1* を満たす最小の関係である． □

2.2 動作意味

π 計算の動作意味は，簡約関係による方法とラベル付き遷移関係による方法の２通りで与えることができる．計算の進行という観点からいえば，両者は等価である．本章では両方の意味定義を導入するが，後の各章で参照するのはどちらか一方のみである．

$$\textsc{Pass}: \frac{}{s!\langle t\rangle;P \mid s?(t');Q \longrightarrow P \mid Q\{t/t'\}} \qquad \textsc{Scop}: \frac{P \longrightarrow P'}{(\nu s)P \longrightarrow (\nu s)P'}$$

$$\textsc{Par}: \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \textsc{Str}: \frac{P \equiv P' \ \wedge \ P' \longrightarrow Q' \ \wedge \ Q' \equiv Q}{P \ \longrightarrow \ Q}$$

図 2.2: 簡約関係

α	::=	$s!\langle t\rangle$	出力
	|	$s?(t)$	入力
	|	$s!(t)$	束縛出力
	|	τ	内部動作

図 2.3: 遷移ラベル

2.2.1 簡約関係

簡約関係は，π 計算のプロセスの内部動作による計算の進行を形式的に定義する．

定義 2.4 (簡約関係) 簡約関係 $\longrightarrow$ は図 *2.2* の規則で帰納的に定義される関係である．$\longrightarrow^*$ は $\longrightarrow$ の反射推移閉包である．$P \longrightarrow P'$ なる P' が存在しないとき $P \not\longrightarrow$ と書く． □

2.2.2 ラベル付き遷移関係

ラベル付き遷移系は，プロセスの動作を環境との相互作用の観点で形式化する方法である．相互作用を表すラベルを図 2.3 で定義する．$s!\langle t\rangle$は t を通信路 s を介して出力である．$s?(t)$は t の s を介した入力である．ここで t は束縛された名前である．$s!(t)$は s を介してプライベートな名前を出力する動作を表す．ここでも t は束縛された名前であり，スコープ射出 (scope extrusion) 規則によって，他の名前と区別される．τ は他のプロセスから観測できない内部動作を表す．$bn(\alpha)$ は α における束縛された名前を表す．$fn(\alpha)$ は α に出現する自由な名前の集合を表す．

定義 2.5 (ラベル付き遷移関係) ラベル付き遷移関係 $\xrightarrow{\alpha}$ は図 *2.4* の規則で帰納的に定義される *3* 項関係である．図 *2.4* では Par-L, Comm-L, Close-L においてそれぞれ P と Q を入れ替えた Par-R, Comm-R, Close-R 規則が省略されている．

$$\textsc{Out}: \frac{}{s!\langle t\rangle;P \xrightarrow{s!\langle t\rangle} P} \qquad \textsc{Inp}: \frac{}{s?(t);P \xrightarrow{s?(u)} P\{u/t\}}$$

$$\textsc{RepInp}: \frac{}{*s?(t);P \xrightarrow{s?(u)} P\{u/t\} \mid *s?(t);P}$$

$$\textsc{Par-L}: \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \emptyset \qquad \textsc{Comm-L}: \frac{P \xrightarrow{s!\langle t\rangle} P' \qquad Q \xrightarrow{s?(t)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Res}: \frac{P \xrightarrow{\alpha} P'}{(\nu s)\,P \xrightarrow{\alpha} (\nu s)\,P'}\ \text{if } s \notin fn(\alpha) \qquad \textsc{Open}: \frac{P \xrightarrow{s!\langle t\rangle} P'}{(\nu t)P \xrightarrow{s!(t)} P'}\ s \neq t$$

$$\textsc{Close-L}: \frac{P \xrightarrow{s!(t)} P' \qquad Q \xrightarrow{s?(t)} Q'}{P \mid Q \xrightarrow{\tau} (\nu t)(P' \mid Q')}\ t \notin fn(Q)$$

図 2.4: ラベル付き遷移関係

事実 2.1 ($\longrightarrow$ と $\xrightarrow{\tau}$ の一致 [Mil92]) $P \longrightarrow P'$ ならば，かつそのときのみ，ある P'' が存在し $P \xrightarrow{\tau} \equiv P'$. □

2.3 i/o 型システムと非同期局所化 π 計算

本節で，非同期局所化 π 計算（ALπ）を導入する．本節で導入する型システムは 3 章でのみ用いる．本節で与える値，型，型環境，型判定は次節のセッション型における定義とは互いに無関係である．

ALπ は，一般的な π 計算に，非同期性および局所性の制約を加えた体系である．

非同期性 非同期 π 計算 (Asynchronous π-calulus) [HT91][Bou92] は出力プレフィクスの継続が $\mathbf{0}$ に限られる体系である．非同期 π 計算では出力プロセス $s!\langle t\rangle;P$ において，P は必ず $\mathbf{0}$ である．すなわち，出力したメッセージが相手に到達したかどうかを単一ステップの出力動作では確認できない．このような条件は現実のネットワークソフトウェアにおいても一般的であるため，非同期 π 計算はネットワークプログラミングの枠組みの一つとして適している．

局所性 π 計算における局所性 (locality) とは，

- 入力プロセス $s?(t);P$ について，束縛された名前 t が継続する P で入力プレフィクスに出現しない
- 名前の同一性を検査するマッチング $[\,x = y\,]\,P$ の構文を認めない

という制約である．このような体系は局所π計算 (Local π-calculus) あるいは局所化π計算 (Localized π-calculus) と呼ばれる [Mer00][Yos02]．局所化π計算では，通信路の出力能力のみをやり取りする．入力能力は $\boldsymbol{\nu}$ 演算子によってしか導入できない．例えばプロセス P，Q 間で通信路 s の入力能力を共有する場合，$(\boldsymbol{\nu} s)(P|Q)$ の形をとらなければならない．このことは，同一の通信ホストに属するプロセス間でのみ入力能力が移動できることを表している．

本節では，通信の入力能力と出力能力を型で区別できる i/o 型 [PS96] を用いて局所化π計算を定式化する．

2.3.1 構文と型システム

定義 2.6 (ALπ の構文と意味) ALπ の構文は，π計算の構文から同期出力 $s!\langle t\rangle;P$ と制限 $(\boldsymbol{\nu} s)P$ を除き，非同期出力 $s!\langle t\rangle;\mathbf{0}$ と型付きの名前制限 $(\boldsymbol{\nu} s : L)P$ を加えたものである．ここで L は後に定義するリンク型である．ALπ の動作意味は，定義 *2.5* のラベル付き遷移系で与える． □

ALπ においては，名前以外の**値**もメッセージとして扱う．値の集合 B は $\mathcal{N}$ に含まれるとする．$s!\langle t\rangle;\mathbf{0}$ を $s!\langle t\rangle$ と略すことがある．

定義 2.7 (ALπ の値型，リンク型，プロセス型) ALπ における型は，値型かリンク型のどちらかである．値型は基本型を含む．今後，基本型 B の集合を仮定する．値型 V は次の文法で定義される．

$$\begin{array}{rll} V ::= & & \textit{(値型)} \\ & B & \text{基本型} \\ \mid & \mathsf{o}V & \text{出力能力} \end{array} \tag{2.2}$$

リンク型 L は次の文法で表される．

$$\begin{array}{rll} L ::= & & \textit{(リンク型)} \\ \mid & \mathsf{o}V & \text{出力能力} \\ \mid & \mathsf{i}V & \text{入力能力} \\ \mid & \sharp V & \text{入出力能力} \end{array} \tag{2.3}$$

プロセス型は $\diamond$ である．通信型，リンク型，プロセス型をを S, T で表す． □

リンク型は，通信路の型である．リンク型は，通信路のもつ能力で出力能力 $\mathsf{o}V$，入力能力 $\mathsf{i}V$，入出力能力 $\sharp V$ の 3 通りがあり，それぞれ値型 V をもつ値の送受に使われる通信路の能力を表す．プロセス型はただ一つの型 $\diamond$ しか存在しない．

値型は，ALπ において送受可能なメッセージの型である．値型には基本型と通信路の出力能力型が含まれる．入力能力を含む型は値型ではない．この制限が ALπ の局所性を特徴づけている．

型環境は，名前に対する型の割当てである．

定義 2.8 (ALπ の型環境) ALπ の型環境は次の構文で与えられる．

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma \cdot x\colon S \tag{2.4}$$

□

ALπ におけるプロセスの型付け規則およびサブタイプ規則は，[PS96] の i/o 型のものを用いる．ここで，基本型の間にサブタイプ関係を仮定しない．i/o 型の型付け規則を図 2.5 に示す．

定義 2.9 (ALπ の型判定) 型判定 $\Gamma \vdash P : S$ は図 *2.7* の規則により帰納的に定義される 3 項関係である．$\Gamma \vdash P : \diamond$ のとき，Γ でプロセス P は型付け可能であるという．

□

π 計算のサブタイプ型システムでは，サブタイプ規則は Sub-Trans, Sub-II，Sub-OO，Sub-BB を含む．しかしながら，ALπ では，値型が常に $\mathsf{oo}\cdots B$ の形であるため，Sub-trans は必要ない．このため，本論文で用いる ALπ のサブタイプ関係は Sub-refl, Sub-$\sharp$I, Sub-$\sharp$O のみですべて導出できる．

命題 2.1 $mV \leq mW (m \in \{\mathsf{i}, \mathsf{o}, \sharp\})$ が，規則 Sub-refl, Sub-$\sharp$I, Sub-$\sharp$O, Sub-II, Sub-OO と Sub-BB から導出される時，$V = W$.

Proof: $\mathsf{o}V \leq \mathsf{o}W$ の場合のみ示す. V が基本型の時, $W \leq V$ は Sub-refl から導出されるため，$V = W$. 一方，$V = \mathsf{o}V'$ *and* $W = \mathsf{o}W'$ の時，$\mathsf{o}V' \leq \mathsf{o}W'$ は Sub-OO から導出される．帰納法の仮定より，$V' = W'$，よって $V = W$. $\mathsf{i}V \leq \mathsf{i}W$ と $\sharp V \leq \sharp W$ の場合も同様. □

従って，サブタイプ関係は Sub-II,Sub-OO と Sub-BB, Sub-Refl のみで得られる．

値と名前の型付け

$$\textsc{Tv-base}: \frac{}{\Gamma \vdash basval : B}\ basval \in B \qquad \textsc{Tv-name}: \frac{}{\Gamma, s : T \vdash s : T}$$

プロセスの型付け

$$\textsc{T-Par}: \frac{\Gamma \vdash P : \diamond \quad \Gamma \vdash Q : \diamond}{\Gamma \vdash P \mid Q : \diamond} \qquad \textsc{T-Nil}: \frac{}{\Gamma \vdash \mathbf{0} : \diamond} \qquad \textsc{T-Res}: \frac{\Gamma, s : L \vdash P : \diamond}{\Gamma \vdash (\nu s : L)P : \diamond}$$

$$\textsc{T-AOut}: \frac{\Gamma \vdash s : \mathsf{o}V \quad \Gamma \vdash t : V}{\Gamma \vdash s!\langle t\rangle;\mathbf{0} : \diamond} \qquad \textsc{T-Inp}: \frac{\Gamma \vdash s : \mathsf{i}V \quad \Gamma, t : V \vdash P : \diamond}{\Gamma \vdash s?(t);P : \diamond}$$

$$\textsc{T-RepInp}: \frac{\Gamma \vdash s : \mathsf{i}V \quad \Gamma, t : V \vdash P : \diamond}{\Gamma \vdash *s?(t);P : \diamond}$$

サブサンプション

$$\textsc{Subsumption}: \frac{\Gamma \vdash v : S \quad S \leq T}{\Gamma \vdash v : T}$$

サブタイピング規則

$$\textsc{Sub-refl}: \frac{}{T \leq T} \qquad \textsc{Sub-}\sharp\textsc{I}: \frac{}{\sharp V \leq \mathsf{i}V} \qquad \textsc{Sub-}\sharp\textsc{O}: \frac{}{\sharp V \leq \mathsf{o}V}$$

$$\textsc{Sub-trans}: \frac{S \leq S' \quad S' \leq T}{S \leq T} \qquad \textsc{Sub-BB}: \frac{W \leq V \quad V \leq W}{\sharp V \leq \sharp W}$$

$$\textsc{Sub-II}: \frac{V \leq W}{\mathsf{i}V \leq \mathsf{i}W} \qquad \textsc{Sub-OO}: \frac{W \leq V}{\mathsf{o}V \leq \mathsf{o}W}$$

図 2.5: i/o 型の型付け規則 [PS96]

2.3.2 ALπの表現能力

ALπはπ計算より表現能力が制限された計算体系であるが，豊富な記述能力をもつ．非同期性の制限のもとで，π計算のように通信相手にメッセージが到達したことを確認するためには，$(\nu t)(s!\langle msg, t\rangle;\mathbf{0} \mid t?();P)$ のようにしてメッセージにプライベートな名前（ここではt）を付加し，その名前を用いて通信相手からのACKを確認する方法がある [HT91][Bou92].

局所性は名前の入力能力の移動を制限する．一方，非同期π計算からALπへの変換が知られており，入力能力の自由な移動を模倣できる [Bor98] [1]．この方法では，入力した名前を任意の場所に転送する「入力マネージャ」というプロセスを用いる．ここでアドレスs_mをもつsの入力マネージャは$s_m \hookrightarrow s$と表記され，次の通り定義される：

$$s_m \hookrightarrow s \stackrel{def}{=} *s_m?(t);s?(u,v);t!\langle u,v\rangle;\mathbf{0}$$

ある名前sを出力する場合には，その名前の入力マネージャを表す名前s_mをsに付加して出力する．入力能力をもたないsで入力する場合には，その名前の入力マネージャs_mにプライベートな名前（例えばtとする）を出力する．次に入力マネージャは，sにおける入力をtに出力するため，結果として，入力能力をもたないsでの入力を，tでの入力で代替できる．

2.4 セッション型システム

この節ではπ計算のセッション型システムについて概観する．すべての定義は [YV07] による. 後で使わないため，セッションを初期化する構文は除いている．さらに，簡単のため再帰，複製，if文，選択などの構文を排除する．

セッション型システムはすべての通信路の使用法を追跡することによりエラーの不在を保証する．セッション型はあるセッションにおける通信路の端点の使われ方を示している．特にセッション型の枠組みにおいては，名前を通信路の端点と呼ぶことがある．

セッションとはプロセス間で共有されたある通信路を介した通信の列である．セッション型を議論する際には，π計算プロセスが送受するメッセージに値, ラベル，通信路の3種を含める．この体系を本節ではラベル分岐付きπ計算と呼ぶ．ラ

[1] 実際のところ，[Bor98] は局所化π計算より制限が強いプライベートπ計算（πI）への変換を定義している．

ベルはセッションの分岐先を指定するために用いられる．ラベルの集合は通信路・値と互いに素であるとされる．本論文では，left と right のみをラベルとして扱う．ラベルを２種に限定することは一般性を失わず，多くのラベルに容易に拡張できる．値は $\{\mathsf{true}, \mathsf{false}\}$ を含む集合とし，v の上を動くとする. 式 e の構文と評価関係 $e \downarrow v$ は与えられているものとする．可算無限個の変数の集合は x, y の上を動き，通信路・ラベル・値の集合と互いに素であるとする．変数は e にしか出現しない.

定義 2.10 (ラベル分岐付き π 計算の構文) ラベル分岐付きπ計算の構文は，定義 *2.1* の構文から複製入力 $*s?(t);P$ を除き，次の通り拡張した文法で与えられる：

$$\begin{aligned} P \quad ::= \quad & \cdots \\ & s!\langle e\rangle;P \mid s?(x);P \mid s \blacktriangleleft \mathsf{left};P \mid s \blacktriangleleft \mathsf{right};P \mid s \blacktriangleright \{\mathsf{left}\colon P_1,\ \mathsf{right}\colon P_2\} \end{aligned} \tag{2.5}$$

$s?(x);P$ は P に出現する自由な x を束縛する．$fv(P)$ は P に出現する自由な変数の集合を示す. $fv(P) = \emptyset$ ならば P を閉じたプロセスと呼ぶ.

$s!\langle e\rangle;P$, $s?(x);P$, $s \blacktriangleleft \mathsf{left};P$, $s \blacktriangleleft \mathsf{right};P$, $s \blacktriangleright \{\mathsf{left}\colon P_1, \mathsf{right}\colon P_2\}$ はプレフィクスされたプロセスである． □

新たに導入された構文は次の直観的意味をもつ.

- $s!\langle e\rangle;P$ は，まず e を評価し，その値を s に出力する.
- $s?(x);P$ は 値を s で入力し, それを P の自由な x に束縛する.
- $s \blacktriangleleft \mathsf{left};P$ と $s \blacktriangleleft \mathsf{right};P$ はラベルの送信による分岐ラベルの選択である.
- $s \blacktriangleright \{\mathsf{left}\colon P, \mathsf{right}\colon Q\}$ は受信したラベルに従い P か Q に分岐する.

定義 2.11 (ラベル分岐付き π 計算の動作意味) セッション型のついたπ計算の動作意味は，図 *2.2* の簡約関係に加え，図 *2.6* で帰納的に定義される２項関係 $\longrightarrow$ である.

メッセージ種別の不整合や通信路の競合はエラーと呼ばれ，セッション型システムにより排除される．例えば，次のプロセスはエラーである：

1. 送信者-受信者間でのメッセージ種別の不整合．例えば， $s \blacktriangleleft \mathsf{left};\mathbf{0} \mid s?(x);\mathbf{0}$ は，受信者が値を入力する一方で，送信者はラベルを出力しているため，エラーである．$s!\langle 1\rangle;\mathbf{0} \mid s?(t);\mathbf{0}$ もまたエラーである．受信者が通信路を入力することを期待しているが，送信者は値を出力しているためである.

$$\textsc{Com}: \frac{e \downarrow v}{s!\langle e\rangle;P \mid s?(x);Q \longrightarrow P \mid Q\{v/x\}}$$

$$\textsc{Label}_1: \frac{}{s \blacktriangleleft \mathsf{left};P \mid s \blacktriangleright \{\mathsf{left}\colon Q_1,\ \mathsf{right}\colon Q_2\} \longrightarrow P \mid Q_1}$$

$$\textsc{Label}_2: \frac{}{s \blacktriangleleft \mathsf{right};P \mid s \blacktriangleright \{\mathsf{left}\colon Q_1,\ \mathsf{right}\colon Q_2\} \longrightarrow P \mid Q_2}$$

図 2.6: ラベル分岐付き π 計算で追加された構文の簡約意味

2. 通信路における競合，すなわち，$s?(x);\mathbf{0} \mid s?(y);\mathbf{0}$, $s \blacktriangleleft \mathsf{left};\mathbf{0} \mid s \blacktriangleleft \mathsf{left};\mathbf{0}$ and $s!\langle t\rangle;\mathbf{0} \mid s!\langle t'\rangle;\mathbf{0}$ 等はエラーである．

エラーは s-プロセスと s-基の概念を用いて定式化される．

定義 2.12 (s-プロセスと s-基) s-プロセスはサブジェクトが s である閉じたプロセスである．s-基は 2 つの s-プロセス P, Q の並行合成 $P|Q$ である．つまり，$s!\langle e\rangle;P_1 \mid s?(x);P_2$, $s \blacktriangleleft \mathsf{left};P \mid s \blacktriangleright \{\mathsf{left}\colon P_1,\ \mathsf{right}\colon P_2\}$, $s \blacktriangleleft \mathsf{right};P \mid s \blacktriangleright \{\mathsf{left}\colon P_1,\ \mathsf{right}\colon P_2\}$，$s!\langle t\rangle;P \mid s?(t');P$ は s-基である． □

直観的には，s-基とは通信路 s でメッセージ種別に不整合なく通信可能なプロセスの対である．

定義 2.13 (エラー) プロセス P は，ある s-プロセス Q_1 と Q_2 が存在し $P \equiv (\nu \tilde{s})((Q_1|Q_2)|R)$ かつ $Q_1|Q_2$ が s-基でないとき，かつそのときのみ，エラーである． □

デッドロックはエラーとして扱わないことに注意すること．

定義 2.14 (セッション型) セッション型は次の文法で定義される．

$$\begin{array}{rl} T, T', T_1, T_2 & ::= !S;T \mid ?S;T \mid \oplus\{\mathsf{left}\colon T_1,\ \mathsf{right}\colon T_2\} \\ & \mid \&\{\mathsf{left}\colon T_1,\ \mathsf{right}\colon T_2\} \\ & \mid ![T'];T \mid ?[T'];T \mid \mathsf{end} \mid \bot \end{array} \tag{2.6}$$

ここで S は $\{\mathsf{bool}\}$ を含むソート（基本型）の上を動く． □

それぞれの型の直観的な意味は次の通りである．

- $!S;T$ はソート S の値を出力し，次に T として振舞う．

- $?S;T$ はソート S の値を入力し，次に T として振舞う．
- $\oplus\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}$ は，ラベル left か right のどちらか一方を出力し，出力したラベルに応じて T_1 か T_2 のどちらかとして振舞う．
- $\&\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}$ は，ラベルを入力し，入力したラベルに従い T_1 か T_2 のどちらかとして振舞う．
- $![T'];T$ は 型 T' をもつ通信路を出力し，次に T として振舞う．
- $?[T'];T$ は 型 T' をもつ通信路を入力し，次に T として振舞う．
- end は停止したセッションを表す．型 end をもつ通信路ではそれ以降の通信が発生しない．
- $\bot$ は，その通信路はすでに２つの並行プロセスに占有されており，それ以外のプロセスがその通信路を使えないことを表す．

型 $!S;T$, $?S;T$, $\oplus\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}$, $\&\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}$, $![T_1];T_2$, $?[T_1];T_2$ はプレフィクスされた型と呼ぶ．end と $\bot$ はプレフィクスされた型ではない．

セッション型の**双対**は，その通信路の端点とは別の，もう片方の端点の使われ方を示す．

定義 2.15 (双対) セッション型 T の双対 $\overline{T}$ は次の等式で定義される．

$$\frac{\overline{!S;T} = ?S;\overline{T} \qquad \overline{![T_1];T_2} = ?[T_1];\overline{T_2}}{\overline{\oplus\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}} = \&\{\mathsf{left}\colon \overline{T_1}, \mathsf{right}\colon \overline{T_2}\}} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$\overline{\bot}$ は定義されない． □

ソーティング $\Gamma, \Gamma', \cdots$ は変数からソートへの有限な写像である．同様に，型環境[2] $\Delta, \Delta', \cdots$ は通信路からセッション型への有限な写像である．それぞれ次の文法で定義される．

$$\Gamma \ ::= \ \emptyset \mid \Gamma \cdot x\colon S \qquad\qquad \Delta \ ::= \ \emptyset \mid \Delta \cdot s\colon T$$

慣習により，型環境 $\Delta \cdot s\colon T$ において Δ は s を含まないものとする．ソーティングにも同様の慣習を適用する．$\Delta(s)$ で Δ における s の型を表す．Δ に関する操作

[2] [YV07] では型付け（typing）と呼ばれるが，他の型システムと用語を統一するため，本論文では型環境と呼ぶこととした．

を次の通り定義する：$\mathrm{dom}(\Delta) = \{s \mid \exists T.\Delta(s) = T\}$, $\mathrm{cod}(\Delta) = \{T \mid \exists s.\Delta(s) = T\}$, $(\Delta \cdot s\colon T)\backslash s = \Delta$. 型環境 Δ は，$\forall s \in \mathrm{dom}(\Delta), \Delta(s) = \mathsf{end}$ のとき完了している（completed）と呼ぶ.

定義 2.16 (型環境の適合性と合成) すべての $s \in dom(\Delta_0) \cap dom(\Delta_1)$ について $\Delta_0(s) = \overline{\Delta_1(s)}$ が成立するとき，型環境 Δ_0 と Δ_1 の間に適合性があるといい，$\Delta_0 \asymp \Delta_1$ とかく．Δ_0 と Δ_1 の間に適合性があるとき，合成 $\Delta_0 \circ \Delta_1$ は次の通り定義される：

$$(\Delta_0 \circ \Delta_1)(s) = \begin{array}{ll} \bot & \textit{if } s \in dom(\Delta_0) \cap dom(\Delta_1) \\ \Delta_0(s) & \textit{if } s \in dom(\Delta_0) \backslash dom(\Delta_1) \\ \Delta_1(s) & \textit{if } s \in dom(\Delta_1) \backslash dom(\Delta_0) \end{array} \tag{2.7}$$

□

定義 2.17 (型判定) 型判定 $\Gamma \vdash P \triangleright \Delta$ は図 *2.7* の規則により帰納的に定義される 3 項関係である．これ以後，*3* 項関係のソート判定 $\Gamma \vdash e \triangleright S$ を仮定する．さらに，すべての $\Gamma, x\ and\ S$ について $\Gamma \cdot x\colon S \vdash x\colon S$, $\Gamma \vdash \mathsf{true} \triangleright \mathsf{bool}$, $\Gamma \vdash \mathsf{false} \triangleright \mathsf{bool}$ を仮定する．プロセス P は，$\Gamma \vdash P \triangleright \Delta$ なる Γ, Δ が存在するとき型付け可能であるという. 型判定 $\emptyset \vdash P \triangleright \Delta$ は $\vdash P \triangleright \Delta$ と略記される. $\vdash P \triangleright \Delta$ ならば $\nvdash P \triangleright \Delta$ と書く. $\nvdash P$ ならば，P は型付けできないという. □

規則 [Send], [Rcv], [Sel1], [Sel2], [Br],[Thr] , [Cat] は，それぞれのチャネルの使用をそのまま型に記録する．[Thr] については，送られた通信路 t が受信先で使用されることを，送信元の型環境に記録している．[Conc] は型環境を $\circ$ で分離する．これは P と Q の型環境の並行合成とみなせる. [Inact] は，停止したプロセス $\mathbf{0}$ においてはすべての通信路の使用が完了（end）していなければならないことを示している．[CRes] は，名前制限で抽象化されるすべての通信路について，互いに双対なセッション型をもつ 2 つのプロセスに使用されなければならないことを示している．[Bot] は主部合同 (subject congruence) を成立させるために [YV07] で導入された．詳細は [YV07] の 2.3 節を参照.

本田らのオリジナルのセッション型システム [HVK98] では，操作的意味定義の Pass 規則が π 計算の規則と異なり，次の規則 Pass2 に置き換わっている：

$$\textsc{Pass2}: \quad \overline{s!\langle t\rangle;P \mid s?(t);Q \longrightarrow P \mid Q} \tag{2.8}$$

この微妙に異なる操作的意味のもとで，セッション型システムの型安全性が示されている.

$$[\textsc{Send}]\ \frac{\Gamma \vdash e \triangleright S \qquad \Gamma \vdash P \triangleright \Delta \cdot s\colon T}{\Gamma \vdash s!\langle e\rangle;P \triangleright \Delta \cdot s\colon !S;T} \qquad [\textsc{Rcv}]\ \frac{\Gamma, x\colon S \vdash P \triangleright \Delta \cdot s\colon T}{\Gamma \vdash s?(x);P \triangleright \Delta \cdot s\colon ?S;T}$$

$$[\textsc{Sel1}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon T_1}{\Gamma \vdash s \blacktriangleleft \mathsf{left};P \triangleright \Delta \cdot s\colon \oplus\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}}$$

$$[\textsc{Sel2}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon T_2}{\Gamma \vdash s \blacktriangleleft \mathsf{right};P \triangleright \Delta \cdot s\colon \oplus\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}}$$

$$[\textsc{Br}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon T_1 \qquad \Gamma \vdash Q \triangleright \Delta \cdot s\colon T_2}{\Gamma \vdash s \blacktriangleright \{\mathsf{left}\colon P, \mathsf{right}\colon Q\} \triangleright \Delta \cdot s\colon \&\{\mathsf{left}\colon T_1, \mathsf{right}\colon T_2\}}$$

$$[\textsc{Thr}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon T}{\Gamma \vdash s!\langle t\rangle;P \triangleright \Delta \cdot s\colon ![T'];T \cdot t\colon T'} \qquad [\textsc{Cat}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon T \cdot t\colon T'}{\Gamma \vdash s?(t);P \triangleright \Delta \cdot s\colon ?[T'];T}$$

$$[\textsc{Conc}]\ \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta' \qquad P \asymp Q}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'}$$

$$[\textsc{Bot}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon \mathsf{end}}{\Gamma \vdash P \triangleright \Delta \cdot s\colon \bot} \qquad [\textsc{Inact}]\ \frac{\Delta\ \text{completed}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad [\textsc{CRes}]\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s\colon \bot}{\Gamma \vdash (\nu s)P \triangleright \Delta}$$

図 2.7: セッション型システムの型付け規則

事実 2.2 (規則 Pass2 を採用した場合の型安全性（[YV07] の定理 2.11）) 図*2.2*の簡約規則のうち，Pass を Pass2 に置き換えた簡約規則では，型付けできるプロセスはエラーに簡約されない． □

[YV07] で指摘されたように，Pass2 は π 計算の意味論としてはあまり適していない．もし $s!\langle t\rangle;P \mid s?(u);Q$ というプロセスにおいて $t \in fn(Q)$ であるとき，α 変換が適用できず簡約がそれ以上できなくなるためである [YV07]. 規則 Pass を採用した場合には主部簡約性が成立しないことが知られている [YV07]．この問題は 5 章で扱う主要なテーマである．

2.5 まとめ

本章で，π 計算の動作意味と，非同期局所化 π 計算（ALπ）とセッション型システムを導入した．ALπ は，π 計算に非同期性と局所性の制限を加えた体系である．非同期性は，出力プレフィクスの継続において通信動作を禁止することで表される．局所性は，名前の入力能力の移動を禁止する制限であり，名前と計算資源の位置をよく対応づける．このような性質から， ALπ は分散システムの具体的な表現として用いられてきた．3 章では，ALπ を用いたネットワークプログラミングの枠組みとその実装手法を議論する．

セッション型システムは，単一の通信路を介した異種のメッセージの送受に関する整合性を検査する型システムである．セッション型は通信路の端点に割り当てられる．他方の端点との通信の整合性は双対性により定式化される．セッション型は通信路に割り当てられた通信手順の表現であり，型検査によって通信記述の正しさを検証できる．4 章では，セッション型を Haskell に導入するプログラミング技法を議論する．

本田らのセッション型システム [HVK98] には，π 計算の通常の簡約規則では主部簡約が成立しない問題が知られている．主部簡約は型安全性の確立のために重要な性質である．5 章では，[HVK98] の型付け規則を拡張した体系において主部簡約を定式化し，これを証明する．

第3章　Haskellにおける非同期局所化π計算に基づくネットワークプログラミング

3.1　はじめに

ネットワークアプリケーションの普及により，動的な通信リンクの変化を伴うネットワークプログラムはより一般的になりつつある．このようなプログラムにおいては，動的なネットワーク構造をもつ柔軟なソフトウェアの構成が可能である反面，ソースプログラムの記述だけから振る舞いを十分に把握することは難しい．

本章ではこうした動的なネットワーク構造の変化を記述できる体系であるπ計算 [Mil99] に基づいて Haskell[Has10] に対する埋め込み言語を定義することによって，ネットワークプログラムの振る舞いを直接記述できるフレームワークを提案する．π計算では，名前の送受や生成を行う並行プロセスを柔軟に記述でき，型理論やプロセスの等価性に関する多くの解析技法が開発されている．

我々は Haskell のための型付き非同期局所化π計算 (typed Asynchronous Localized π-calculus; 以下，ALπ と略記する) に基づくネットワークプログラミングフレームワーク PiMonad を実装する．ここで ALπ の型システムと構文を Haskell に埋め込み，PiMonad プログラムのネットワーク機能の動作を Haskell で実装する．

本フレームワークの特徴は，Haskell の型システムにより，ALπ の通信路が型付けされ，内部動作と通信の両面において実行時エラーが発生せず信頼性が高いことである．さらに，フレームワークの実装において ALπ の動作意味を直接的に模倣しており，π計算における既存の技法を適用して振る舞いを解析できる．さらに，Haskell の抽象化能力の高さを利用して，非常に小さなコア部分から様々な通信動作を表現したコンビネータを構成できる．

ALπ は，π計算を次の２つの点で限定した体系である．(1) 出力プレフィックスの後に **0** 以外のプロセスを認めない (非同期性)，(2) 通信路の入力能力を他のプロ

セスに移譲しない (局所性)．非同期性は，計算能力を限定することなく，実装を容易にする [PT00]．局所性によって通信路の入力能力の移動が制限され，通信が発生するホストを構文的に特定することができる．このような特徴は、IP アドレスやノード番号に基づいて通信を行う多くのネットワークプログラムの概念によく当てはまり，比較的低レベルなネットワークプログラミングのモデルとして適当である．

PiMonad において，ALπ の構文は Haskell のモナドとして表現される．Haskell で入出力に用いられる `IO` モナドと同様に，通信などの副作用を伴う部分を，PiMonad として分離して記述するのが特徴である．純粋関数的部分は Haskell の処理系および標準ライブラリを利用することができる利点がある．さらに，埋め込み言語によって実装することで，Haskell 処理系が持つ厳密さに基づいて言語処理系を構築することができる．

本章では，PiMonad の構成，Haskell の I/O システムへの適用，さらにネットワーク上での実装を示す．まず I/O システムとの相互作用を伴わない体系，π 計算の記述のための関数および評価器について定義する．型付き言語である ALπ を Haskell に埋め込む際の問題の一つは，Haskell の型システムにはサブタイプ機構がないことである．

本章では，サブタイプ関係を多引数型クラス [DO02] を用いて表現する．次に，I/O システムとの相互作用を伴う体系への拡張を示す．`IO` モナドを出力能力をもつ通信路として表し，この実装により一般的な入出力の記述が可能であることを示す．さらに，ネットワーク入出力機能を表す `IO` モナドを提供し，通信路の表現に通信ホストの位置情報を組み込むことで，ネットワークを介した名前通信へ拡張する．このように拡張を行うことで実際的なネットワークプログラムが抽象的なモデルに近い形で直接的に記述できる．

ALπ の利点は，ネットワーク実装において示される．局所性を仮定しない場合，通信がネットワーク上のどのホストで発生するのか特定できず，実装が複雑になる．本フレームワークは文法を ALπ に限定し，記述の対象から複雑な挙動を除外する．このため，処理系としては軽量な実装が可能になっている．

本章の構成は以下のとおりである．3.2 節では非同期 π 計算に対する PiMonad を構成する方法を示す．3.3 節では π 計算の i/o 型とサブタイプ関係による局所性を型クラスで実現する方法について述べる．3.4 節では Haskell の入出力システムとの相互作用のために拡張する．3.5 節でネットワーク実装について述べ，3.6 節でプログラム例を与える．3.7 節でまとめと今後の課題について述べる．

3.2 非同期 π 計算の Haskell における表現

本節で，サブタイプ関係がない非同期 π 計算 (Aπ) の Haskell での表現を示す．ここで問題となるのが，ν 演算子に対応する一意な名前の生成方法である．

ここでは複数のモナドの機能の組み合わせにより，一意な名前を生成する機構をもつ PiMonad を与え，Aπ を Haskell で表現する．モナドの構造を把握するため，まず，ν 演算子なしの非同期 π 計算 ($A\pi^{-\nu}$) のプロセスの Haskell での表現を与え，$A\pi^{-\nu}$ の文脈 (context) を表す継続渡しモナド `Ctx` を与える．次に，名前生成器の状態をモナドに保持させるため，ν 演算子を含んだ Aπ の文脈を，`Ctx` に状態計算を導入したモナドである PiMonad を与える．

3.2.1 $A\pi^{-\nu}$ の Haskell での表現

通信路の型定義 型 a を送受信する通信リンク ♯a を表す型を，`Ch a` として表す．例えば，π 計算において型 ♯String で表される通信路を `Ch String`，型 ♯♯() を `Ch (Ch ())` 等と表す．一方，名前には型の情報は与えず，すべて整数で表す．通信路型 `Ch a` の定義を以下のように与える．

```
data Ch a = Name Int
```

型 `Ch a` は，定義の左辺に現れる型変数が右辺には現れない幽霊型であり，型情報を忘却することで，名前はすべて `Int` 型の整数表現を持つ一方，送受信のためには違った型を持つ．

プロセス $A\pi^{-\nu}$ のプロセスの Haskell における表現を表 3.1 にまとめる．Haskell で，出力プロセス $s!\langle t\rangle$ を `Send s t`，並行合成プロセス $P|Q$ を `Par P Q`，終了したプロセス **0** を `Zero` で表す．入力プロセス $s?(t);P$ による名前束縛は λ 束縛で表現し，`Recv s (\t->P)` で表す．複製 $*s?(t);P$ も同様に `ReprRecv s (\t->P)` と表す．

各データ構築子の型を，`Send :: forall t. Ch t -> t -> P` や `Recv :: forall t. Ch t` 等と決める．これにより，型 ♯t の通信路で送受信可能な値の型が t であることを，Haskell の型システムが保証する．これは，Haskell を存在型で拡張した処理系で扱える．

しかしながら，存在型はそれぞれの値の出現で違った型として扱われるため，τ 遷移の計算に必要な `Send c1 v` と `Recv c2 f` から `f v` を計算するなどの処理は型付けできずコンパイル時エラーとなる．

表 3.1: $A\pi^{-\nu}$ と Haskell の項の対応関係

$A\pi^{-\nu}$	Haskell
$s?(t);P$	`Recv` s `(\`t`->`P`)`
$s!\langle t\rangle$	`Send` s t
$*s?(t);P$	`ReprRecv` s `(\`t`->`P`)`
$P \mid Q$	`Par` P Q
$\mathbf{0}$	`Zero`

例：$s?(t);(s!\langle u\rangle \mid *s?(v);v!\langle w\rangle)$	`Recv` s `(\`t`->`
	`Par (Send` s u`)`
	`(ReprRecv s (\`v`->`
	`Send` v w`)))`

ここで，`Send :: forall t. Typeable t => Ch t -> t -> P` のように型 `t` が `Typeable` クラスに属すれば，関数 `cast` による型キャストを行うことができる[1]．これにより，プロセスの τ 遷移を計算する関数を以下のように定義する．

```
tau :: P -> P -> Maybe P
tau (Par (Send (Name i) v)
         (Recv (Name j) f)) =
  if i==j then Just (Par Zero (f (cast v)))
          else Nothing
```

これらのデータ構築子をまとめ，通信路型と併せて，$A\pi^{-\nu}$ を表す型の定義を図 3.1 に示す．通信路自身も送受信可能とするため，データ型 `Ch a` も `Typeable` のインスタンスとして宣言する[2]．

3.2.2　継続モナドによる $A\pi^{-\nu}$ の文脈表現

モナドによるプロセスの表現は，後の名前生成を導入した体系において不可欠な役割を果たす．ここでは，まず型 `P` を結果の型とする継続モナド [Wad92]`Ctx` を次の様に定義する[3]：

```
newtype Ctx a = Ctx {runCtx :: ((a -> P) -> P)}
```

[1]強制的に型を変換する `unsafeCoerce` を使ってもよい．
[2]キャストに unsafeCoerce を用いる場合，これは必要ない
[3]Haskell の標準ライブラリにある `Cont` の answer type を `P` としたモナドを用いてもよい．

プロセス

`data P =`	
`  Par P P`	並行合成 $P\|Q$
`| Zero`	終了 **0**
`| forall t.(Typeable t) => Recv (Ch t) (t -> P)`	入力 $s?(t);P$
`| forall t.(Typeable t) => Send (Ch t) t`	非同期出力 $s!\langle t\rangle;\mathbf{0}$
`| forall t.(Typeable t) => ReprRecv (Ch t) (t -> P)`	複製入力 $*s?(t);P$

図 3.1: A$\pi^{-\nu}$ の Haskell での表現

```
instance Monad Ctx where
  return x = Ctx (\k -> k x)
  Ctx c >>= f =
       Ctx (\k -> c (\a -> runCtx (f a) k))
```

型 `Ctx a` を持つモナドは型 `a` を受け取る継続を要求する文脈と見なすことができる．>>=演算子はある文脈と同じ型の自由変数を持つ別の文脈から新たな文脈を合成する演算と見なせる．

文脈の穴にプロセス**0**を代入しプロセスを得る関数 `ctx2Pr` を以下に定義する：

```
ctx2pr :: Ctx () -> P
ctx2pr (Ctx f) = f (\_ -> Zero)
```

さらに，モナド `Ctx` の組み合わせにより A$\pi^{-\nu}$ の基本的な演算子を表現するため，A$\pi^{-\nu}$ の文脈に対応する関数を与える．

入力　π 計算の入力ガード式 $s?(t);P$ がもつ文脈は，$s?(t);[\cdot]$ である．対応する関数 `recv` を以下に定義する：

```
recv :: (Typeable a) => Ch a -> Ctx a
recv ch = Ctx (\k -> Recv ch k)
```

引数は入力に用いられる通信路を表す．定義における継続 `k` は型 `a -> P` を持ち，入力動作に継続するプロセスを表す．

非同期出力　非同期出力プロセス $s!\langle t\rangle;\mathbf{0}$ には継続するプロセスがないため，そのままでは文脈を構成しない．その代わりに，文脈 $(s!\langle t\rangle;\mathbf{0}|[\cdot])$ を与える．このようにすると，継続が非同期出力の完了に依存しないが，非同期処理を伴うプログラミングにおいてそのような状況は自然である．この文脈に対応する関数 send を以下に定義する：

```
send :: (Typeable a) => Ch a -> a -> Ctx ()
send ch v = Ctx (\k ->
                  (Par (Send ch v) (k ())))
```

第一引数は出力に用いられる通信路を，第二引数は出力される値を表す．定義における継続 k は型 () -> P を持ち，並行合成演算の右手側に置かれる．

並行合成　並行合成プロセス $(P|P')$ より文脈 $(P|[\cdot])$ を得る．対応する関数 fork を以下に定義する：

```
fork :: Ctx () -> Ctx ()
fork c =
  let p = ctx2pr c
  in Ctx (\k  -> Par p (k ()))
```

第一引数は別の文脈で，関数 ctx2Pr によりプロセスに変換された後，並行合成演算の左手側に置かれる．

複製　複製プロセス $!x(y).P$ からは，入力プロセスの場合と同様のやり方では文脈 $!x(y).[\cdot]$ が作れる．しかしながら，入力プロセスの場合と違い，継続するプロセスが無限に複製されるため直観にあわない．そこで非同期出力プロセスの場合と同様に，文脈 $(!x(y).P|[\cdot])$ から対応する関数 rep を以下に定義する：

```
rep :: (Typeable a) =>
       Ch a -> (a -> Ctx ()) -> Ctx ()
rep ch cont =
  let k'  = \a -> ctx2pr (cont a)
  in Ctx (\k ->
           (Par (ReprRecv ch k') (k ())))
```

第一引数は入力に用いられる通信路を，第二引数は入力動作に継続する文脈を表す．文脈は fork の場合と同様に ctx2pr によりプロセスに変換され並行合成される．

3.2.3 Aπ の Haskell における表現

名前制限演算子を含む Aπ のプロセスの Haskell での表現を与える．以下で，この直観的意味を説明する．

名前生成器 名前は型 `Int` の整数値で表現する．ν 演算子で新しい名前を生成するための源として，`Int` の十分に大きいリストを与える．これを名前生成器と呼ぶ．名前生成器の型を `Gen` と名付ける．新しい名前を表す整数はリストの `head` を用い，生成器の次状態は `tail` を用いる．生成器の初期状態を `initial` と名付ける．

プロセスの型 名前制限されたプロセスの Haskell での表現を型 `APi` で与える．直観的に，そのスコープでの名前制限の (空かもしれない) リストと，名前制限の下のプロセスの対である．

名前制限 1 つの名前制限 $(\nu s : L)$ を，`NewChan` s として表す．

名前制限演算子の下のプロセス 名前制限演算子の下のプロセスを表す型を `APiSub` とする．データは型 `P` にほぼ対応するが，入力プレフィクスに関わるデータ構築子 `Recv`，`ReprRecv` は，入力動作後に継続が適用されプレフィクスの下のプロセスが活性化されるため，引数に生成器を追加する必要がある．さらに，適用した生成器の次の状態を得る必要から，継続の戻り値にも生成器を追加する．よって，Recv および ReprRecv を次のように定義する：

```
...
| Recv (Ch a) (a -> Gen -> (APi, Gen))
| ReprRecv (Ch a) (a -> Gen -> (APi, Gen))
...
```

継続するプロセスがない `Send`, `Par`, `Zero` については `P` のデータ構築子をそのまま用いる．

例 3.1 $(\nu t : \sharp a)s!\langle t\rangle$ を `([NewChan` t`], Send` s t`)` と表す．継続するプロセスで名前制限があるプロセス $s?(t);(\nu u : \sharp a)t!\langle u\rangle$ は，

```
Recv s (\t -> gen ->
  let u = (Name (head gen)) :: Ch a
  in (([NewChan u], Send t u), tail gen)
```

表 3.2: Aπ と Haskell の項の対応関係

Aπ	Haskell
$(\nu u : L)s?(t);P$	`([NewChan` u`], Recv` s `(\`t`->\gen->(([],`P`),gen)))`
$(\nu t : L)s!\langle t\rangle$	`([NewChan` t`], Send` s t`)`
$*s?(t);(\nu u : L)P$	`([], ReprRecv` s `(\`t`->\gen->(([NewChan` u`],`P`),gen)))`

と表す．その他のプロセスの対応関係の例を表 *3.2* に挙げる．　□

3.2.4　状態モナドを用いた名前生成機能の追加

ここで，Aπ の文脈に対応する一意な名前生成が可能なモナドを，3.2.2 節で与えた $A\pi^{-\nu}$ の文脈に対応するモナドに，状態を伴う計算を表すモナドの機能を組み合わせ構成する．

モナドで生成器の状態を保持するため，3.2.2 節で導入した継続モナド `Ctx a` に，状態モナドを合成する．継続モナドと状態モナドの合成 `CS` を以下に示す[4]：

```
data ContState a =
  CS ((a-> S -> (R, S)) -> S -> (R, S))
instance Monad ContState where
  return a = CS (\k -> \s -> k a s)
  (CS f) >>= g =
    CS (\k -> \s ->
         f (\a -> \s' ->
             let (CS g') = g a in g' k s')
           s)
```

ここで状態を表す型を `S`，継続の結果の型を `R` としている．直観的には，このモナドに型 `(a -> S -> (R, S))` を持つ継続と，型 `S` を持つ初期状態を適用すると，型 `(R, S)` で表される継続の結果の型と最終状態の対が得られる．

[4]Haskell プログラミングにおいてはモナド変換子 `StateT` を使うのが普通であるが，型の構造を示すためこのようにした．

このモナドの状態の型を Gen，結果の型を APi として，Aπ の文脈を表すモナドを図 3.3 に示す．これを PiMonad と呼ぶことにする．

本節の目的としていた名前制限の文脈 $(\nu s : L)[\cdot]$ に対応する関数 new および補助関数 addNew を以下に定義する：

```
new :: (Typeable a) => PiMonad (Ch a)
new = PiMonad (\k -> \gen ->
    let newname = Ch (head gen)
        gen'    = tail gen
        (process, gen'') = k newname gen'
    in (addNew newname process, gen'')
  )
  where addNew n (ns ,p) = (NewChan n:ns, p)
```

新しい名前は生成器の head から取り，生成器の次状態を tail としている．継続に新しい名前と生成器を適用し，プロセスを得，名前制限を関数 addNew によりプロセスに記録する．

3.2.2 節で定義した，A$\pi^{-\nu}$ の基本的な演算子に対応する send，recv，rep，fork も，Aπ を表す PiMonad の形に変更する必要がある．ここでは send，recv，fork のみ示す：

```
send :: (Typeable a) =>
        Ch a -> a -> PiMonad ()
send ch v = PiMonad (\k -> \gen ->
    let (p, gen') = k () gen
    in (([], Par (Send ch v) p), gen')
  )
recv :: (Typeable a) => Ch a -> PiMonad a
recv ch = PiMonad (\k -> \gen ->
                  (([], Recv ch k), gen))
fork :: PiMonad () -> PiMonad ()
fork (PiMonad f) = PiMonad (\k -> \gen ->
    let (p, gen')  = f (\_->Zero) gen
        (q, gen'') = k () gen'
    in (([], Par p q), gen'')
```

)

send において，継続 k に生成器 gen を適用し並行合成するプロセス p を得る．一方，recv は生成器を消費しない．入力動作が行われる時に生成器が適用され，名前生成が行われる．rep も同様に定義される．

3.2.5　抽象評価器による PiMonad の実行

PiMonad () 型の値として定義したプロセスはそのままでは実行されることはない．Aπ の項と同様な実行意味を与えるのは，Haskell で定義した関数 eval :: APi -> [Trans] である．Trans は Aπ における遷移ラベルと遷移後のプロセスの対 (α, P) に相当し，eval は図 2.4 で示した π 計算の遷移規則の通りに，遷移ラベルと遷移先のプロセスを返すよう定義されているものとする．

遷移を表す型 Trans の定義　Trans を，次の通り定義する：

```
data Trans =
      TauAct APi
  | forall x. (Typeable x) =>
      SendAct [NewChan] (Ch x) x APi
  | forall x. (Typeable x) =>
      RecvAct (Ch x) (x -> APi)
```

データ構築子 TauAct は，規則 Comm や Close-{L,R} に起因する τ 遷移に相当する．引数は，遷移後のプロセスに相当する．

SendAct は，規則 AOut に起因する出力遷移に相当する．第二引数と第三引数は，それぞれ遷移ラベルの通信路と送られる値に対応する．第四引数は，遷移後のプロセスに対応する．第一引数 ([NewChan]) は，送られる名前が通信路型の場合に束縛された名前のリストに対応する．第一引数が空でない時は，規則 Open に対応する実行と見なせる．

RecvAct は，規則 Inp または RepInp に起因する入力遷移に相当する．第一引数は遷移ラベルのサブジェクトに相当する．第二引数は，入力値に適用すると遷移後のプロセスを返す関数を表す．

フレッシュな名前の生成器

```
type Gen  = [Integer]
initial  = [1..]
```
整数の無限リスト

名前制限のないプロセス

```
data APiSub  =
  Par APi APi
 |Zero
 |forall x.(Typeable x) =>
    Recv (Ch x) (x -> Gen -> (APi, Gen))
 |forall x.(Typeable x) =>
    Send (Ch x) x
 |forall x.(Typeable x) =>
    ReprRecv (Ch x) (x -> Gen -> (APi, Gen))
```
並行合成 $P|Q$
終了 $\mathbf{0}$
入力 $s?(t);P$
非同期出力 $s!\langle t\rangle;\mathbf{0}$
複製入力 $*s?(t);P$

プロセス

```
type APi = ([NewChan], APiSub)
```
名前制限されたプロセス

名前制限

```
data NewChan = forall x.  NewChan (Ch x)
```
名前のプレースホルダ

図 3.2: Haskell における Aπ の表現

PiMonad の実行　PiMonad の実行は，プロセスを関数 eval の適用結果から一つのプロセスを選び，再度，eval を適用することを繰り返して得られる系列となる．図 2.4 で定義した遷移規則は Early 意味論であるため，入力遷移 RecvAct の遷移先のプロセスは入力ラベルで束縛された名前で抽象化されている．

3.3 i/o型のエンコーディングによる 非同期局所化π 計 算の表現

本節で，ALπ の 3 つの通信路型の構築子 $\sharp$，i，o に対応する型を導入し，さらに値型の概念を型クラスで定める．次に，図 2.5 で紹介した ALπ のサブタイプ関係を Haskell の多引数型クラス [SPJM97] により表現する方法を示す．前節で導入し

```
newtype PiMonad a =
  PiMonad {runPi::(a -> Gen -> (APi,Gen))
                  -> Gen
                  -> (APi,Gen)}

instance Monad PiMonad where
  return x = PiMonad (\k -> \g -> k x g)
  m >>= f =
    PiMonad (\k -> g ->
            runPi m (\a-> \g' ->
                       runPi (f a) k g') g
```

図 3.3: PiMonad : Aπ の文脈の Haskell での表現

た send,recv, rep, new をこの型クラスを用いた形で拡張し，Haskell への ALπ の埋め込みを達成する．

3.3.1　i/o 型および値型の導入

ALπ の 3 つの通信路型と Haskell の関係をそれぞれ，入力能力 ia を I a，出力能力 oa を O a，そして入出力能力 ♯a を L a として与える．

ALπ では基本型および出力能力のみが値型である．値型に対応する型クラス ValueType を定義し，出力能力と，値として送受信可能としたい Haskell の基本型を ValueType のインスタンスとして宣言する．cast 関数のため (3.2.1 節)，ValueType は Typeable のサブクラスとして定義する．

上述 3 つの通信路型および値型を表す型クラスを図 3.4 に与える．これを用い，サブタイプ関係を用いない ALπ を図 3.5 に与える．ここで，Aπ に対応する図 3.2 の 型 APiSub のデータ構築子 Send (Ch x) x，Recv (Ch x) ⋯, ReprRecv (Ch x) ⋯ をそれぞれ i/o 型のために特化した 型 ALPiSub の Send (O x) x，Recv (I x) ⋯，ReprRecv (I x) ⋯ とし， さらに 型 x の動く範囲を ValueType のインスタンスに制限している．名前制限のために用いられる NewChan は入出力能力 L に特化する．

Channel for both input and output capability

```
data L a = L Integer
```

Input capable channel

```
data I a = I Integer
```

Output capable channel

```
data O a = O Integer
```

Type class for value types

```
class (Typeable a) => ValueType a where
  -- no functions

instance ValueType Int where
instance ValueType Float where
...

instance (ValueType a) => ValueType (O a) where
```

図 3.4: i/o チャネル型と値型の Haskell での表現

Unrestricted process

```
data ALPiSub =
  Par ALPi ALPi                                  parallel composition P|Q
 |Zero                                           inactive process 0
 |forall x.(ValueType x) =>
   Recv (I x) (x -> Gen -> (ALPi, Gen))          input x(y).P
 |forall x.(ValueType x) =>
   Send (O x) x                                  asynchronous output x̄y.0
 |forall x.(ValueType x) =>
   ReprRecv (I x) (x -> Gen -> (ALPi, Gen))      replicated input !x(y).P
```

Process type

```
type ALPi = ([NewChan], ALPiSub)                 process with restriction
```

Restricted name

```
data NewChan = forall x.  NewChan (L x)          placeholder of the name
```

図 3.5: ALπ の Haskell での表現

3.3.2　サブタイプ関係の型クラスによるエンコーディング

サブタイプ関係を型付けに関連づける SubSumption 規則を，coercion の役割を果たすメソッドを持つ多引数型クラスで表現する．

出力チャネル型　出力チャネルには，型 ♯V または oV を用いる．♯V は oV に coerce する．これを次の多引数型クラス OChan で表す：

```
class (ValueType a) => OChan o a where
  coerceO :: o a -> O a
instance (ValueType a) => OChan O a where
  coerceO = id
instance (ValueType a) => OChan L a where
  coerceO (L i) = O i
```

出力対象の型はクラス宣言の文脈で値型に制限する．メソッド coerceO は最外の型構築子を O に変換する．インスタンスは O および L について宣言され，それぞ

れの coerce0 の実装は，恒等関数と，名前表現を保持し型のみを L から 0 に変換する関数で与えられる．

入力チャネル型 ALπ では，入力チャネルには，型 $\sharp V$ または iV を用いる．出力チャネルと同様に，次の通り型クラス IChan を宣言すれば十分である：

```
class (ValueType a) => IChan i a where
  coerceI :: i a -> I a
instance (ValueType a) => IChan I a where
  coerceI = id
instance (ValueType a) => IChan L a where
  coerceI (L i) = I i
```

値型 値型についてはサブタイプ関係を直接多引数型クラスと coercion のための関数を与える．次の型クラス Subtype によりこれを表現する：

```
class (ValueType c2) =>
      Subtype c1 c2 where
  coerce :: c1 -> c2
instance (ValueType v) =>
         Subtype (L v) (O v) where
  coerce (L i) = O i
instance (ValueType v) =>
         Subtype (O v) (O v) where
  coerce = id
instance Subtype Int Int where
  coerce = id
instance Subtype Float Float where
  coerce = id
(and many declarations for other base types)
```

スーパータイプが値型であることを要請するため，クラス宣言の文脈で型の第二引数を ValueType に制限する．図 2.5 に現れるサブタイプ関係のうち，Sub-$\sharp$O に対応する coercion 関数のみ，型のみを L から 0 に変換する関数で与える．残りは全て Sub-Refl に対応するため，恒等関数で与える．

```
send :: (OChan c a, Subtype b a)
        => c a -> b -> PiMonad ()
send ch v = PiMonad (\cont -> \gen ->
    let (cont',gen') = cont () gen
      ch' = coerce0 ch
      v' = coerce v
      p = ([], Send ch' v')
    in (([], Par p cont'), gen')
  )

recv :: (IChan i a) => i a -> PiMonad a
recv ch = PiMonad (\f -> \gen ->
      let ch' = coerceI ch
      in (([], Recv ch' f), gen)
  )
```

図 3.6: サブタイプ関係を表現した型クラスの ALπ への適用

ALπ の演算子への適用　導入した型クラスを用いて，3.2 節で導入した基本的な演算子 send,recv を拡張し図 3.6 に示す．rep も同様に与える．new，fork についてはサブタイプ関係と関連しないため省略する．これらの関数により，ALπ のプロセスが Haskell に埋め込まれる．

例 3.2 (PiMonad によるセマフォの記述) 図 *3.7* に，*2.3.2* 節で示したセマフォを *PiMonad* によって記述する．ここで send_sync *s* は同期のためのプロセス抽象で，$s!(t);t?();\mathbf{0}$ に対応する．セマフォの最大値の指定に変数 cnt を用い，標準ライブラリ関数 Control. Monad. replicateM_ でプロセス send a () を複製している．□

3.4　Haskell の I/O システムの統合

PiMonad を Haskell の I/O システムと統合する手法について示す．Haskell の IO アクションは，ALπ の出力動作として定式化する．

```
send_sync :: (ValueType a) => O (O a) -> PiMonad a
send_sync ch = do
  (a::L a) <- new
  send ch a
  recv a

semaphore :: (L (O (O (O ()), O (O ())))) -> Int -> PiMonad ()
semaphore sem cnt = rep sem $ \x -> do
  (p::L (O ())) <- new
  (v::L (O ())) <- new
  send x (p, v)
  (a::L ()) <- new
  rep a $ \_ -> do
    r <- recv p
    send r ()
    s <- recv v
    send s ()
    send a ()
  replicateM_ cnt (send a ())

user :: (L (O (O (O ()), O (O ())))) -> PiMonad ()
user sem = do
  (x::L (O (O ()), O (O ()))) <- new
  send sem x
  (p, v) <- recv x
  send_sync p
  -- some works here --
  send_sync v

all :: PiMonad ()
all = do
  (sem::(L (O (O (O ()), O (O ()))))) <- new
  semaphore sem 1 -- binary semaphore
  user sem
```

図 3.7: PiMonad でのセマフォの記述

まず，ALπの出力能力型に，Haskellの IO アクションを対応づける拡張を行う．次に，I/Oシステムから値を返却するよう拡張することで，I/Oシステムとの相互作用を可能にする．

3.4.1　一方向通信

HaskellのI/Oシステムとの相互作用の実装として，まずは値を与えて出力操作を実行するのみの一方向の通信を実装する．

Haskellの入出力は，IOモナドで記述したアクションを介して行われる．(π計算のアクションと区別するためHaskellアクションと呼ぶことにする)．Haskellアクションを出力チャネルとして記述するため，チャネル型Oを以下の様に修正する：

```
data O a =
    O Integer
  | Action String (a->IO ())
```

以後，データ構築子Oのチャネルをローカルチャネル，Actionのチャネルをアクションチャネルと呼ぶ．データ構築子Actionの第一引数は，系内で一意なチャネルの名前を文字列で与える．第二引数は，チャネルに対する出力動作に対応するHaskellアクションを与える．アクションチャネルに対し出力動作を行う時，このHaskellアクションを関数Control．Concurrent．forkIO[JGF96]により並行実行するよう，3.2.5節で導入した評価器を拡張する．

例 3.3 (標準出力への非同期書き込み) *Haskell*アクションのputStrLnを用いて，標準出力へのチャネルcoutを作る：

```
cout :: O String
cout = Action "cout" (\x -> putStrLn x)
```

coutを用いて，標準出力へ文字列"Hello，world!"の書き込みを行うプロセスを以下のように定義できる：

```
hello :: PiMonad ()
hello = send cout "Hello, World!"
```

□

3.4.2 同期と双方向通信

Haskell の I/O システムからの値の返却 ここまでの実装では，非同期出力に対応する Haskell アクションが発火しても，Haskell アクションから評価器への値の返却の手段が存在しないため，Haskell アクションの完了を待機するプロセスや，戻り値を持つような Haskell アクションの呼び出しを記述することが出来ない．

Haskell の I/O システムが返却値を格納し，評価器が取り出すバッファ`RecvBuf`を，FIFO バッファである`Control. Concurrent. Chan`を用いて準備する：

```
type RecvBuf = Chan RecvData
data RecvData =
  forall x. (ValueType x) =>
    RecvData (O x) x
```

`RecvData`は受信に用いたローカルチャネルと受け取った値を引数に持つ．バッファに投入された`RecvData`型のデータは，評価器により取り出され，出力プロセスとして PiMonad のプロセスと並行合成される．こうして値の受け渡しが実現する．例えばアクションチャネル`x`を介して値`v`が送られた場合，バッファにはデータ`RecvData x v`が投入され，評価器によりプロセス`Send x v`が並行合成される．

次に，Haskell アクションが値を`RecvBuf`に格納するための方法を述べる．

Haskell アクション内でのローカルチャネルの扱い PiMonad の評価器から Haskell の I/O システムへローカルチャネルを渡す際，渡すローカルチャネルを，`RecvBuf`へ値を投入するアクションチャネルへ変換する．Haskell アクション側は，これを実行することで，値の返却を行う．このため，前節で導入した`ValueType`に，以下のように変換メソッド`trans`を追加する：

```
class (Typeable x) => ValueType x where
  ...
  trans :: RecvBuf -> x -> IO x
  trans _ x = return x
```

`trans`は変換を行わないデフォルトメソッドを持つ．出力チャネルについてのみこれをオーバーライドする：

```
instance (ValueType x) =>
```

```
          ValueType (O x) where
  ...
  trans buf ch@(O i) =
    return (Action (show i) (\x ->
      writeChan buf (RecvData ch x)))
  trans _ ch = return ch
```

transは，ローカルチャネルを，RecvBufへ値を投入(writeChan)するアクションチャネルへ変換する．アクションチャネルの文字列表現には元のローカルチャネルの整数表現が用いられる(show i).

例 3.4 (ファイル書き込みチャネル) ファイルへの書き込みに用いるチャネルfopenWを図*3.8*に定義する．fopenWはファイル名を表す文字列filenameとチャネルretChの対を受け取り，ファイルを開き，書き込み用チャネルwriteとファイルを閉じるためのチャネルcloseの*2*つを，渡されたチャネルretChを介して返す．書き込み用チャネルは書き込む文字列strとチャネルchを受け取る．文字列がファイルに書き込まれた後，chを介し値 () を送信する．これにより利用者は書き込みの完了を知ることができる．closeも同期のためのチャネルを受け取る．

fopenWを使い，ファイル書き込みは以下の様に書く事が出来る：

```
helloWriter :: String -> PiMonad ()
helloWriter filename = do
  (ret::O (O (String, O ()), O (O ())))
    <- new
  send fopenW (filename, ret)
  (write,close) <- recv ret
  (sync::O ()) <- new
  send write ("Hello world!",sync)
  _ <- recv sync
  send_sync close
```

□

```
fopenW :: O (String, O (O (String,O ()), O (O ())))
fopenW = Action "fopenW" (\(filename,retCh) -> do
    handle <- openFile filename WriteMode
    let write :: O (String, O ())
        write  = Action ("write"++show handle) (\(str,ch) -> do
            hPutStrLn handle str
            putBuf ch ()
          )
        close :: O (O ())
        close  = Action ("close"++show handle) (\ch -> do
            hClose handle >> putBuf ch ()
            putBuf ch ()
          )
    putBuf retCh (write, close)
   )
  where
    putBuf :: O a -> a -> IO ()
    putBuf (Action _ f) v = f v
```

図 3.8: ファイル書き込みチャネル fopenW

3.5　ネットワーク実装

Haskellで提供されるネットワークAPIと，前節で与えたHaskellのI/Oシステムへの拡張を組み合わせて，PiMonadのネットワーク実装を与える．

表面上は2つのチャネルを加えることで実現できる．一つは，リモートホストからの接続を待ち受けるチャネルであり，もう一つはリモートホストへと接続するためのチャネルである．これらを初期チャネルと呼ぶことにする．通信するホスト同士は，まずこれら2つのチャネルを介し相互にローカルチャネルを渡し合う．交換したチャネルを介して，通信の続きを行う．

さらに，前節で定義した出力チャネルに，ネットワーク上の通信ホストの位置情報を含めるよう修正する．他に，Haskellのネットワーク通信に必要なスレッドを準備し，値の直列化を実装する．

3.5.1　基本メカニズム

直列化　Haskellのネットワーク通信は文字列の通信として記述されるため，値型として定義した型クラス`ValueType`に文字列への直列化と，値の復元の関数を追加する．通信はRecvBufを直列化し行う．すなわち，各通信ごとに，ピア間で送信チャネルと値の文字列表現の対がやりとりされる．

アクションチャネルの修正と接続ホスト管理　アクションチャネルを，ネットワーク接続情報を追加するために以下の様に修正する：

```
data O a =
    O Integer
  | Action
     String
     Location
     (PiPeer -> a -> IO ())
```

追加された第二引数`Location`は，ホスト名等のネットワーク上の通信ホストの位置情報を格納するために用いる．Haskellアクションの追加引数`PiPeer`は，(1) 当該ホストのホスト名，(2) 受信のために用いられるローカルチャネルと，その文字列表現を対応付けるテーブル，(3) 利用できるTCPコネクションのリストからなるデータである．

着信モニタ 評価器に TCP コネクションの待ち受けスレッド (モニタ) を追加する．コネクション要求を受け付けた時，モニタは受信データを待ち受けるスレッド，送信のためのスレッドを一つずつ生成する．2 つのスレッドは，それぞれ入力動作と出力動作に対応した通信を行う．

初期入力チャネル 初期チャネルのうちの一つが，初期入力チャネルである．初期入力チャネルは TCP コネクション確立時に入力動作が発火される．全てのホストは，後で導入する名前解決チャネルを介してお互いの初期入力チャネルを得，このチャネルに向けて送信を行い，コネクションを確立する．初期入力チャネルはトップレベルの PiMonad プログラムの引数としてのみ与えられる．初期入力チャネルは型 `I a` を持ち，型変数 `a` はアプリケーション固有の通信の内容を表すように呼び出し側で具体化する．

例 3.5 (エコーバック・サーバー) 初期入力チャネルで着信を待ち，文字列とチャネルの対を受け取り，チャネルを介して文字列をそのまま返すエコーバック・サーバーの例を示す：

```
echo :: I (String, O String) -> PiMonad ()
echo init =
  rep init (\(str, reply) -> send reply str)
```

ここで引数 `init` が初期入力チャネルである．多相的な型付けにより，初期入力チャネルで受け取る型は任意の型を取ることができる． □

名前解決チャネル もう一つの初期チャネルが，名前解決チャネル `resolve` である．名前解決チャネルはホスト名とポート番号の対を受け取り，対応するホストで待ち受けているプロセスの初期入力チャネルのコピーを返す．このチャネルを介した出力動作が，TCP コネクションの確立に対応する．`resolve` は次の型シグネチャを持つ：

```
resolve :: (ValueType a) =>
         O ((Hostname,PortNumber), O (O a))
```

型変数 `a` は，初期入力チャネル同様，使用者側で具体化する．

ネットワーク通信における各チャネル型の役割　初期入力チャネルはリモートホストからのコネクションを待つためだけに用いるため，入力能力型`I a`をもつ．一方，通信において多く用いるのは出力能力型`O a`とチャネル型`L a`である．ALπの局所性のため，他プロセスから入力プレフィクスで受信するチャネルは全て出力能力型である．出力能力型はネットワークの通信ホストの位置情報を持つよう拡張する．一方，`new`で生成するチャネルはチャネル型を持ち，`send`により出力能力のみをリモートホストに渡しそのチャネルで入力することで，継続したネットワーク通信が可能になる．

3.5.2　多相型付けされた初期チャネルと通信の安全性

アプリケーション固有の型を送受信する余地を残すため，2つの初期チャネルは多相的に型付けされている．それゆえ，それぞれのホストで期待する値型が違う可能性があり，通信が失敗するおそれがある．

そこで，初期チャネルでの通信のみ，相互の型の比較を型名のレベルで行う．初期チャネルの型が整合していれば，継続する通信の型は常に整合していると推測される．

3.5.3　問題点

名前の枯渇　　π計算は加算無限個の名前の集合を仮定している．本フレームワークにおいて名前はHaskell処理系の実装に依存した有限の整数で表現されるため，サーバプロセス等長時間動作するプロセスでは名前の数が足りなくなることがある．

振る舞い等価性等の諸規則により使われない名前を特定してごみ集めすることはある程度可能である．しかしながら，I/Oシステムやリモートホストに送信された名前についての参照カウント等を用いて管理しなければ，名前の枯渇を防ぐ事はできない．

プロセスの溢れ　　同様に，それ以上動作しないプロセスがメモリ領域を占有することがある．そうしたプロセスはπ計算の振る舞い等価性により削除できる．例えば，$(\nu x : L)(!x(y).P) \cong_c \mathbf{0}$より，通信プロセス`([NewChan x], ReprRecv x (\y->p))`はこれ以上相互作用しないことが保証されるため，系から取り除くことができる．

3.6 プログラム例

構築したフレームワークを用いてインスタント・メッセンジャーの例を示す．プログラムはクライアントがサーバに接続する事でメッセージをやり取りする．キーボードや画面表示は単一チャネルで表されているものとし，ネットワーク通信に関係のある部分のみ取り上げる．

3.6.1 メッセージ交換プロセス

図 3.9 で与えられるプロセス imMain はサーバとクライアントの両方で実行される．このプロセスはメッセージ交換に共通する部分を与える．最初の 2 つの引数はそれぞれユーザへの入力と出力を表すものとする．残りの 2 つの引数は，リモートのユーザ入力と出力である．imMain は並列に起動される receiver と sender によって成り立つ．

receiver はループ用チャネル rloop で待った後，リモートホストからチャネル rin を介し入力を待つ．メッセージが来ると，直ちにユーザに送られ，メッセージが表示されるまで待つ．表示が完了したら，rloop に値 () を送り次のサイクルを起動する．

sender はループ用チャネル sloop で待った後，ユーザ入力へチャネルを送り，待つ．ユーザ入力がなされた時点でリモートホストへ入力文字列を送る．この際 sloop をリモートホストに送ることで，リモートホストからの応答があった時点で直ちに次のサイクルが起動される．

3.6.2 スタートアッププロセス

図 3.10 の server, client はそれぞれサーバおよびクライアントのスタートアッププロセスである．それぞれ初期チャネルを用いて通信を開始する部分である．サーバ，クライアント共に仮引数 cin，仮引数 cout はそれぞれローカルのユーザ入力と出力に対応するものとする．

server は，初期入力チャネル incoming よりクライアントへメッセージを送るチャネルを rout をクライアントからのメッセージを受け取るチャネルを送るチャネル orin の対を受け取る．ユーザにコネクションが来た旨を伝えた ("connection comes") 後，クライアントからのメッセージを受け取るチャネル rin を受け取り，imMain へと継続する．

```
imMain ::
  (I (String, O ())) -> (O (String, O ())) ->
  (L (String, O ())) -> (O (String, O ())) -> PiMonad ()
imMain cin cout rin rout = do
    (rloop::L ()) <- new
    rep rloop (receiver rloop)
    (sloop::L ()) <- new
    rep sloop (sender sloop)
    send rloop ()
    send sloop ()
  where
    receiver :: L () -> () -> PiMonad ()
    receiver rloop _ = do
      (str, sync) <- recv rin
      (sync::L ()) <- new
      send cout (str, sync)
      _ <- recv sync
      send rloop ()
    sender :: L () -> () -> PiMonad ()
    sender sloop _ = do
      (inp::L String) <- new
      send cin inp
      str <- recv inp
      send rout (str, sloop)
```

図 3.9: メッセージ交換プロセス imMain

client は，引数 hostname および portnumber で与えられるホストの初期チャネルを，名前解決チャネル resolve1 を介して送ったチャネル res より取得する．resolve1 は多相的に型付けされた resolve チャネルを具体型にしている．初期チャネルが取得できた時，ユーザに文字列"connected."を通知し，チャネルのやり取りによりリモートからの受信チャネル rin，リモートへの送信チャネル rout を得て imMain へと継続する．

3.6.3 従来の Haskell ネットワークプログラムとの比較

Haskell のネットワーク API を用いた通信では，文字列と値の相互変換を行う必要がある．通信手順の誤りにより，例えば，整数型の値を受信しようとしているピアに対し，文字列型の値を送信し，値の復元に失敗し実行時エラーとなる場合がある．

本フレームワークでは，初期チャネルの型が一致していれば，データの直列化と復元に失敗せず，後の通信が失敗しないと期待できる．

3.7 おわりに

本章で，我々は非同期局所化 π 計算 (ALπ) に基づくネットワークプログラミングフレームワークとして PiMonad および非同期局所化 π 計算に基づく型付け手法を提案した． 本フレームワークの実装においては，π 計算のラベル付き遷移系を忠実にエンコードした．そのため，本フレームワークに基づくネットワークプログラムの解析に π 計算の解析技法がそのまま扱えるのが利点である．その反面，実行効率の面では既存の並行分散プログラミングの枠組みよりもやや劣ると考えられる．それまでの π 計算に基づくプログラミングの枠組みはプログラミング言語の構文などを一から設計したものである一方，本手法においては Haskell の抽象化能力と静的型付けを活用し，実装を軽量なものとしたことは新規な点である．

PiMonad を定義することで Haskell に対する埋め込み言語として ALπ 計算の演算子を実現した．ALπ では、型付けを行うためにサブタイプ関係が必要であるため，ALπ の型付けにおけるサブタイプ関係の表現には Haskell の標準的な拡張である多引数型クラスを用いた．本手法は単純な coercion による定義ができない一般のサブタイプ関係については適用できない．本フレームワークでは，データに基

```
server ::
  O (O (String, O ())) -> O (String, O ()) ->
  I (O (String, O ()), (O (O (String, O ())))) -> PiMonad ()
server cin cout incoming = do
  (rout,orin) <- recv incoming
  (sync::L ()) <- new
  send cout ("connection comes.", sync)
  _ <- recv sync
  (rin::L (String, O ())) <- new
  send orin rin
  imMain cin cout rin rout

client :: String -> Int ->
  O (O (String, O ())) -> O (String, O ()) ->
  I () -> PiMonad ()
client hostname portnumber cin cout _ = do
  (res::L (O (O (String, O ()), O (O (String, O()))))  <- new
  send resolve1 ((hostname,portnumber), res)
  remote <- recv res
  (sync::O ()) <- new
  send cout ("connected.", sync)
  _ <- recv sync
  (rin::L (String, O ())) <- new
  (irout::L (O (String, O()))) <- new
  send remote (rin, irout)
  rout <- recv irout
  imMain cin cout rin rout

resolve1 :: O ((String, Int),O (O (O (String, O ()), O (O (String, O())))))
resolve1 = resolve
```

図 3.10: スタートアッププロセス server および client

づくサブタイプ関係を仮定しないことで Haskell で ALπ の型システムを表現することに成功した．

ALπ の操作意味論に基づく評価器によって，Haskell の `IO` アクションを π 計算のチャネルとしてモデル化した．プログラマは任意のアクションをチャネルとして記述でき，それ自身も他のチャネルを介して通信できる．具体例としてファイル書き込みチャネルの例を示した．さらにネットワーク通信の実装において，2 つの初期チャネルを用いたネットワークプログラミングを実現した．例としてインスタント・メッセンジャーの例を示した．

第4章　Haskellにおけるセッション型推論の実装

4.1　はじめに

セッション型システム [HVK98] は通信プロトコルの静的検査の方法を提供する型システムである．セッション型を既存のプログラミング言語に導入することにより，通信指向のソフトウェアの信頼性をより容易に高められるようになることが期待される．

しかしながら，セッション型の型付けの方法を既存のプログラミング言語の型システムと統合するのは容易ではない．これはセッション型システムが通信プリミティブの出現順序に依存して型を付けることと，線形型システム [KPT99] にも見られるような型環境の分離を伴うことに由来する．

セッション型の実装は [NT04, PT08, SE08] で既に提案されている．しかしながら，これらにおいてはプログラムコードに手作業での注釈が必要であり，Haskellが備えている型推論の恩恵を受けられない．このため，[NT04] と [SE08] においては 単純なプログラムが必要以上に冗長になる．[PT08] では型推論が働くものの，複数のチャネルを扱う場合にチャネルのスタックを入れ替える操作を明示的に記述しなければならない．

そこで本章では，Haskell における，注釈を必要としないセッション型推論を与える．本手法では，型注釈を必要とした Pucella と Tov [PT08] の手法を拡張する．対象の型システムは 2.4 節でみた本田ら [HVK98] のセッション型である．

型レベル表現におけるマッチング問題　先行研究である [PT08] と [SE08] は複数の通信路のセッション型を追跡するための型環境に相当する 記号表 を Haskell の型に埋め込む点で共通している．推論されたプロセスの型は $\mathrm{P}\,\{c_1 \mapsto s_1; c_2 \mapsto s_2, \cdots\}$ の形をしている．ここで P はプロセス型構築子であり $\{c_i \mapsto s_i; \cdots\}$ は 通信路 c_i にセッション型 s_i を割り当てる記号表である．記号表は型レベルで表現されている

ため，ここでの c_i は値ではなく，そのスコープにおける通信路の識別子の型レベルにおける表現である．

[PT08] では，(型レベルで表現された）記号表において識別子が型変数で表現される．記号表はキーと値のペアのリストあるいはスタックで表現されており，記号表の参照においては型変数のマッチングが不可欠である．しかしながら，Haskellの型レベルプログラミングにおいて，そのような型変数同士のマッチングはできない．

[PT08] は，型レベルプログラミングで記号表を参照することをあきらめ，記号表のスタック表現のトップを操作する `dig` と `swap` で明示的に記述する方法を採用した．[PT08] において，通信プリミティブはスタックのトップにある通信路のみを対象とする．`swap` はスタックのトップと2番目を入れ替える操作である．`dig` p は，プロセス p における通信プリミティブが，スタックの2番目を対象とするように書き換える操作である．

[PT08] では，記号表の情報を拡張しなければスタック操作の自動化は不可能であるとしたが，その方法は文献には示されていない．

主たるアイデア 型レベルのマッチングを実現するため，記号表におけるする識別子の表現に *de Bruijn* レベル [DB72] を表現した自然数を用いる．*de Bruijn* レベルは，識別子の束縛のネストの段数を表す表現である．例えば，ラムダ式 $\lambda x.\lambda y.xy$ は， $\lambda.\lambda.0\ 1$ と表現される．記号表は型レベルのリストで表現され，de Bruijn レベル n の識別子の型はリストの n 番目に位置する．型レベルリストを位置で参照する方法は既存の技法 [KLS04] で知られているため，型推論は全自動となる．

主たる貢献は Haskell において注釈をほとんど必要としないセッション型の推論系を与えることである．さらに，既存の セッション型実装である [PT08] においては通信路の通信能力しかプロセス間で渡せなかったが，我々の体系においては通信路そのものをプロセス間で渡すことができる．この通信路渡しの機能は型レベルのマッチング問題とは関わりはない．このため，[PT08] の手法は，型推論に注釈を必要とするものの，我々の実装と同等の記述能力をもつ．[PT08] を，通信路そのものを渡せるよう拡張する方法についても示す．

本論文の付録に，本章で与えるセッション型の実装の全体を示す．The Glasgow Haskell Compiler 7.0.3 [ghc] で動作する実装 `full-sessions` は Haskell のパッケージシステム Hackage に登録されており，`http://hackage.haskell.org/package/full-sessions/` から取得できる．

関連研究 Neubauer と Thiemann [NT04] は単一の通信路に関するセッション型を Haskell で実装した．この方法では表現能力を制限しており，単一の通信路しか用いないため，通信路の識別子がプログラムコードに陽に現れず，記号表の表現が必要ない．

Pucella と Tov [PT08] は Haskell[Has10], ML[MTHM97], OCaml[Ler11], C# にセッション型をエンコードする一般的な技法を示した．この方法はスタック操作 `swap` と `dig` を利用しているため，型レベルプログラミングをほとんど必要とせず，パラメトリック多相をもつ多くのプログラミング言語に対して適用可能である．その一方で，スタック操作を必要とするため全自動の型推論を達成しているとはいえない．

Sackman と Eisenbach [SE08] はセッション型の全ての機能を Haskell 上に実現したが，この方法ではセッション型の記述をプログラムコードに陽に記述しなければならない．しかしながら，セッション型の構造について豊富な記述能力を認めており，セッション型の型表現の可読性は比較的高いといえる．この点については後の節で議論する．

我々の手法と関連研究の比較を次の表に示す．

	通信路渡し	型注釈	Haskell 以外に適用可能か
Neubauer et al. [NT04]	No	必要なし	No
Pucella et al. [PT08]	限定的	スタックの操作が必要	Yes
Sackman et al. [SE08]	Yes	必要	No
本手法	Yes	必要なし	No

本章の構成は以下の通りである．4.2 節で本手法のアイデアを説明する．4.3 節で本手法による Haskell におけるセッション型推論を与え, 他の手法と比較する．4.4 節において，我々の実装を用いた SMTP クライアントの実装を示す．4.5 節において，通信路渡しについて我々の実装が [PT08] よりも表現能力が高いことを議論し，[PT08] を拡張して同様の表現能力を獲得する方法について述べる．4.7 節ではセッション型実装の実用性について議論する．4.8 節でまとめを述べる．

4.2 de Bruijn レベルを用いたセッション型の実装

セッション型の実現において共通するのは，セッション型の型環境そのものを，型レベルプログラミングの技法を用いて，Haskell の型表現に埋め込むことである．

ここで，型環境とは，プログラム中の変数を索引として型を返す記号表である．これにより，通信プリミティブの型にセッション型の型付け規則による制約を表現でき，セッション型をHaskellで利用できるようになる．

本章の基本的なアイデアは，Haskellの型に埋め込まれるセッション型の型環境を，de Bruijnレベル[DB72]を用いて実装することで，セッション型の型推論を自動化することである．Haskellの型レベルプログラミングにおいては，型の構造に基づく計算を帰納的に定義でき，型推論の過程で計算させることができる．de Bruijnレベルは変数に自然数を割り当てるため，帰納的に扱えることからHaskellの型レベルプログラミングによる自動化と親和性が高い．

型レベルプログラミングは，依存型を持たないHaskellにおいて，プログラムの項に依存する型をエンコードする技法として活用されている[Mcb02]．例えば，通常は依存型を扱うことでしか表現できない配列アクセスの境界条件の制約が，型レベルプログラミングにより表現できることが知られている[KS08]．型レベルプログラミングにおいては，Haskellの型クラス[Jon00]や型族[SJCS08]の機能を用いて，型推論において推論される型を特化する．この方法により，型推論の過程で自然数やリストなどを用いた計算が可能である．多くの具体例はKiselyovらによるheterogeneous collections[KLS04]で与えられている．

Haskellの型に埋め込まれるセッション型の型環境は，索引となる変数に依存する依存型である．しかしHaskellは依存型をもたないため，値レベルの変数を型レベルに直接埋め込むことはできない．そこで変数の型レベルにおける表現として，de Bruijnレベルを採用する．

de Bruijnレベルは，de Bruijnがラムダ式をα変換によらず一意に表現するために提案した方法の一つである．もう一つの方法としてde Bruijnインデックスがある．de Bruijnレベルは，項の外側から順に，束縛変数に自然数を割り当てる．例えば，$\lambda x.\lambda y.x$ と $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$ は，de Bruijnレベルにより，それぞれ $\lambda.\lambda.0$ と $\lambda.\lambda.\lambda.(0\ 2)(1\ 2)$ と表される．de Bruijnインデックスは，変数の出現とその束縛の間に出現する個数で与えられる．例えば，$\lambda x.\lambda y.x$ と $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$ は，de Bruijnインデックスにより，それぞれ $\lambda.\lambda.1$ と $\lambda.\lambda.\lambda.(2\ 0)(1\ 0)$ と表される．

変数の型レベルにおける表現はde Bruijnレベルが適している．de Bruijnレベルは束縛変数に割り当てられる自然数が変数束縛のネストの数によらず変化しない．このため，変数の型レベル表現が一定となり，型環境における変数をうまく表現できる．一方，de Bruijnインデックスは，変数束縛のネストの数によらず項の表現が一定であり合成性をもつのが利点であるが，束縛変数の表現は一定にならな

いため適さない．

de Bruijn レベルによりセッション型の型環境を型レベルに実装でき，Haskell におけるセッション型推論が可能になる．Pucella らの実装では de Bruijn レベルを用いておらず，型レベルプログラミングにおける自動化の制限から，型レベルの辻褄合わせのコードを余分に記述する必要があった．de Bruijn レベルは自然数であるため型レベルプログラミングで扱うことができ，セッション型推論の自動化を可能にしている．

de Bruijn レベルによる実装により，セッション型の型付け規則は Haskell の型制約として直接的に表現できる．本手法による型推論の実装は，本田らのセッション型を忠実に埋め込んだものであり，対応関係は比較的容易に確認できる．本論文の付録に，本実装の全体を示す．

4.3 Haskell におけるセッション型推論

最初に，我々の実装において π 計算の動作意味を実現する通信プリミティブを与える．次に，セッション型を埋め込むために必要な基本的な技法を示す．最後に，de Bruijn レベルを用いて複数チャネルのセッション型を推論する方法を示す．

4.3.1 `full-sessions` における通信プリミティブ

モナドは Haskell において通信を記述する最もよく親しまれた方法であるため，`full-sessions` は π 計算の構文を埋め込まずモナドを用いた構成を採用している．

オリジナルのセッション型システムと我々の実装の対応を保つため，通信プリミティブを継続渡しモナド [Wad92] に対応づける．それぞれのプリミティブの振る舞いは継続渡しモナド `((a -> Pi` $\mathtt{d}_1$`) -> Pi` $\mathtt{d}_2$`)` の型をもつ．ここで `Pi` $\mathtt{d}_i$ はプロセス項の型であり，$\mathtt{d}_i$ はセッション型環境 Δ に対応する．それぞれの通信プリミティブの意味は表 4.1 に示す．この表においては Higher Order Abstract Syntax[PE88] の λ 記法を用い，値ないし通信路からプロセスを構成する構文上の関数を表現している．k は 型 `a -> Pi` $\mathtt{d}_1$ の継続である．可読性のため，プログラムを手続き的に書く `ixdo` 記法 [PT08] を用いる．例えば， `ixdo send` c e`; recv` c と `ixdo fork (send` c e`); recv` c は $\lambda k.c!\langle e\rangle;c?(x);(k\ x)$ と $\lambda k.(c?(x);(k\ x) \mid c!\langle e\rangle;\mathbf{0})$ にそれぞれ展開される．

継続渡しモナドで構成されたプロセスは関数 runS で実行される．runS p は π 計算プロセス $p(\lambda\_.\mathbf{0})$ を実行する．以後，表 4.1 のプリミティブを型 Session をもつ**セッション**と呼ぶ．

	Function	Meaning
通信路生成	`new`	$\lambda k.(\nu s)(k\ s)$
値の出力	`send` $c\ e$	$\lambda k.c!\langle e\rangle;(k\ ())$
値の入力	`recv` c	$\lambda k.c?(x);(k\ x)$ (ただし x は fresh)
選択	`sel`i c $(i \in \{1,2\})$	$\lambda k.c \blacktriangleleft i;(k\ ())$
分岐	`offer` $c\ p_1\ p_2$	$\lambda k.c \blacktriangleright \{1\colon (p_1\ k),\ 2\colon (p_2\ k)\}$
通信路の出力	`sendS` $c\ c'$	$\lambda k.c!\langle c'\rangle;(k\ ())$
通信路の入力	`recvS` c	$\lambda k.c?(d);(k\ d)$
並行動作	`fork` p	$\lambda k.((k\ ())\mid (p\ (\lambda\_.\mathbf{0})))$
Haskell の *I/O* 呼び出し	`io` m	(Haskell の `IO` アクション m を実行)
セッションの再帰	`recur1` $f\ c$	$\lambda k.f\ c\ k$ (セッション $(f\ c)$)
再帰型の展開	`unwind` c	$\lambda k.k\ ()$(セッション型の再帰 `Rec` $n\ u$ を u[`Var` $n \mapsto$ `Rec` $n\ u$] に展開する)

表 4.1: `full-sessions` ライブラリの通信プリミティブ

4.3.2 単一チャネルにおけるセッション型推論

並行動作を含まないプロセス

簡単のため，並行動作を含まないプロセスから始める．この場合において，セッション型はセッションの進行とともに**前進**する．例えば，型 `Send Int End` は，その型をもつ通信路から整数が出力されたのち， `End` に前進する．このようなセッション型の前進を追跡するため，通信路に**前の型**と**後の型**を割り当てる．前の型はセッションが開始する前のセッション型を表す．同様に，後の型はセッションが終わった後の型を表す．

多くの場合において，後の型は２つのセッション型を合成するためのプレースホルダとして機能する．例えば，セッション `send` c `True` における c の前の型は `Send Bool` u であり，後の型は u であり，ここで u は型変数である．これは通信路 c を使う他のセッションを `send` c `True` の後に連接できることを表す．

２つのセッション型の連接は単一化によって行われる．２つのセッション $s_1;s_2$ の連接において，s_1 の後の型は s_2 の前の型と単一化される．連接されたセッション $s_1;s_2$ の前の型は s_1 のそれと同一である．同様に $s_1;s_2$ の後の型は s_2 のそれと同一

である．このようにして，(send c True; send c "abc") は前の型として (Send Bool (Send String u_2)) をもち，後の型に u_2 をもつ．ここで u_2 は以前の u と異なる型変数である．

より複雑な例として，単純な算術を実現するサーバーを示す．

```
server c =
  ixdo x ← recv c;
       y ← recv c;
       offer c (send c (x+y::Int))
               (send c (x<y))
```

server は最初に２つの Int 型の値を入力し，続いて入力したラベルで分岐する．さらに，入力したラベルに従い，型 Int または Bool の値を出力する．

server の前の型と後の型は *GHC* の型検査器によって推論させることができる．これには補助関数 channeltype1 を用いる．項 (channeltype1 server) の型を GHC の対話環境に入力することで，次の応答が得られる：

```
prompt> :t channeltype1 server
channeltype1 server
  :: (Recv Int (Recv Int (Offer (Send Int a)
                                (Send Bool a))), a)
```

(可読性のため，対話環境の応答には適宜改行を挿入している.)

セッション型の双対性の表現

並行動作プリミティブである fork は２つのセッションそれぞれの前の型が双対であることを要求する．これを前述の算術サーバーの例を用いて示す．サーバーと通信するクライアントは，例えば次のようになる：

```
client c =
  ixdo send c 123;
       send c 456;
       sel2 c;
       ans ← recv c;
       io (putStrLn (if ans then "Lesser" else "Greater or Equal"))
```

client において，c の前の型と後の型はそれぞれ (Send Int (Send Int (Select u_1 (Recv Bool u))) と u である．fork を用いて server と client を次のように並行動作させることができる：

```
calc c    =
  ixdo fork (server c);
```

```
        client c;

startCalc =
  ixdo c ← new;
       calc c
```

このコードは，client と server それぞれにおける通信路 c の利用が双対であるため正しく型付けされる．calc の前の型は，[HVK98] の型付け規則が示す通り，Bot である．後の型はその通信路の使用が終了したことを示す End である．これは，server が fork により分岐したのち server が終了するため通信路 c の利用も終了し，それゆえ対応する client でも c の利用が終了するためである．（このような型付けはスレッド生成の機能をもつ体系にみられ，例えば [CDCY07] の **Spawn** 規則における条件 “ended”がそうである．）これを確認する：

```
prompt> :t channeltype1 calc
channeltype1 calc :: (Bot, End)
```

セッションは関数 runS で実行できる．runS startCalc を実行することで，コンソールに “Lesser” が出力される：

```
prompt> runS startCalc
Lesser
```

4.3.3 de Bruijn レベルによる複数の通信路の追跡

型レベルにおいて複数の通信路の使用を追跡するために，型レベルにおいて各通信路に de Bruijn レベルを割り当てる．図 4.1 はあるセッションにおける de Bruijn レベルを示す．図において，変数の de Bruijn レベルを変数の束縛位置における出現の右肩に示した．通信路のみがセッション型をもつため，de Bruijn レベルは通信路を表す変数にのみ割り当てる．すなわち図 4.1 において変数 c, d, e, f には de Bruijn レベルを与えるが，x には与えない．

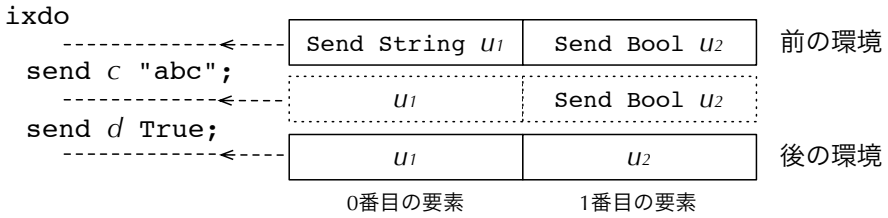
de Bruijn レベルはチャネルの型に埋め込まれる．チャネルは型 Channel t n をもち，ここで n は de Bruijn レベルであり，t は「型のタグ」[**?**] である．型のタグは，runS で実行されるセッションの外側に通信路がエスケープするのを静的に防止するために付与される．de Bruijn レベルを示す自然数はペアノ数を表す型 Z と S n で表現され，それぞれ 0 と $n+1$ を意味する．例えば，de Bruijn レベル 2 をもつ通信路は型 Channel t (S (S Z)) を持つ．この自然数は型環境における位置を指している．

```
ixdo c^n ← new; d^{n+1} ← new;
     fork (ixdo e^{n+2} ←catch c; ...)
     x ← recv d;
     f^{n+2} ← catch d;
     ...
```

図 4.1: セッション中に出現する変数の De Bruijn レベル

前の型と後の型に対応する２つの型環境が型 Session に埋め込まれる．それぞれ**前の型環境**，**後の型環境**と呼ぶ．

セッション型の型環境はセッション型のリストで表現され，各要素は de Bruijn レベルにより参照され操作される．例として，図 4.2 にセッション send c "abc"; send d True と，このセッションがもつ前の型環境と後の型環境を示す．図において，c と d の前の型はそれぞれ Send String u_1 と Send Bool u_2 であり，後の型はそれぞれ u_1 and u_2 である. 図 4.2 には send c "abc"の直後における型環境も示した．この文脈では，c は セッション型 u_1 を d は セッション型 Send Bool u_2 をもつ．



(c,d の de Bruijnレベルをそれぞれ 0,1 とする．すなわち
c :: Channel t Z と d :: Channel t (S Z) を仮定する)

図 4.2: de Bruijn レベルによる型環境の表現

セッションは型 Session t ss tt a をもつ．ここで ss と tt はそれぞれ前の型環境と後の型環境である. 型変数 a はセッションが返す値の型であり，t は型のタグである．ss と tt は型レベルのリスト [KLS04] で表現される．型レベルリストは ss :>

u または Nil のどちらかであり，前者はコンスを，後者は空リストを表す．ss :> u において ss はリストの残りの部分であり， u はセッション型である．型構築子 :> は左結合である．例えば ss:>a:>b は (ss:>a):>b と読む. 通常のリストと異なり，リストは末尾から数えることとする．一般的なリストの記法と異なり，左手にリストの残りが来るため，リストの位置は左から数えればよい．例えば，リスト Nil:>a:>b:>c の 0 番目の要素は a である. 具体的な例として，図 4.2 のセッション (send c "abc"; send d True) は，型 Session t (Nil:>Send String u_1:> Send Bool u_2) (Nil:>u_1:>u_2) () をもつ．

本手法においては，一般に，de Bruijn レベルは定数でなく，$n+1$ や $n+2$ といった値をとる．例えば，セッション型環境 ss の長さが n であり，c と d の型が Channel t n and Channel t (S n) のとき，(send c 1; send d True) は型 SList ss n => Session t (ss:>Send Int u_1:> Send Bool u_2) (ss:>u_1:>u_2) () をもつ．型クラス SList ss n はセッション型環境 ss の長さが n であることを示す．このように，型変数 ss がプレースホルダーとして現れることにより，後の型環境で新たに識別子を追加することができる余地を残している．

新しい通信路が生成されるとき，後の型環境は生成された通信路のために拡張される．プリミティブ new と catch において，そのような型環境の拡張がおこる．new は前と後の型環境がそれぞれ ss と ss:>Bot である．同時に new は新たに生成された通信路に型 Channel t n を割り当てる．ここで n は 型環境 ss の長さであるため，生成された通信路は型 Bot をもつことになる．

図 4.3 はセッション (c ← new; fork (send c True)) の前と後の型環境を示している. 後の型環境は新しく生成された通信路のセッション型を保持している．新しい通信路の後の型は Send Bool End の双対であり，並行動作しているプロセスと通信するために適切なセッション型が割り当てられている．

同様に，catch c の前と後の型環境はそれぞれ ss と tt:>u' となっている．ここで tt は，ss の要素のうち c が指す型 Catch u' u_1 を u_1 に前進した，他の要素については ss と同一の要素をもつリストである．catch により新たに束縛される（渡された）通信路の de Bruijn レベルは ss（および tt）の長さに等しい．図 4.4 は catch の使用に伴うセッション型環境を示している．

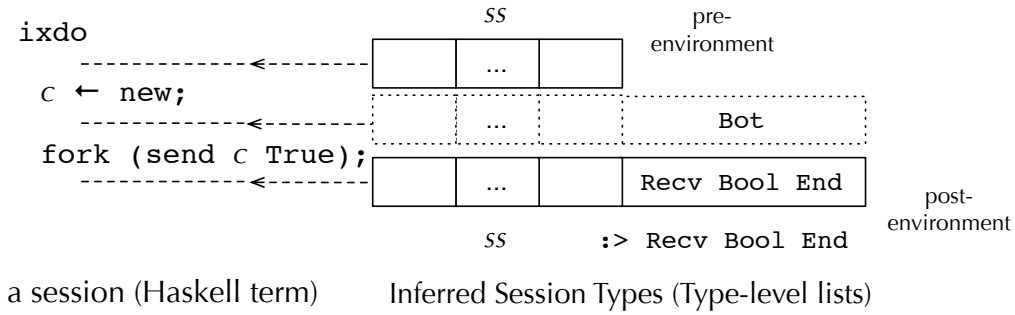


図 4.3: new による型環境の拡張

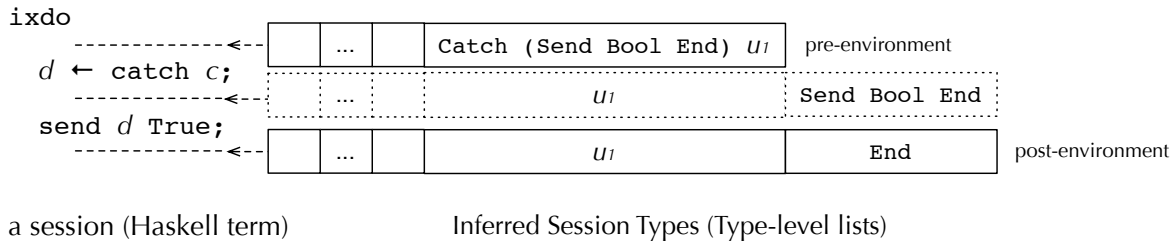


図 4.4: catch による型環境の拡張

4.3.4 既存の実装との比較

de Bruijn レベルによる型環境のエンコーディングにより，既存の実装で必要とされた型注釈のほとんどが不要になった．他の実装との比較により，注釈が不要である利点を議論する．

スタックに基づく実装 Pucella と Tov による実装 [PT08] は型環境の表現にスタックを採用している. 各プロセスは通信路のスタックを保持しており，各通信プリミティブはスタックのトップにある通信路しか使うことができない．このため，複数の通信路を扱うプログラムではスタックを陽に操作しなければならない．スタック操作のためコンビネータ dig と swap が提供されている．swap はスタック最上部の 2 つの要素を入れ替える．一方，dig は引数にとったセッションを，一段深いスタックを参照するセッションに変換する．例えば，c と d がこの順にスタックに積まれているとすれば，次のコードは我々の実装における (send c "abc"; send

d True) と等価である:

```
ixdo send c "abc"; swap; send d True
```

または，次のように書くこともできる：

```
ixdo send c "abc"; dig (send d True)
```

チャネルの数が増えるにつれ，より多くのスタック操作が必要になる．[PT08] においてこの問題は議論されているものの解法が与えられておらず，我々が用いたような数による型環境の参照については触れられていない．

セッション型の手動構築 Sackman と Eisenbach [SE08] の実装は豊富な通信プリミティブを持つものの，その代償としてセッション型を手動でプログラムコードに記述しなければならない．通信路の他に，プロセス固有の識別子 Pid による通信が可能である．Pid による通信のもっとも単純な例を次に示す．この例は整数 10 を他のプロセスに渡して終了するプロセスである．これは我々の実装における runS (ixdo c <- new; fork (send c 10); recv c) と等価である．

```
(s, a) = makeSessionType (
           newLabel ~>>= λa →
           a .= send (undefined :: Int) ~>> end
           ~>> sreturn a)

p = run s a (ssend 10) srecv
```

ここで makeSessionType はセッション型の列 s とセッション型 a を返している．makeSessionType の引数ではセッション型が手続き的な方法で構築されている．この実装でもまた，プロトコルの数が増えるにつれ，手作業でのセッション型の記述が必要になる．通信路による通信でも同様である．

4.4 SMTP クライアントの記述例

full-sessions のネットワーク通信機能を，複数の通信路を用いた SMTP[Kle08] クライアントの記述例により示す．単一の通信路を用いた SMTP クライアントの型付けは [NT04] により示されている．

表 4.2 にネットワーク通信のための追加のプリミティブを示す．ネットワークプロトコルをモデル化するために，型ベースの分岐構文 sel$i$N と offerN を追加した．sel$i$N と offerN は何も通信しないが，それぞれ後に連接された出力・入力プリミティブに基づき選択と分岐を実現するための注釈である．

	Syntax	Meaning
サービスへの接続	$c \leftarrow$ `connectNw` s	サービス s に接続し通信路 c を確立する
型ベースの分岐	`offerN` c p_1 p_2	２つの受信セッション p_1 と p_2 に分岐する．
選択	`sel`i`N` c $(i \in \{1,2\})$	分岐 `Select` u_1 u_2 のうち どちらが選択されたのかを示す

表 4.2: ネットワーク通信のための拡張プリミティブ

SMTP クライアントの実装を次に示す．まず，SMTP のコマンドと応答を次の型で表現する．

```
-- Types for SMTP commands.
newtype EHLO = EHLO String
newtype MAIL = MAIL String
newtype RCPT = RCPT String
data    DATA = DATA
data    QUIT = QUIT
newtype MailBody = MailBody [String]

-- Types for SMTP server replies (200 OK, 500 error and 354 start mail input)
newtype R2yz = R2yz String; newtype R5yz = R5yz String; newtype R354 = R354 String
```

TCP のキャラクタストリームを基礎とした通信を扱うのに必要な，それぞれのコマンドや応答の構文解析と印字の機能は与えられているものとする．SMTP クライアントの本体は次の通りである：

```
-- auxiliary functions
send_receive_200 c mes = ixdo send c mes; (R2yz _) ← recv c; ireturn ()
send_receive_354 c mes = ixdo send c mes; (R354 _) ← recv c; ireturn ()

sendMail c d = ixdo  -- the body of our SMTP client
  (R2yz _) ← recv c                    -- 応答220 を受信
  send_receive_200 c (EHLO "mydomain") -- EHLO コマンドを送り，250 を受信
  unwind0 c                            -- 注釈: 外側のループを表す再帰型を展開
  sel1N c                              -- 注釈: MAIL FROM を送信するための選択
  from ← recv d                           -- (1) 送信者のメールアドレスを準備
  send_receive_200 c (MAIL from)       -- コマンドMAIL FROM を送り，250 を受信
  unwind1 c                            -- 注釈: 内側のループを表す再帰型を展開
  sel1N c                              -- 注釈: RCPT TO を送信するための選択
  to ← recv d                             -- (2) 受信者のメールアドレスを準備
  send c (RCPT to)                     -- コマンドRCPT TO を送る
  offerN c (ixdo                       -- 応答によりセッションを分岐
    (R2yz _) ← recv c                  -- もし 250 OK を入力したならば
    sel1 d; mail ← recv d                 -- (3) メールの本文ををを準備
```

```
    unwind1 c                          -- 注釈: 内側のループを表す再帰型を展開
    sel2N c                            -- 注釈: DATA を送信するための選択
    send_receive_354 c DATA            -- コマンドDATA を送り， 354 を受信
    send_receive_200 c (MailBody mail) -- メール本文を送信
    unwind0 c                          -- 注釈: 外側のループを表す再帰型を展開
    sel2N c                            -- 注釈: QUIT を送信するための選択
    send c QUIT; close c               -- 注釈: QUIT を送信し接続を閉じる
   ) (ixdo
    (R5yz errmsg) ← recv c;            -- もし500 ERROR を受信したならば
    sel2 d; send d errmsg;                -- (4) エラーメッセージを出力
    send c QUIT; close c)              -- QUIT を送信し接続を閉じる
  close d
```

sendMail は c と d の２つの通信路を引数にとる. 前者は SMTP サーバーと通信するために，後者はメールの情報を別のプロセスから入力して準備するために用いられる. 補助関数 typecheck2 により 2 引数をとるセッションの型を調べられる. typecheck2 sendMail の型を GHC の型検査器に問い合わせることで，次の応答が得られる：

```
typecheck2 sendMail :: (SList ss l, IsEnded ss b1) ⇒ Session t
  (ss :>
     Recv R2yz (Send EHLO (Recv R2yz (Rec Z (SelectN
       (Send MAIL (Recv R2yz (Rec (S Z) (SelectN
         (Send RCPT (OfferN
           (Recv R2yz (Var (S Z)))
           (Recv R5yz (Send QUIT Close))))
         (Send DATA (Recv R354 (Send MailBody (Recv R2yz (Var Z))))))))
       (Send QUIT Close))))) :>
     Recv String (Recv String (Select
      (Recv [String] Close)
      (Send [String] Close))))
  (ss :> End :> End) ()
```

クライアント側の SMTP プロトコルが，c の前の型により表現されていることが見て取れる．SMTP サーバーのプロトコルは，クライアントの型の双対をとることで得られる．A server that have the dual of this type can communicate with this client.

これら２つの通信路について，スタック操作などの注釈は不要であった．一方, [PT08] では d の出現の前後で swap 操作が必要となる．つまり，(1), (2), (3), (4), の直前と直後に swap; を挿入しなければならない．より多くの通信路が関わるプログラミングでは，さらに複雑なスタック操作が要求される．我々の体系は複数の通信路を扱う際にも注釈が必要なく，簡潔な記述を保つことができる．

4.5 表現能力の比較

本節では我々と他の実装の表現能力を比較する．de Bruijn レベルによるエンコーディングはプログラムに必要な注釈の数を減らすものの，表現できるプロセスのクラスに違いがないことを示す．[PT08] は通信路渡しに型を与える高階セッションの機能に制限があるため，実際には我々が対象とする π 計算の方が表現能力が高い．しかしながら，この表現能力は de Bruijn レベルに由来したものではなく，[PT08] の枠組みに通信路渡しの機能を加えることができる．

4.5.1 通信能力渡しと通信路渡し

[PT08] は通信路渡しのプリミティブを備えておらず，その代わり通信路の**通信能力**をやり取りするプリミティブ send_cap と recv_cap をもつ．このような通信能力渡しは，通信路渡しよりも表現能力が低く高階セッションの能力を十全に活用できないことを議論する．

send_cap と recv_cap は通信路を用いて同期を行うものの，その通信路を介したデータのやり取りは伴わない．これらのプリミティブでは，送信者の側がもつ，ある通信路の能力が受信者の側に移譲される．

いくつかの場合では通信能力渡しは通信路渡しを模倣するのに十分である．ここで，我々の実装を用いたプログラムを send_cap と recv_cap で書き換える例を示す．我々の実装を用いた次のプログラムは

```
p c  = ixdo d ← new;
           throw c d;
           str ← recv d;
           io (putStrLn str)
q c  = ixdo d ← catch c;
           send d "Hello"
pq c = ixdo fork (q c);
           p c
```

セッション p は通信路 d を生成し，それを c を介して q に送る．次に d を介して文字列を受信し，コンソールにその文字列を出力する．セッション q は c で受信した通信路に，文字列 "Hello"を出力する．

プリミティブ new と fork が [PT08] で提供されていれば，次の通り書き換えられる：

```
p' c d = ixdo send_cap c;
```

```
              dig (ixdo str ← recv d;
                       io (putStrLn str))
q' c d = ixdo recv_cap c;
              dig (send d "Hello")
pq' c  = ixdo d ← new;
              fork (q' c d); p' c d
```

ここで通信路渡しが使えないため，通信路dを前もってp'とq'の間で共有し，p'からq'に通信能力を伝達する方法を採っている．

通信能力渡しの問題点は，受信側に渡される通信能力をもつ通信路が前もって知られていなければならないことである．この制限は分散アプリケーションにおいては致命的である．さらに，再帰かループを含む場合においても，上でみた書き換えは不可能である．次のプログラムは，新しく生成した通信路を，別の通信路cを介して繰り返し送信し，さらにその通信路に文字列"Hello"を出力する．

```
loop c = ixdo unwind c; d ← new; throw c d; send d "Hello"; recur1 loop c
```

newがループの内側にあるため，このチャネル生成はループの外側にくくりだすことができない．このようなプログラムは[PT08]の枠組みでは不可能である．

4.5.2 digとswapによる通信路渡しの実現

[PT08]では，型環境は識別子とセッション型の対のスタックで表される．より具体的には，通信路の型はChannel iであり，この通信路のセッション型を表す型環境のエントリは型Cap i e rで表現される．ここでiは通信路の識別子の型レベル表現であり，eとrがセッション型である．eは再帰型の展開のために用いられるが，本筋から外れるためここでは詳述しない．

send_capの制限は，プロセスをまたがり送受される通信路の名前が**静的に**定まることを要求していることに由来する．このことはsend_capの型シグネチャから確かめられる：

```
send_cap ::
     Channel i
  → Session (Cap i e (Cap i' e' r' :!: r), (Cap i' e' r', x))
            (Cap i e r, x)
            ()
```

通信路の前の型はCap i e (Cap i' e' r' :!: r)であり，ここでCap i' e' r' :!: rは，識別子i'をもち，セッション型r'をもつ通信路を送信すること示している．同時に，前の型環境のエントリCap i' e' r'が後の型環境では取

り除かれ，送信側には通信能力が残らないことを示している．recv_cap の型シグネチャは次の通りである．

```
recv_cap ::
     Channel i
  → Session (Cap i e (Cap i' e' r' :?: r), x)
            (Cap i e r, (Cap i' e' r', x))
            ()
```

send_cap と同様に，渡される通信路の名前 i' が型に記述されている．後の型環境にはエントリ Cap i' e' r' が増え，受信側に通信能力が移動したことを示している．

send_cap と recv_cap から，渡されるチャネルの識別子 i' を除くため，識別子をもたずチャネルの能力のみを表す型 Cap2 e' r' を導入する．Cap2 を用いて，通信路の送信側の型シグネチャは次の通り記述できる：

```
send_chan ::
     Channel i
  → Channel i'
  → Session (Cap i e (Cap2 e' r' :!: r), (Cap i' e' r', x))
            (Cap i e r, x)
            ()
```

一方，受信側は，受信した新しい通信路の型レベル表現と，そのエントリを型環境加えるために，フレッシュな型変数を導入する必要がある．このため型シグネチャはかなり複雑になる．

```
recv_chan ::
     Channel i
  → (forall i'. Channel i'
             → Session (Cap i e r, (Cap i' e' r', x))
                       (Cap i e rr, y)
                       ())
  → Session (Cap i e (Cap2 e' r' :?: r), x)
            (Cap i e rr, y)
            ()
```

4.6 セッション型の実装に関するその他の側面

4.6.1 セッション型の再帰の表現

セッション型の多くの文献では*equi-recursive*な再帰型を扱う[Pie02]．equi-recursiveな再帰型とは，型$\mu t.T$とその展開$T\{\mu t.T/t\}$を同一視する型である．残念なことにHaskellや多くのプログラミング言語はそのような型付けができない．それゆえセッション型の実装[NT04, SE08, PT08]は，再帰型のもう一つの表現であるiso-recursiveな方法を採っている．

Neubauerと Thiemannによる最初の実装[NT04] NeubauerとThiemannの方法は，再帰を伴うセッション型毎に新しい型宣言を導入する．型`Rec :: (* → *) →`は，型コンストラクタの不動点を与える次のような構造をもつ：

```
newtype Rec f = MkRec (f (Rec f))
```

再帰的なセッション型を宣言するために，再帰する箇所を型パラメタで抽象化した型を宣言する：

```
newtype G self = G (Send Int self)
```

ここで型変数`self`は再帰する型それ自身を表すプレースホルダである．`Rec`をこの型に適用することで，μt.`Send Int` tと同型な型`Rec G`が得られる．

このようなiso-recursiveな型表現は手動で展開されなければならず，その実装は型クラスで与えられる．型`Rec G`を`Send Int (Rec G)`に展開する場合を考える．展開のための型クラス`RECBODY`は[NT04]で次の通り与えられている．

```
class RECBODY t c | t → c where ...
```

最初の型パラメータ`t`は展開前の再帰型であり，２番目の型パラメータは`c`展開された再帰型である．関数依存[Jon00] `t → c`を与えることで，型注釈なしでもHaskellの型検査器が自動的に`t`から`c`を推論できるようになっている．実際の展開は型クラス`RECBODY`のインスタンスとして，次のように与えられる．

```
instance RECBODY (Rec G) (Send Int (Rec G)) where ...
```

しかしながら，このように一つの再帰型ごとに型とインスタンスを宣言するのは煩雑である．[NT04]ではセッション型を陽に宣言する必要があり，さらなる煩雑さはなるべく避けられるほうが望ましい．我々の実装や他の２つの実装はそのような宣言は不要である．次に我々の実装とPucellaとTovの実装における再帰型

のエンコーディングをみる．[SE08] は再帰型のエンコーディングに関しては述べられていない．

型レベル計算による再帰型の展開 我々が与えた実装においては，再帰型は `Rec Z (Send Int (Var Z))` の形をしている．ここで `Rec` n r において型 n は再帰変数の束縛の de Bruijn レベルを表しており，`Var` n はその参照である．つまり `Rec Z (Send Int (Var Z))` は μt.`Send Int` t を意味する．プリミティブ `unwind` が再帰型を展開する．例えば，セッション型 `Rec Z (Send Int (Var Z))` は `Send Int (Rec Z (Send Int (Var Z)))` のように展開される．もう一つのプリミティブ `recur1` f c は fc と振る舞い的に等価であるが，c 以外のセッションの前の型が終了していることを型で強制する．このような注釈は，チャネルが明示的な終了型を与えず型推論器がうまく推論できないときに有効に働く．

我々のエンコーディングは Haskell の型レベル計算に強く依存している．一方，Pucella と Tov によるエンコーディングは Haskell に依存せずより一般的な枠組みでも扱えるのが利点であるが，型表現は少し複雑になる．

再帰本体の展開の遅延 [PT08] [PT08] においては，再帰変数の束縛の型レベル表現に de Bruijn インデックス [DB72] が用いられる．例えば，整数を繰り返し送信するセッションの型は `Rec (Send Int (Var Z))` となる（ここでの `Rec` は前項とは異なる型である）．型 `Var` n は型レベルの再帰変数の表現であり，n は de Bruijn インデックスの束縛位置を表しているである．.

[PT08] における能力型は `Cap t e r` の形をしており，`t` は型のタグ [**?**] であり `r` はセッション型である．２番目の型パラメータ `e` は 再帰変数により束縛されている型のスタックである．このスタックは 4.3.4 節における複数チャネルを管理するスタックとは異なる．

この記法を使って，再帰型の展開はあたかも遅延されたかのようになる．例えば，再帰的なセッション `Cap t e (Rec (Send Int (Var Z)))` の展開考える．再帰変数の束縛はスタックに積まれ，展開後の型は `Cap t (Send Int (Var Z),e) (Send Int (Var Z))` となる．計算が前進し， `Cap t (Send Int (Var Z),e) (Var Z)` となり再帰変数が露出した時点で，代入が実際に発生し `Cap t (Send Int (Var Z),e) (Send Int (Var Z))` となる．

このように，記法を工夫することで Pucella と Tov は言語非依存な形でセッション型の再帰をエンコードしている．一方，我々の実装は他の部分で型レベル計算

に強く依存しているため型レベルプログラミングを避ける理由はなく，直接的な形で再帰を実現している．

4.6.2 Pidによる通信

[SE08]と他の枠組みの違いの一つは，[SE08]ではプロセスの識別子を介した通信プリミティブが提供されていることである．自由度の観点からいえば，この枠組みはPidと通常の通信路の両方が平易に扱えるので望ましい．しかしながら，[SE08]では通信路の使用は多くの注釈を要求するため，記述が煩雑になりがちである．[SE08]において典型的な，Pidを用いたプロセス間通信の例を次に示す．これはparentがプロセスchildを起動し，整数52を送信する例である:

```
(st, a) = makeSessionType (
            newLabel ~>>= λa →
            a .= recv int ~>> send bool ~>> end ~>>
            sreturn a)
  where
    int = undefined :: Int
    bool = undefined :: Bool

parent = fork a dual (cons (a, notDual) nil) child
         ~>>= λ(_, childPid) →
         createSession a dual childPid
         ~>>= λchildCh →
         withChannel childCh (ssend 52 ~>> srecv)
         ~>>= sliftIO . print

child _ parentPid
       = createSession a notDual parentPid
         ~>>= λparentCh →
         withChannel parentCh
           (srecv ~>>= ssend . ((==) 42))
```

ここでaは項にエンコードされたセッション型の表現である．stはPidに関連づけられたセッション型だが，ここでは使われない．値dualとnotDualは，プロトコルのどちら側を使っているのかを示す定数である．実際の通信はwithChannelの２番目の引数に隠されている．このような複雑さは型レベルの記号表を扱うために引き起こされる．

これと比較すると，我々の枠組みにおける同等のセッションの記述は

```
parent = ixdo
```

```
  ch ← new;
  fork (child ch)
  send ch 52 >>> recv ch >>>= io . print

child ch = recv ch >>>= send ch . ((==) 42)
```

となる．de Bruijn レベルを用いた記号表の表現のおかげで，我々の実装においては必要最小限の通信記述のみで目的を達成できる．

4.7 セッション型プログラムの可読性と注釈

本節で，セッション型実装を用いたプログラミングのやりやすさをいくつかの観点から議論する．

型推論と型注釈のトレードオフ 4.3.4 節でみたように, 我々の実装で必要とされる注釈の数は他の実装よりも少ない. しかしながら，項レベルでセッション型を構築する [SE08] の実装にはいくらかの利点もある．(1) [SE08] においては，再帰する部分にラベルを用いることでより平易に再帰型が構成できるようになっている．一方，4.4 節でみた我々の実装を用いた SMTP クライアントの例では，再帰的なネットワークプロトコルを扱うために，項に注釈 unwind_i を入れなければ鳴らなかった．(2) 型注釈を常に書くようにすれば，セッション型のサブタイピング [GH05] を導入できるようになる．一方，我々の実装では並行合成される一方のプロセスにおけるセッション型からもう片方を推論する．ここでセッション型の双対性は一対一対応であるため，サブタイピングをエンコードする余地が失われている．

型エラーメッセージの読みやすさ 型検査において，２つのセッション型の双対性が確認できなかった場合には，型エラーとなる．例えば，4.3.2 節のセッションにおいて整数 456 を文字列"456"に置き換えた場合，次のエラーメッセージが報告される：

```
examples/calc.hs:<xx>:0:
    Couldn't match expected type ‘[Char]’ against inferred type ‘Int’
      Expected type: tt' :> Send [Char] a
      Inferred type: tt' :> Send Int (Select (Recv Int End) (Recv Bool End))
    When generalising the type(s) for ‘plus’
```

このエラーは client の前の型が期待される型と異なることを示している. エラーが報告されたプログラムコード上の位置<xx> は fork の位置となっている．このメッセージにより，プログラマは並行合成される 2 つのプロセス間での通信路の利用に矛盾があることに気づける．一般に型レベルプログラミングは可読性の低いエラーメッセージを生成しがちだが，我々の実装においてはエラーメッセージを簡潔なものにとどめることに成功している．

4.8　おわりに

本章では Haskell におけるセッション型実装を示した．我々の実装は，[PT08] で必要とされたスタック操作が不要になっている点で優れている．さらに，de Bruijn レベルの簡潔さにより，通信路渡しなどのエンコーディングも比較的容易になっている．

ある言語を別の言語に埋め込む場合においては，束縛の扱いに注意を要する [ABF+05]．我々の実装では，項レベルの計算と型レベルの計算で束縛の扱いを区別した．項レベルでは，HOAS の方法を用いて，フレッシュな通信路の束縛をラムダ抽象で表現した．さらに，通信路の型レベル表現においては，項レベルの束縛の de Bruijn レベルを型にエンコードすることで，複数のチャネルのためのセッション型推論が可能になった．我々の手法は Haskell の型レベルプログラミングの方法に依存しているため，残念ながらこの手法を他のプログラミング言語に移行するのは簡単ではないが，例えば Scala[OSV08] などの型レベルプログラミング技法が豊富なプログラミング言語や，ATS[Xi, Xi07] などの依存型言語においては，本手法の応用が期待できる．

第5章　主部簡約性をもつセッション型システム

5.1　はじめに

本田らのオリジナルのセッション型システムでは，チャネル渡しを伴うプロセスにおいて主部簡約が成立しない例があることが吉田ら [YV07] により指摘された．これをみるために，まず次にリモートプロシージャコール（RPC）の例を与える．次のセッション型は，まず整数を入力し，次に整数を出力するプロトコルを与えている：

$$?\mathsf{int};!\mathsf{int};\mathsf{end} \tag{5.1}$$

この型は，入力した整数の２倍の値を出力する次のプロセスにおける通信路 *rpc* に付与することができる．

$$rpc?(x);rpc!\langle x*2\rangle;\mathbf{0} \tag{5.2}$$

$$init?((dbg));rpc?(x);\Big(rpc!\langle x*2\rangle;\mathbf{0} \;\Big|\; dbg!\langle x\rangle;dbg?(y);\mathbf{0}\Big) \tag{5.3}$$

(5.3) は通信路 *init* を介して通信路を受信し，次に RPC の引数を *rpc* を介して入力する．最後に，*rpc* を介して計算結果を出力するとと並行して，引数を *dbg* にそのまま出力し，何かを入力する．プロセス (5.3) は型 (5.1) を *rpc*，次の型を *init* に割り当てれば型付けできる．

$$?[!\mathsf{int};?\mathsf{int};\mathsf{end}];\mathsf{end} \tag{5.4}$$

ここで環境から通信路 *rpc* が (5.3) の *init* に送られたとする．RPC のクライアントは通信路 *rpc* の**他方の端点**を保持しており，かつ，そのプロセスの文脈では *rpc* が次の型をもつ．

$$!\mathsf{int};?\mathsf{int};\mathsf{end} \tag{5.5}$$

この型は (5.1) と双対であり，さらに型 (5.4) の [·] の内側の型と同一である．このため，RPC のクライアントが *init* を介して *rpc* を送るのは合法的である．1 ステップ感訳することで，プロセス (5.3) で束縛された名前 *dbg* は次のとおり *rpc* に置換される．

$$rpc?(x);\Big(\underline{rpc!\langle x*2\rangle;\mathbf{0}} \,\Big|\, \underline{rpc!\langle x\rangle;rpc?(y);\mathbf{0}}\Big) \tag{5.6}$$

ここで，オリジナルのセッション型システム [HVK98] は (5.6) を型付けできず，主部簡約性が成立しない. この不成立は下線部どうしでプロトコルが構文的に不整合するのが原因である. | の左手は整数の出力であるが，これは整数を出力し整数を入力する | の右手が期待する振る舞いと異なる．

(5.3) と (5.6) の両方は，型付け可能であるべきである．まず, (5.3) は 受信した通信路が *rpc* 以外であれば正しく振舞い，簡約後の型もさらに, もし受信されたチャネルが *rpc* であり (5.6) に簡約された場合でも，このプロセスはデッドロックしているため，定義により通信安全性は保たれている．クライアントとサーバー通信するのに使われていた *rpc* の両端点がプロセス (5.6) に集まっており，(5.6) はこれ以上環境と *rpc* で通信できない．デッドロックの検出はかなり複雑な型解析を必要とする [KSS00, CDCY07, CdY08] が，我々の目的はデッドロックフリーを達成することではなく，安全性と簡潔さの両立にある．

この観測に基づき，我々は主部簡約性をもつセッション型システムの拡張 **R** を提案する．鍵となるアイデアは，セッションの不整合を示す特別な型 `0dd` を導入し，ある部分において不整合を意図的に認めることである．例えば，(5.6) における通信路 *rpc* は型 `!int;0dd` を持ち，*rpc* に整数が出力された後は，プロセスがエラー状態に遷移するかもしれないことを示している．そして **R** における主部簡約定理は，**R** におけるすべてのセッション型は `0dd` に変化しない，すなわちプロセスはエラー状態に遷移しないことを述べる．

R の設計の核は [HVK98] で型付け可能なプロセスをすべて認めることである．まとめると，我々は 次の性質をもつ **R** の型判定 $\vdash_R$ を確立する．

1. $\vdash P$ ならば $\vdash_R P$.
2. $\vdash_R P$ ならば P は安全である．
3. 主部簡約性が成立する．すなわち $\vdash_R P$ かつ $P \longrightarrow P'$ ならば $\vdash_R P'$ となる．

以上より，簡約の長さに関する自明な帰納法により，$\vdash P$ なる P について，P が常に安全なプロセスにしか簡約されないことが示される．

本章の構成は次の通りである．5.2 節で, 本田らの型システム [HVK98] における主部簡約の不成立について詳細に述べる．5.3 節で, 主部簡約性をもつ型システム **R** を提案する．5.4 節で，**R** の主部簡約性を証明する．最後にまとめを 5.5 節で述べる．

5.2 主部簡約性の不成立

本節で，本田らのセッション型システム [HVK98] における主部簡約の不成立について詳細に調べる．主部簡約は成立しないものの，通信安全性は損なわれていないことを述べる．このことは，[HVK98] の型付け規則を再編成することで主部簡約性を回復できる可能性を示唆する．

主部簡約の不成立は，次のような通信路渡しを伴う簡約の特別な場合に起こる．

$$s!\langle\!\langle t\rangle\!\rangle;P \mid s?(\!(u)\!);Q \longrightarrow P \mid Q\{t/u\} \tag{5.7}$$

ここで $t \in fc(Q)$ かつ $Q\{t/u\}$ が型付けできない場合がある．この型付け不能性は図 2.2 の [Conc], [CRes], [Thr] に由来する．

5.2.1 簡約に伴う型の変化

主部簡約性を不成立にする簡約はすべて *change* 簡約に伴い型が変化している．[HVK98] を含む多くの型システムにおいて主部簡約とは型が簡約によって変化しないことを意味するが，型が変化するからといって通信安全性が損なわれるわけではない．

典型的なのは**転送者** (forwarder) と呼ばれる，値を s から t へと転送する次のプロセスである [YV07]：

$$\vdash s?(x);t!\langle x\rangle;\mathbf{0} \triangleright s : ?S;\mathsf{end} \cdot t : !S;\mathsf{end} \tag{5.8}$$

ここで，Pass 規則により t が s に置換される場合がありうる．このとき，プロセスの型付けは次の通り変化する．

$$\vdash s?(x);s!\langle x\rangle;\mathbf{0} \triangleright s : ?S;!S;\mathsf{end} \tag{5.9}$$

これは [YV07] により指摘された. この型の変化は，はじめに述べたプロセス (5.6) でそうであったように，デッドロックしているものの通信安全性は保たれており，

(5.9) はエラーではない．しかしながら，このような型の変化はオリジナルのセッション型システムでは許されていない．

例 5.1 P_{e1} を

$$(\nu s)(u!\langle\!\langle s\rangle\!\rangle;\mathbf{0} \mid u?(\!(t)\!);s?(x);t!\langle x\rangle;\mathbf{0}) \tag{5.10}$$

とする．$\vdash P_{e1} \triangleright u\colon \bot$ である一方，P_{e1} を簡約したプロセス

$$(\nu s)s?(x);s!\langle x\rangle;\mathbf{0} \tag{5.11}$$

は型付けできない．部分項 $s?(y);s!\langle y\rangle;\mathbf{0}$ において，s は型 $?S;!S;\mathsf{end}$ を持つが，規則 [CRes] は型 $\bot$ を要求するためである． □

5.2.2　到達されないエラー状態

もうひとつは，(5.6) でみた | の両側のセッション型の不整合に起因する場合である．次の例でその型付けについて詳細に述べる．

例 5.2 P_{e2} を

$$u?(\!(t)\!);s!\langle\mathsf{true}\rangle;\Big(s?(x);\mathbf{0} \,\Big|\, t?(y);t!\langle y\rangle;\mathbf{0}\Big) \mid u!\langle\!\langle s\rangle\!\rangle;\mathbf{0} \tag{5.12}$$

とする．$\vdash P_{e2} \triangleright u : \bot, s : \bot$ である．ここで $P_{e2} \longrightarrow P'_{e2}$ なる 次の P'_{e2} が存在し，

$$s!\langle\mathsf{true}\rangle;\Big(s?(x);\mathbf{0} \,\Big|\, s?(y);s!\langle y\rangle;\mathbf{0}\Big) \tag{5.13}$$

ここで $\nvdash P'_{e2}$ である．なぜなら P'_{e2} の直接の部分項が型付けできない，すなわち $\nvdash s?(x);\mathbf{0} \mid s?(y);s!\langle y\rangle;\mathbf{0}$ であるためである． □

[Conc] 規則はプロセス (5.13) の | の両側のセッション型の不整合を拒否する．しかしながら， (5.13) は $s!\langle\mathsf{true}\rangle;$ でプレフィクスされておりエラーではない．

5.2.3　通信路渡しの型付け規則における問題

３つめの不成立の原因は プロセス $s!\langle\!\langle t\rangle\!\rangle;P$ に型を与える [Thr] 規則の制限である．例えば簡約の結果が $s!\langle\!\langle s\rangle\!\rangle;P$ になるようなプロセスがあり得，これは [Thr] では型付けできない．次の例でプロセス $s!\langle\!\langle s\rangle\!\rangle;P$ を生成する簡約を示す．このプロセス自体はやや込み入っており，直観的な意味や有用な応用があるわけではないが，$s!\langle\!\langle s\rangle\!\rangle;P$ を生成する最小の例である．

$$\dfrac{\dfrac{\dfrac{\vdash u?(x);r!\langle x\rangle;\mathbf{0} \triangleright t : \mathsf{end}\cdot u :?S;\mathsf{end}\cdot r :!S;\mathsf{end}}{\vdash u!\langle\!\langle s\rangle\!\rangle;u?(x);r!\langle x\rangle;\mathbf{0} \triangleright t : \mathsf{end}\cdot s :!S;\mathsf{end}\cdot u :![!S;\mathsf{end}];?S;\mathsf{end}\cdot r :!S;\mathsf{end}}\ (\text{by } \textsc{Thr})}{\vdash s?(\!(r)\!);u!\langle\!\langle s\rangle\!\rangle;u?(x);r!\langle x\rangle;\mathbf{0} \triangleright t : \mathsf{end}\cdot s :?[!S;\mathsf{end}];!S;\mathsf{end}\cdot u :![!S;\mathsf{end}];?S;\mathsf{end}}\ (\text{by } \textsc{Cat})}{\vdash t?(\!(u)\!);s?(\!(r)\!);u!\langle\!\langle s\rangle\!\rangle;u?(x);r!\langle x\rangle;\mathbf{0} \triangleright t :?[![!S;\mathsf{end}];?S;\mathsf{end}];\mathsf{end}\cdot s :?[!S;\mathsf{end}];!S;\mathsf{end}}\ (\text{by } \textsc{Cat})$$

図 5.1: プロセス (5.10) の型付けの導出の一部

例 5.3 P_{e3} を

$$t?(\!(u)\!);s?(\!(r)\!);u!\langle\!\langle s\rangle\!\rangle;u?(x);r!\langle x\rangle;\mathbf{0} \mid t!\langle\!\langle s\rangle\!\rangle;\mathbf{0} \tag{5.14}$$

とする．ここで $\vdash P_{e3} \triangleright s : \bot \cdot t : \bot$ である．実際に (5.14) が片付けできるのは次の理由による．(5.14) における | の左側のプロセスの型環境は

$$t :?[![!S;\mathsf{end}];?S;\mathsf{end}];\mathsf{end}\cdot s :?[!S;\mathsf{end}];!S;\mathsf{end}$$

となる．図 *5.1* に導出の一部を示す．| の右手の型環境は

$$t :![![!S;\mathsf{end}];?S;\mathsf{end}];\mathsf{end}\cdot s :![!S;\mathsf{end}];?S;\mathsf{end}$$

であり，左手と右手の型環境は適合することが確認できる．

P_{e3} は，次のプロセスに簡約される：

$$s?(\!(r)\!);s!\langle\!\langle s\rangle\!\rangle;s?(x);r!\langle x\rangle;\mathbf{0} \tag{5.15}$$

どんな Q においても，$\not\vdash s!\langle\!\langle s\rangle\!\rangle;Q$ であるため，このプロセスは型付けできない．□

(5.15) もやはりデッドロックしている．

ただし，プロセス $s!\langle\!\langle s\rangle\!\rangle;P$ は我々が**自己移譲**と呼ぶ新しい通信パターンを構成できることがある．この点については後の節で議論する．

5.2.4 型付けできない部分への到達不能性

これまでに見た通り，主部簡約を不成立にする $P'_{e1}, P'_{e2}, P'_{e3}$ は実際にはエラーではない．この観測から，[HVK98] は通信安全性を示すのに厳し過ぎる型付け規則を持っているといえる．すべての $\vdash P \wedge P \longrightarrow^* P'$ なる P' はエラーではないことを示したいが，元の型システムでは主部簡約が成立しない．

5.3 型付け規則の再構成

本節で，主部簡約が成立するセッション型システム **R** とその型判定 $\vdash_R$ を確立する．換言すると，**R** ではすべての P について， $\vdash_R P \triangleright \Delta$ ならば， P はエラーではなく，かつすべての P' s.t. $P \longrightarrow^* P'$ について $\vdash_R P' \triangleright \Delta'$ for some Δ' となる．**R** の主部簡約定理は型の変化を認めていることが一つの鍵であるが，前節でみたように型の変化を認めるだけでは十分ではない．

5.3.1 Odd 型の導入と適合性条件の緩和

まず，エラー状態のプロセスに意図的に型を与える型システム **D** (*D*anger) を導入する．セッション型 Odd はその通信路においてエラーが発生する可能性を示している．例えば， (5.13) の部分項 $s?(x);\mathbf{0} \mid s?(y);s!\langle y\rangle;\mathbf{0}$ は **D** のもとで型付け可能であり， s には 型 Odd が割り当てられる．

後の節で，全ての通信路の型が Odd に変化しないという主部簡約定理（定理5.1）を示す．言い換えると，もし if $\vdash_D P \triangleright \Delta$ かつ $\forall s, \Delta(s) \neq \mathtt{Odd}$ ならば， 全ての $P \longrightarrow^* P'$ なる P' について $\vdash_D P' \triangleright \Delta'$ かつ $\forall s.\Delta'(s) \neq \mathtt{Odd}$ が成立する．

定義 5.1 (Odd 型) **D** におけるセッション型は定義 *2.16* から次の通り修正される．

$$T, T', T_1, T_2 \quad ::= \quad \cdots \mid \mathtt{Odd} \tag{5.16}$$

Odd はプレフィクスされた型ではない．さらに，T がプレフィクスされており Odd 型を含まないとき，T はスレッド化しているという．$\overline{\mathtt{Odd}}$ は未定義である． □

セッション型の合成と型環境の適合性は次の通り緩和される：

定義 5.2 (セッション型の合成の拡張) **D** における適合性関係 $\asymp_D$ は，$\Delta_0 \asymp_D \Delta_1$ ならば，かつそのときのみ $\overline{\Delta_0(s)}$ *and* $\overline{\Delta_1(s)}$ *is defined for all* $s \in dom(\Delta_0) \cap dom(\Delta_1)$ となる２項関係である．**D** における型環境の合成 $\circ_D$ は次の等式で定義される．

$$\begin{array}{rcll}
(\Delta_0 \circ_D \Delta_1)(s) & = & \bot & \textit{if} \quad s \in dom(\Delta_0) \cap dom(\Delta_1) \\
 & & & \quad \textbf{and } \Delta_0(s) = \overline{\Delta_1(s)} \\
 & & \mathtt{Odd} & \textit{if} \quad s \in dom(\Delta_0) \cap dom(\Delta_1) \\
 & & & \quad \textbf{and } \Delta_0(s) \neq \overline{\Delta_1(s)} \\
 & & \Delta_0(s) & \textit{if} \quad s \in dom(\Delta_0) \backslash dom(\Delta_1) \\
 & & \Delta_1(s) & \textit{if} \quad s \in dom(\Delta_1) \backslash dom(\Delta_0)
\end{array}$$

□

$$[\text{Ex-Conc}]\ \frac{\Gamma \vdash_D P \triangleright \Delta_1 \qquad \Gamma \vdash_D Q \triangleright \Delta_2 \qquad \Delta_1 \asymp_D \Delta_2}{\Gamma \vdash_D P \mid Q \triangleright \Delta_1 \circ_D \Delta_2}$$

$$[\text{Ex-CRes}]\ \frac{\Gamma \vdash_D P \triangleright \Delta \cdot s : T \qquad T = \bot \vee T \text{ prefixed}}{\Gamma \vdash_D (\nu s)P \triangleright \Delta}$$

$$[\text{Ex-Thr}]\ \frac{\Gamma \vdash_D P \triangleright \Delta \quad \Delta \asymp_D t : T' \quad \Delta \circ_D t : T' = \Delta' \cdot s : T}{\Gamma \vdash_D s!\langle\!\langle t \rangle\!\rangle;P \triangleright \Delta' \cdot s :![T'];T}$$

図 5.2: **D** における型付け規則の置き換え

0dd を導入した意図は，２つの並行な通信路の使用が互いに双対でない箇所に印をつけることである．例えば，$s!\langle\mathsf{true}\rangle;\Big(s?(x);\mathbf{0} \;\Big|\; s?(y);s!\langle y\rangle;\mathbf{0}\Big)$ は s :!bool;0dd のもとで型付けできる．さらに 0dd の双対は定義されていないため，(すなわち，!bool;0dd の双対も定義されないため), このプロセスはそれ以上 s を用いて環境と通信することはできず，エラー状態に到達することはない．

命題 5.1 次の２つが成り立つ

1. すべての T_1 と T_2 について， $T_1 \asymp T_2$ ならば $T_1 \asymp_D T_2$.
2. すべての Δ_1 と Δ_2 について， $\Delta_1 \asymp \Delta_2$ ならば $\Delta_1 \circ \Delta_2 = \Delta_1 \circ_D \Delta_2$.

□

5.3.2 型付け規則の置き換え

新しい型判定 $\Gamma \vdash_D P \triangleright \Delta$ of **D** を導入する．

定義 5.3 (型システム D) 型判定 $\Gamma \vdash_D P \triangleright \Delta$ は図. *2.7* において [Conc], [CRes]，[Thr] を除いた規則と，図 *5.2* の [Ex-Conc], [Ex-CRes]，[Ex-Thr] の規則から帰納的に定義される． □

置き換えられる型付け規則は，5.2 節でみた主部簡約の不成立に関する３つのモードに対応している．

[Ex-Conc] は適合性の条件を $\asymp_D$ に緩和し，通信路の使用に関するエラーを意図的に許容する．もし $(\Delta_1 \circ_D \Delta_2)(s) = \mathsf{0dd}$ ならば， P における s の使用について何

らかのエラーがある可能性を示している. (**R** においては，そのような 0dd は型付け不能として拒否される.) これに加えて，0dd は双対をもたないため，型 0dd を含むセッション型 (like !bool;0dd) をもつ通信路は使われることがない．このことから型が 0dd に変化することはなく，プロセスがエラーに簡約されないことが保証される．

[Ex-CRes] は [CRes] を s がプレフィクスされた型も許容し， (5.13) のようなプロセスも型付けできるように拡張している. 同時に s が 0dd 型を持つことを拒否することで， s におけるエラーの可能性を排除している.

[Ex-Thr] は， [Thr] よりも多様な通信パターンを許容する規則である．前提の $\Delta \circ_D t : T'$ は，

通信路 t を移譲した後には, Δ と並行して t は T' の通りに用いられる

ということを示している．$\circ$ (と， $\circ_D$) は，型環境どうしの並行合成であることを思い出されたい．ここで，[Thr] での $\cdot$ ではなく $\circ_D$ を用いることで， $t \in \mathrm{dom}(\Delta)$ である場合をも許容している．すなわち，$s!\langle\!\langle t \rangle\!\rangle;P$ において P に t が出現する場合を許容する．さらに，$\Delta \circ_D t : T' = \Delta' \cdot s : T$ は

移譲の後，s は どこかで T の通りに用いられる

ということを示している．ここで $s = t$ の場合が網羅されているのは従来の型システムにない新しさであり，プロセス $s!\langle\!\langle s \rangle\!\rangle;P$ に型を与え， s が移譲先で使われることが許容される．結論部の $s :![T'];T$ は，これまでと同じように

型 T' の通信路を s で送信するプロトコルは $![T'];T$ である

ということを示している.

[Ex-Thr] の型付けは，

- $s = t$ であるか，$s \neq t$ か
- $t \in \mathrm{dom}(\Delta)$ か，すなわち送信動作の継続 P が t を使用するか

という２点に分けて理解される．最初に， $s \neq t$ かつ $t \notin \mathrm{dom}(\Delta)$ の場合を考える. この場合において，[Ex-Thr] は [Thr] と等価になる. 定義より $\Delta \circ_D t : T' = \Delta \cdot t : T'$ であるため，規則の前提は $\Delta \cdot t : T' = \Delta' \cdot s : T$ と読み替えられる. $\Delta \cdot t : T = \Delta' \cdot s : T = \Delta'' \cdot s : T \cdot t : T'$ を満たす Δ'' を選べば， [Thr] と等価であることが確認できる.

$$[\textsc{Pseudo-Ex-Thr}_1]\quad \frac{\Gamma \vdash_D P \triangleright \Delta \cdot s : T \cdot t : T_1 \quad t : T_1 \asymp_D t : T' \quad t : T_1 \circ_D t : T' = t : T_2}{\Gamma \vdash_D s!\langle\!\langle t \rangle\!\rangle;P \triangleright \Delta \cdot s :![T'];T \cdot t : T_2}$$

$$[\textsc{Pseudo-Ex-Thr}_2]\quad \frac{\Gamma \vdash_D P \triangleright \Delta \cdot s : T_1 \quad s : T_1 \asymp_D s : T' \quad s : T_1 \circ_D s : T' = s : T}{\Gamma \vdash_D s!\langle\!\langle s \rangle\!\rangle;P \triangleright \Delta \cdot s :![T'];T}$$

$$[\textsc{Pseudo-Ex-Thr}_3]\quad \frac{\Gamma \vdash_D P \triangleright \Delta}{\Gamma \vdash_D s!\langle\!\langle s \rangle\!\rangle;P \triangleright \Delta \cdot s :![T];T}$$

図 5.3: [Ex-Thr] 規則から導出される 3 つの規則

次に，$s \neq t$ かつ $t \in \mathrm{dom}(\Delta)$，すなわち $t \in fc(P)$ で，t が移譲された後も P が t を使い続ける場合を考える. これにより，*bound output* と呼ばれる，$(\nu t)s!\langle\!\langle t \rangle\!\rangle;P$ の形をしたプロセスの型付けが許される. このプロセスは P に局所化されたプライベートな通信路の端点 t を s を介して送信する．この場合と同等な規則を図 5.3 の [Pseudo-Ex-Thr$_1$] 規則に示した. 同等の型付け規則は他のセッション型システムにも見られ [GHVY09, GV10, CV09]，とくに新しいものではない. どのようにして型付けが正当化されるのかを見る．まず $\Delta = \Delta'' \cdot s : T \cdot t : T_1$ とする. 前提より $t : T_1 \asymp_D t : T'$ がいえ，よってある T_2 が存在し $\Delta \circ_D t : T' = \Delta'' \cdot s : T \cdot t : T_1 \circ t : T' = \Delta'' \cdot s : T \cdot t : T_2$ であり，これより $\Delta \circ_D t : T_1 = \Delta' \cdot s : T$ から $\Delta' = \Delta'' \cdot t : T_2$ が得られる. T_1 と T' がお互いに双対であるかどうかに依存して，T_2 は $\bot$ か $\mathtt{0dd}$ のどちらかになる．もし $\bot$ ならば，t を用いた通信にエラーがないことを期待できる．このとき t を受け取ったプロセスは 型 T' で示される通りに t を使い，同時に P は型 $\overline{T'}$ で示される通りに t を使うことで，通信が続行される.

$s = t$ の場合は $s \in \mathrm{dom}(\Delta)$ か否かに分けて考えられる. もし $s \in \mathrm{dom}(\Delta)$ であるとき，プロセスはデッドロックしていることが導かれる．図 5.3 の 規則 [Pseudo-Ex-Thr$_2$] は [Ex-Thr] において $s = t$ かつ $s \in \mathrm{dom}(\Delta)$ とした場合に得られる規則である. ここで $T = \mathtt{0dd}$ か $T = \bot$ の両方の場合があるが，どちらの場合でも s の型はプレフィクスされており，さらに s を用いて通信するプロセスが環境には存在しえないため，プレフィクスは簡約され得ず，デッドロックしていることがわかる.

最後に残ったのは，従来のセッション型システムにはない興味深い型付けである．[Ex-Thr] において $s = t$ かつ $s \notin \mathrm{dom}(\Delta)$ を仮定する，すなわち P が s をこれ以上使用しないと仮定すると，図 5.3 の規則 [Pseudo-Ex-Thr$_3$] が得られる. 規則

[Pseudo-Ex-Thr$_2$] と異なり，この場合に型付けされるプロセスはデッドロックされておらず s を介して環境と相互作用できる. 型 $![T];T$ は，「型 T の通信路を移譲した後, s は 型 T の通りに使われる」と述べている．ここで P は s で通信しないものの, s の受信者が型 T の通りに s を使うため，このような型付けが成立する．この型付けは次のような通信パターンを正当化する.

$$\vdash_D s!\langle\!\langle s\rangle\!\rangle;P \mid s?(\!(t)\!);Q \triangleright \Delta \cdot s : \bot$$

ここで $s \notin fc(P)$ である. $s!\langle\!\langle s\rangle\!\rangle;P$ は s の残りの能力を s それ自体を介して送る. さらに，ある Δ' について $\vdash_D Q \triangleright \Delta' \cdot s : \overline{T} \cdot t : T$ であるため，簡約により生成されるプロセス $Q\{s/t\}$ が s で通信し続けることが期待される. 次の簡約はその一例である.

$$\begin{array}{ll} & s!\langle\!\langle s\rangle\!\rangle;\mathbf{0} \mid s?(\!(t)\!);\Big(s!\langle \mathsf{true}\rangle;\mathbf{0} \,\Big|\, t?(x);\mathbf{0}\Big) \\ \longrightarrow & s!\langle \mathsf{true}\rangle;\mathbf{0} \mid s?(x);\mathbf{0} \\ \longrightarrow & \mathbf{0} \end{array} \tag{5.17}$$

この型付けは，i/o 型や [PS96] や線形型 [KPT99] のような型システムでは型を付けられなかった $s!\langle\!\langle s\rangle\!\rangle;P$ の形のプロセスに適切な型を与える点で新規性がある. Milner のソート [Mil93] によれば適切に正当化されるが，ソートそれ自体はセッションのような柔軟さをもたない.

定義 5.4 (セッション型システム R) 型判定 $\vdash_R$ は 3 項関係 $\Gamma \vdash_R P \triangleright \Delta$ であり，$\Gamma \vdash_D P \triangleright \Delta$ かつ $\forall s, \Delta(s) \neq \mathsf{0dd}$ と論理的に同値である.

R は 0dd 型環境に 0dd を禁止することで，エラーを排除している.

5.4　主部簡約と型安全

5.4.1　基本的性質

最初に $\vdash_D$ と $\vdash_R$ の基本的な補題を示す.

補題 5.1 (弱化) $\Gamma \vdash_D P \triangleright \Delta$ とする. 次が成り立つ：

1. $x \notin fv(P)$ ならば $\Gamma \cdot x : U \vdash_D P \triangleright \Delta$.
2. $s \notin fc(P)$ ならば $\Gamma \vdash_D P \triangleright \Delta \cdot s : T$ かつ $T = \mathsf{end}$ または $T = \bot$.

同様のことは $\vdash_R$ でも成立する.
Proof: 型判定の導出の構造に関する帰納法による. *2.* については, 基底は *[Inact]* であり $T = \bot$ ならば規則 [Bot] を導出木の途中で使用する. □

補題 5.2 (強化) $\Gamma \vdash_D P \triangleright \Delta$ とする.

- $x \notin fv(P)$ ならば $\Gamma\backslash x \vdash_D P \triangleright \Delta$.
- $s \notin fc(P)$ ならば $\Gamma \vdash_D P \triangleright \Delta\backslash s$.

同様のことは $\vdash_R$ でも成立する.
Proof: 型判定の導出の構造に関する帰納法による. □

型システムの安全性の証明においては, よく逆補題 (inversion lemma) と呼ばれる, 型付け可能な項の直接の部分項の型環境を得るための補題が用いられる. セッション型システムにおいては, 構文のみでは型環境の形は決まらない. これは [End] 規則により導入された型 end をもつ fresh な名前が, [Bot] 規則により型 $\bot$ を割り当てることができるためである. [Bot] 規則の使用による型 $\bot$ の導入を, 半順序 $\prec$ により定式化する.

定義 5.5 ([YV07] より) $\prec$ は, $\Delta \prec \Delta'$ ならば $\Delta \cdot s : \mathsf{end} \prec \Delta' \cdot s : \bot$ を満たす半順序である. □

補題 5.3 (逆補題) 次が成立する:

1. $\Gamma \vdash_D P_1|P_2 \triangleright \Delta$ ならば, ある $\Delta_1, \Delta_2, \Delta_1', \Delta_2'$, Δ' が存在し $\Gamma \vdash_D P_1 \triangleright \Delta_1$, $\Gamma \vdash_D P_2 \triangleright \Delta_2$, $\Delta_i \prec \Delta_i'$ $(i \in \{1,2\})$, $\Delta_1' \circ_D \Delta_2' = \Delta'$, $\Delta' \prec \Delta$.
2. $\Gamma \vdash_D (\nu s)P \triangleright \Delta$ ならばある Δ', T *such that* $\Delta' \prec \Delta$ が存在し $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T$.

同様のことは $\vdash_R$ でも成立する.
Proof: *(1)* 導出木 $\Gamma \vdash_D P_1|P_2 \triangleright \Delta$ の部分木は有限個の [Bot] に続いた [Conc] のノードを含む. それゆえ, [Bot] の子をたどることで $\Gamma \vdash_D P_1|P_2 \triangleright \Delta'$ かつ $\Delta' \prec \Delta$ なる Δ' が得られ, さらに, [Conc] の子をたどると, ある Δ_1, Δ_2 が存在し $\Delta' = \Delta_1 \circ_D \Delta_2$ かつ $\Gamma \vdash_D P_1 \triangleright \Delta_1$ と $\Gamma \vdash_D P_2 \triangleright \Delta_2$ が得られる. $\vdash_R$ についても, $\forall s.(\Delta_1 \circ_D \Delta_2)(s) \neq \mathsf{0dd}$ ならば $\Delta_i(s) \neq \mathsf{0dd}$ *for* $i \in \{1,2\}$ であるため, 同様に成り立つ. *(2)* 規則 [Ex-Cres] により自明. □

さらに，後のために [Ex-Thr] の逆補題を証明する．これは [Ex-Thr] が様々な場合分けを含む洗練された型付け規則であるがゆえである．直観的には，前節でみた [Thr], [Pseudo-Ex-Thr$_1$], [Pseudo-Ex-Thr$_2$], [Pseudo-Ex-Thr$_3$] の４つに分類できる．

補題 5.4 (逆補題２) 次が成立する：

1. $\Gamma \vdash_D s!\langle\!\langle t \rangle\!\rangle;P \triangleright \Delta \cdot s : T_s \cdot t : T_t$ かつ T_s と T_t の両方がスレッド化しているならば, あるスレッド化した T'_s, $T_s = ![T_t];T'_s$ かつ，ある $\Delta' \prec \Delta$ について $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T'_s$.

2. $\Gamma \vdash_D s!\langle\!\langle t \rangle\!\rangle;P \triangleright \Delta \cdot s : T_s \cdot t : T'$ かつ T_s がスレッド化しているが T' はスレッド化していないならば，ある T'_s について $T_s = ![T_t];T'_s$ であり，ここで T'_s, T_t はスレッド化している．さらに，ある $\Delta' \prec \Delta$ とスレッド化した T'_t について $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T'_s \cdot t : T'_t$ である . さらに $T' = \bot$ ならば $T'_t = \overline{T_t}$.

3. $\Gamma \vdash_D s!\langle\!\langle s \rangle\!\rangle;P \triangleright \Delta \cdot s : T_s$ かつ T_s はスレッド化していないならば，あるスレッド化した T_1 とスレッド化していない T' について $T_s = ![T_1];T'$ かつある $\Delta' \prec \Delta$ とスレッド化した T_2 について $\Gamma \vdash_D P \triangleright \Delta' \cdot T_2$ である．

4. $\Gamma \vdash_D s!\langle\!\langle s \rangle\!\rangle;P \triangleright \Delta \cdot s : T_s$ かつ T_s がスレッド化しているならばあるスレッド化した T'_s が存在し $T_s = ![T'_s];T'_s$ かつ，ある $\Delta' \prec \Delta$ について $\Gamma \vdash_D P \triangleright \Delta'$.

同様のことが $\vdash_R$ でも成立する．

Proof: 導出木を根からたどると [Ex-Thr] のノードが存在し，ある $\Delta' \prec \Delta$ について，*1,2* については $\Gamma \vdash_D s!\langle\!\langle t \rangle\!\rangle;P \triangleright \Delta' \cdot s : T_s \cdot t : T_t$ が得られ，*3* については $\Gamma \vdash_D s!\langle\!\langle s \rangle\!\rangle;P \triangleright \Delta' \cdot s : T_s$ が得られる. *(1)* [Ex-Thr] の前提より, ある Δ_0, T'_t について $\Gamma \vdash_D P \triangleright \Delta_0$ かつ $\Delta_0 \circ_D t : T'_t = \Delta' \cdot t : T_t \cdot s : T'_s$. T_t はスレッド化されているので, $\circ_D$ の定義より $T_t = T'_t$ かつ $\Delta_0 = \Delta' \cdot s : T'_s$. *(2)* [Ex-Thr] の前提より, ある Δ_0 について $\Gamma \vdash_D P \triangleright \Delta_0$ かつ $\Delta_0 \circ_D t : T_t = \Delta' \cdot t : T' \cdot s : T'_s$. $\circ_D$ の定義より, $\Delta_0(t) = \overline{T_t}$ かつ $\Delta_0 = \Delta' \cdot s : T'_s \cdot t : T'_t$. *(3)* [Ex-Thr] の前提より, ある Δ_0, T' について $\Gamma \vdash_D P \triangleright \Delta_0$ かつ $\Delta_0 \circ_D s : T'_1 = \Delta' \cdot s : T'$. $\circ_D$ の定義より, あるスレッド化された T_2 について $\Delta_0 = \Delta' \cdot s : T_2$. *(4)* [Ex-Thr] の前提より, ある Δ_0, T' について $\Gamma \vdash_D P \triangleright \Delta_0$ かつ $\Delta_0 \circ_D s : T'_s = \Delta' \cdot s : T'$. T_s はスレッド化しているため, $\circ_D$ の定義より, T'_s はスレッド化しており, ゆえに $T_s = ![T'];T'_s$ かつ $T' = T'_s$. □

型は構造合同性によるプロセスの変形について保存される．

補題 5.5 (主部合同) *If* $\Gamma \vdash_R P \triangleright \Delta$ *and* $P \equiv Q$, *then* $\Gamma \vdash_R Q \triangleright \Delta$.
Proof: 証明は *[YV07]* に類似する．部分項の型付けについては補題 *5.3* を用いる． □

次の命題は， **R** が [HVK98] のセッション型システムの conservative extension であることを示す．

命題 5.2 $\Gamma \vdash P \triangleright \Delta$ ならば $\Gamma \vdash_R P \triangleright \Delta$.
Proof: 型判定の導出木の構造に関する帰納法による．図 *5.2* による型付け規則の置き換えは $\vdash$ の型付けを変更しない． □

R ではより多くのプロセスに型を与えるが，エラーは排除できている．

補題 5.6 (エラーの排除) $\Gamma \vdash_R P \triangleright \Delta$, ならば P はエラーではない.
Proof: P はエラーである，すなわち，ある s-プロセス Q_1 と Q_2 が存在し $Q_1|Q_2$ が s-基でなく， $P \equiv (\nu \tilde{s})((Q_1|Q_2)|R)$ ． 補題 *5.5* より，ある Γ, Δ' が存在し $\Gamma \vdash (\nu \tilde{s})((Q_1|Q_2)|R) \triangleright \Delta'$. 逆補題 *(*補題 *5.3)* より，ある Δ_1 と Δ_2 が存在し $\Gamma \vdash_R Q_1|Q_2 \triangleright \Delta_1 \circ_D \Delta_2$ かつ $i \in \{1,2\}$ について $\Gamma \vdash_R Q_i \triangleright \Delta_i$ が成り立つ．Q_i は s-プロセスであるが $Q_1|Q_2$ は s-基ではないため, $(\Delta_1 \circ_D \Delta_2)(s) = \mathtt{Odd}$ であり， $\mathtt{Odd} \in cod(\Delta)$. である．これは $\Gamma \vdash_R P \triangleright \Delta$ に矛盾する． □

5.4.2 主部簡約と型安全性

本章の主要な成果である主部簡約定理と型安全性定理を述べ，これらを証明する．証明においては，代入が関わる簡約について微妙な扱いを要求される．そこでまず，通信路の代入について型付けが保存されることを述べる 3 つの補題を証明する．

名前の代入は Pass 規則 （図 2.2）によってのみ起こる．Pass による簡約は，基底においては $s!\langle\!\langle t \rangle\!\rangle;P \,|\, s?(\!(u)\!);Q \longrightarrow P \,|\, Q\{t/u\}$ の形である．それゆえ，次の 2 通りのみを考えればよい： (i) $t \notin fc(Q)$ であるか， (ii) Q における t の型が u の型と双対である，言い換えるとある Δ, T, T' について $\vdash_R Q \triangleright \Delta \cdot t : T \cdot u : T'$ であり，さらに $T' = \overline{T}$ である．次の補題は (i) について証明する．

補題 5.7 (代入補題 1) 次のことが成立する：

1. $\Gamma \vdash_R P \triangleright \Delta$ かつ $s \notin dom(\Delta)$ ならば，$\Gamma \vdash_R P\{t/s\} \triangleright \Delta$.

2. $\Gamma \vdash_R P \triangleright \Delta \cdot s : T_s$ かつ $t \notin dom(\Delta)$ ならば $\Gamma \vdash_R P\{t/s\} \triangleright \Delta \cdot t : T_s$.

同様のことが $\vdash_D$ でも成り立つ.

Proof: 前者は自明である．後者は，$\vdash_D$ の導出木の構造に関する帰納法による. □

(ii) については，単純な帰納法では証明できない．証明したいのは $\vdash_R P \triangleright \Delta \cdot s : T \cdot t : \overline{T}$ ならばある Δ' が存在し $\vdash_R P\{s/t\} \triangleright \Delta'$ かつ $\forall s.\Delta'(s) \neq \mathtt{0dd}$ が成立することである. しかしながら， P が s でプレフィクスされたプロセスである場合には, T の型が変わるため帰納法の仮定が使えない. たとえば， $\vdash_R s!\langle e\rangle;P_1 \triangleright \Delta \cdot s :!S;T \cdot t :?S;\overline{T}$ において，部分項の型付けは $\vdash_R P' \triangleright \Delta \cdot s : T \cdot t :?S;\overline{T}$ であるが， s と t の型が双対でない，つまり $\overline{T} \neq ?S;\overline{T}$ であるため帰納法の仮定が成り立たない. そのような場合のため，まずは代入により型がエラーになるような **D** に関する代入補題を証明する.

補題 5.8 (D の代入補題) $\Gamma \vdash_D P \triangleright \Delta \cdot s : T_s \cdot t : T_t$ かつ T_s, T_t がスレッド化されているならば，ある T' が存在し $\Gamma \vdash_D P\{s/t\} \triangleright \Delta \cdot s : T'$.

Proof: 証明は型判定の導出木の構造に関する帰納法による．ここで T' は $\mathtt{0dd}$ でもあり得ることに注意せよ. P が並行合成かプレフィクスの場合のみを考える. 他の場合は自明である.

$P = P_1|P_2$ のとき，２つに場合分けできる．どちらも逆補題を使う. *(P1)* $i = 1$ か $i = 2$ の一方について $\Gamma \vdash_D P_i \triangleright \Delta_i \cdot s : T_s \cdot t : T_t$ の場合．このとき，帰納法の仮定が使える．*(P2)* ある Δ' について $\Delta_1 \circ_D \Delta_2 = \Delta'$ *and* $\Delta' \prec \Delta$ であり，$\Gamma \vdash_D P_1 \triangleright \Delta_1 \cdot s : T_s$ ， $\Gamma \vdash_D P_2 \triangleright \Delta_2 \cdot t : T_t$ のとき．補題 *5.7* より， $\Gamma \vdash_D P_1\{s/t\} \triangleright \Delta_1 \cdot s : T_s$ かつ $\Gamma \vdash_D P_2\{s/t\} \triangleright \Delta_2 \cdot s : T_t$ である. T_s と T_t はスレッド化しているため，これらの双対も定義されており，ある T' について $\Gamma \vdash_D (P_1|P_2)\{s/t\} \triangleright \Delta \cdot s : T'$ である．ここで $T_s = \overline{T_t}$ か否かによって $T' = \bot$ か $T' = \mathtt{0dd}$ のどちらか一方が成り立つ.

プレフィクスされたプロセスについては，ほとんどの場合，型付け規則の前提から得られる型判定に帰納法の仮定を使い，さらにもう一度型付け規則を適用することで，T' が得られる．しかしながら，[Ex-Thr] において $P = u!\langle\!\langle u'\rangle\!\rangle;P'$ の場合については微妙な扱いが必要である．ここでは $u, u' \in \{s, t\}$ の場合のみを考える. 他の場合は自明である.

(T1) $P = s!\langle\!\langle t\rangle\!\rangle;P'$ のとき．補題 *5.4* より，ある $T_s = ![T_t];T'_s$ なる T'_s について $\Gamma \vdash_D P' \triangleright \Delta \cdot s : T'_s$. 補題 *5.7* をさらに適用して $\Gamma \vdash_D P'\{s/t\} \triangleright \Delta \cdot s : T'_s$ が成り立ちさらに [Ex-Thr] を適用することで，ある $s : T'_s \circ_D s : T_1 = s : T_2$ なる T_1, T_2 について $\Gamma \vdash_D s!\langle\!\langle s\rangle\!\rangle;P'\{s/t\} \triangleright \Delta \cdot s :![T_1];T_2$ が成り立つ. *(T2)* $P = t!\langle\!\langle s\rangle\!\rangle;P'$ *(T1)* と同様.

(T3) $P = s!\langle\!\langle s \rangle\!\rangle;P'$ のとき. 逆補題 *5.4* より $\Gamma \vdash_D P' \triangleright \Delta \cdot t : T_t$ を得る. 代入補題 *5.7* より $\Gamma \vdash_D P'\{s/t\} \triangleright \Delta \cdot s : T_t$. 規則 [Ex-Thr] を適用すれば，ある $T_1 \circ_D T_t = T_2$ なる T_1, T_2 について $\Gamma \vdash_D P'\{s/t\} \triangleright \Delta \cdot s :![T_1];T_2$. *(T4)* $P = t!\langle\!\langle t \rangle\!\rangle;P'$ *(T3)* と同様. □

補題 5.8 は， $T_s \neq \overline{T_t}$ ならば $P\{s/t\}$ はエラーであり得ることを示している. しかしながら，型のついた安全な通信では $T_s = \overline{T_t}$ であるため， 次の補題で示されるように P はエラーではない.

補題 5.8 を使って，**R** のための代入補題を証明する.

補題 5.9 (R の代入補題) $\Gamma \vdash_R P \triangleright \Delta \cdot s : T \cdot t : \overline{T}$ ならば $\Gamma \vdash_R P\{s/t\} \triangleright \Delta \cdot s : T'$ *for some* T'.

Proof: 型判定の導出木の構造に関する帰納法による. 重要なのは P が s か t でプレフィクスされている場合であり，このとき帰納法の仮定の代わりに補題 *5.8* を用いる. $P = s!\langle \mathsf{true} \rangle;P'$ の場合について証明を概観する. [Send] の前提より，$\Gamma \vdash_R P \triangleright \Delta \cdot s : T' \cdot t : \overline{T}$ である. ここで T がスレッド化されているため，T' もスレッド化されている. 補題 *5.8* を用いると，ある T'' について $\Gamma \vdash_D P \triangleright \Delta \cdot s : T''$ が得られる. 再度 [Send] を適用すれば $\Gamma \vdash_R s!\langle \mathsf{true} \rangle;P' \triangleright \Delta \cdot s :!\mathsf{bool};T''$ が得られる. □

オリジナルのセッション型システム [HVK98] における代入補題の証明では，, 代入補題における $\vdash P\{s/t\}$ の型付けは $s \notin fc(P)$ のみしか考慮されなかった. 一方，補題 5.9 は $s \in fc(P)$ のときも $\vdash_R P\{s/t\}$ が成立することを述べており，これが主部簡約定理の証明に重要な役割を果たす. $\vdash_R$ の代入補題の系として， $\vdash$ の代入補題が得られる.

系 5.1 ($\vdash$ の代入補題) $\Gamma \vdash P \triangleright \Delta \cdot s : T \cdot t : \overline{T}$ ならば $P\{s/t\}$ はエラーではない.

Proof: 補題 *5.6* と 補題 *5.9* より明らか. □

このように，代入により 型 $\mathsf{0dd}$ が現れることがあるものの，常にそれはプレフィクスされている. 主部簡約定理を証明する前に，通信が発生する通信路の型を調べる補題を証明しておく.

補題 5.10 (s-基の型) $\Gamma \vdash_R P \triangleright \Delta$ かつ P が s-基ならば， $\Delta(s) = \bot$.

Proof: $\Delta(s)$ がプレフィクスされているならば, P の型判定の導出木の構造に関する帰納法により，P は s-基ではないことが示され，矛盾. さらに，型判定 $\vdash_R$ の定義より $\forall t, \Delta(t) \neq \mathsf{0dd}$ であるため， $\Delta(s) = \bot$. □

本章の主たる定理である，**R** の主部簡約と型安全性は次の通り確立される.

定理 5.1 (主部簡約) ある t について $\Gamma \vdash_R P \triangleright \Delta \cdot t : T_t$ かつ $P \longrightarrow P'$ ならば $\Gamma \vdash_R P' \triangleright \Delta \cdot t : T'_t$. さらに T_t がプレフィクスされているならば, $T_t = T'_t$.

Proof:　$P \longrightarrow P'$ の導出木の構造に関する帰納法による. 代入が発生し型の変化を伴う $s!\langle\!\langle t \rangle\!\rangle;P_1 \mid s?(\!(u)\!);P_2 \longrightarrow P_1 \mid P_2\{t/u\}$ のみを考える. 他の場合は型の変化を伴わず $T_t = T'_t$ である.

($t \in fc(P_2)$) $T_t = \bot$ である．逆補題 *(*補題 *5.3* と 補題 *5.4* の *1)* と， [Cat] の前提より，型判定の導出木は次のノードをもつ*:*

$$\frac{\Gamma \vdash_R P_1 \triangleright \Delta'_1 \cdot s : T_s}{\Gamma \vdash_R s!\langle\!\langle t \rangle\!\rangle;P_1 \triangleright \Delta'_1 \cdot s :![T_1];T_s \cdot t : T_1} \tag{5.18}$$

と

$$\frac{\Gamma \vdash_R P_2 \triangleright \Delta'_2 \cdot s : \overline{T_s} \cdot u : T_1 \cdot t : \overline{T_1}}{\Gamma \vdash_R s?(\!(u)\!);P_2 \triangleright \Delta'_2 \cdot s :?[T_1];\overline{T_s} \cdot t : \overline{T_1}} \tag{5.19}$$

ここで $\Delta'_1 \circ_D \Delta'_2 = \Delta'$ かつ $\Delta' \prec \Delta$ である. **R** の代入補題 *(*補題 *5.9)* より, ある T' について $\Gamma \vdash_R P_2\{t/u\} \triangleright \Delta'_2 \cdot s : \bot \cdot t : T'$. [Conc] を適用し, $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta' \cdot s : \bot \cdot t : T'$ が得られる. さらに [Bot] を適用すれば $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta \cdot s : \bot \cdot t : T'$ が得られる.

($s = t$ の場合) 補題 *5.10,* より，$\Gamma \vdash_R s!\langle\!\langle s \rangle\!\rangle;P_1 \mid s?(\!(u)\!);P_2 \triangleright \Delta \cdot s : \bot$ が得られる. 逆補題 *(*補題 *5.3* と 補題 *5.4* の *4)*, 規則 [Cat] の前提より, 型判定の導出木は次のノードを含む：

$$\frac{\Gamma \vdash_R P_1 \triangleright \Delta'_1}{\Gamma \vdash_R s!\langle\!\langle s \rangle\!\rangle;P_1 \triangleright \Delta'_1 \cdot s :![T];T} \tag{5.20}$$

と

$$\frac{\Gamma \vdash_R P_2 \triangleright \Delta'_2 \cdot s : \overline{T} \cdot u : T}{\Gamma \vdash_R s?(\!(u)\!);P_2 \triangleright \Delta'_2 \cdot s :?[T];\overline{T}} \tag{5.21}$$

ここで $\Delta'_1 \circ_D \Delta'_2 = \Delta'$ かつ $\Delta' \prec \Delta$ である．**R** の代入補題 *(*補題 *5.9)* より，$\Gamma \vdash_R P_2\{s/u\} \triangleright \Delta'_2 \cdot s : T'$ *for some* T'. 規則 [Conc] より $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta' \cdot s : T'$ となる. さらに 規則 [Bot] より， $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta \cdot s : T'$ が得られる.

($s \neq t$ の場合) 補題 *5.10* より，ある T_t について $\Gamma \vdash_R s!\langle\!\langle t \rangle\!\rangle;P_1 \mid s?(\!(u)\!);P_2 \triangleright \Delta \cdot s : \bot \cdot t : T_t$. この場合は複数のサブケースに分けられる.

*(*サブケース $t \notin fc(P_2)$*)* $t \notin fc(P_1)$ ならば， 最初の代入補題 *(*補題 *5.7)* を用いれば明らかに $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta \cdot s : \bot \cdot t : T_t$ である. $t \in fc(P_1)$ の場合を考える. 逆転

補題 *(*補題 *5.3* と 補題 *5.4* の *2)* と，規則 [Cat] の前提より，型判定の導出木は次のノードを含む：

$$\frac{\Gamma \vdash_R P_1 \triangleright \Delta_1' \cdot s : T_s \cdot t : \overline{T_1}}{\Gamma \vdash_R s!\langle\!\langle t \rangle\!\rangle; P_1 \triangleright \Delta_1' \cdot s :![T_1];T_s \cdot t : \bot} \tag{5.22}$$

と

$$\frac{\Gamma \vdash_R P_2 \triangleright \Delta_2' \cdot s : \overline{T_s} \cdot u : T_1}{\Gamma \vdash_R s?(\!(u)\!); P_2 \triangleright \Delta_2' \cdot s :?[T_1];\overline{\overline{T_s}}} \tag{5.23}$$

ここで $\Delta_1' \circ_D \Delta_2' = \Delta'$ かつ $\Delta' \prec \Delta$ である．最初の代入補題 *(*補題 *5.7)* より，$\Gamma \vdash_R P_2\{t/u\} \triangleright \Delta_2' \cdot s : \bot \cdot t : T_1$ が成り立つ．規則 [Conc], より，$\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta' \cdot s : \bot \cdot t : \bot$ が得られる．さらに，規則 [Bot] を適用すれば $\Gamma \vdash_R P_1 \mid P_2\{t/u\} \triangleright \Delta \cdot s : \bot \cdot t : \bot$ が得られる． □

型安全性定理は，補題 5.1 と定理 5.1 より直ちに導かれる．

定理 5.2 (型安全性) ある Δ が存在して $\vdash_R P \triangleright \Delta$ ならば, P はエラーではない．
Proof: 補題 *5.6* と 定理 *5.1* より明らか． □

系として，オリジナルのセッション型システム [HVK98] においても通信安全性が成立することを示す．

系 5.2 ([HVK98] の型安全性) $\vdash P$ ならば $P \longrightarrow P'$ なるすべての P' について，P' はエラーではない．
Proof: 命題 *5.2* と 定理 *5.1* より示される． □

5.5 おわりに

本章では主部簡約性をもつセッション型システム **R** を確立した．鍵となるアイデアは簡約の途中で型の変化を認めることであった．この型システムにおいて，主部簡約性とは「安全でない型」0dd のプレフィクスが外れて露出することがないことであった. 代入補題は率直な帰納法では証明できず，２つの補題に分けて証明された．このことは [HVK98] において主部簡約が成立しなかったことと対応する．**R** を用いて，オリジナルのセッション型システムの型安全性を示せた．

新規性の高い型付け規則 [Ex-Thr]（図 5.2）は，従来のセッション型システムより多くのプロセスを一貫性のあるかたちで型付けする．この型付け規則は 5.3.2 節

でみた自己移譲 $s!\langle\!\langle s \rangle\!\rangle;P$ に適切な型を与える点で新しい．我々の知る限り，このようなプロセスに型を与えられるセッション型システムはこれまでに提案されていない．我々はこの型付け規則を，4章で示した Haskell におけるセッション型実装に導入した．我々はこの型付け規則がセッション型システムをより柔軟なものにする．

R の型付け規則は，オリジナルのセッション型システム [HVK98] では排除できていた，望ましくないプロセスに型を与えることがある．この違いは，オリジナルのセッション型システムは「プロセスが動作する前に」安全性を型検査する，通常の意味での型システムであるのに対し，我々の型システムは「プロセスが動作しているあいだの」性質を述べるための拡張であることに由来する，**R** で導入された全ての型付け規則は，プロセスが動作しているあいだの安全性を調べるために導入された．一方，たとえばプログラムのコンパイル時の型検査に用いるには，次のようにしなければならない．(i) 型 Odd の出現は拒否されるべきである．換言すると，$\circ_D$ と $\asymp_D$ の代わりに $\circ$ and $\asymp$ を使わなければならない．(ii) 規則 [Ex-CRes] と [Ex-Conc] の代わりに [CRes] と [Conc] をそれぞれ使わなければならない．この条件のもとでも, **R** は 規則 [Ex-Thr] のおかげでより多くのプロセスを型付けできる．このような，コンパイル時と実行時で異なる型付け規則を用いるやり方は Featherweight Java [IPW01] における *stupid cast* にもみられる．Stupid cast は Java の主部簡約性を証明するために導入されたが，エラーを発生させる実行時キャストを型付けしてしまうためコンパイラの型検査器には導入されていない．

Giunti らの研究との関連 近年，Giunti らは線形型によく似た方法でセッション型システムを定式化した [GV10]．この型システムでは，並行に使われる通信路の両端点の型をセッション型の対 (T, T') で追跡する．（我々の型システムにおいては，そのような通信路には型 $\bot$ か Odd を与えていた．）この型の対の一方に対してプレフィクスする特別な型付け規則を導入することで，我々の型システムでは Odd を含むような簡約されたプロセスにおいても，対が「バランスした」形を保つようになっている．我々の記法を用いてこれを説明する．例えば，プロセス (5.11) $s!\langle\mathsf{true}\rangle;\Big(s?(x);\mathbf{0} \,\Big|\, s?(y);s!\langle y\rangle;\mathbf{0}\Big)$ は，[GV10] の規則 [T-OutC] を用いて $s : (?S_2;\mathsf{end}, ?S_1;!S_2;\mathsf{end})$ の左手に $!S_1;$をプレフィクスすることにより型の対を用いて $s : (!S_1;?S_2;\mathsf{end}, ?S_1;!S_2;\mathsf{end})$ と型付けされる．

Giunti らの [GV10] に対して，我々の型システムがもつ利点は次の通りである．(i) 我々の型システムはプロセスの構文ひとつに対して型付け規則がただ1つ存在

しするが，[GV10] では入力と出力プレフィクスについてそれぞれ２つの型付け規則, [T-In] と [T-InC], [T-Out] と [T-OutC] を必要とする．このことは 4 章でみた Haskell における実装で用いられるような単純な単一化ベースの型推論を困難にするだろう．(ii) 我々の型システムでは通信路渡しで発生する不整合について明示的に型 odd で印を付ける．[GV10] では，そのような不整合は導出木における型付け規則 [T-InC] と [T-OutC] のノードとして現れることになる．

さらに，我々が提案する [Ex-Thr] は便利なプロセスに型を与えられる．Giunti らの型システムでも $s!\langle\!\langle s \rangle\!\rangle;P$ の形のプロセスに型を与えるが，それはセッションを伴わない（同種のメッセージのみを通信する能力をもつ）通信路の送信である．我々の型付け規則は s がセッションを伴う場合についても,「通信能力の残りを通信路自身から送る」という型付けにより与えている．この型付けは [T-Out] を変更することで、Giunti らの型システムにも導入することができる．[GV10] の [T-Out] 規則は次のようになっている：

$$\frac{\Gamma_1 \vdash v : T \qquad \Gamma_2, x : S \vdash P}{\Gamma_1 \cdot (\Gamma_2, x : \mathsf{lin!}T.S) \vdash \overline{x}\langle v \rangle.P}$$

[Ex-Thr] と同様に，[T-Out] は次のように置き換えられる：

$$\frac{\Gamma_1 \vdash v : T \qquad \Gamma_2 \vdash P \qquad \Gamma_1 \cdot \Gamma_2 = \Gamma_3, x : S}{\Gamma_3, x : \mathsf{lin!}T.S \vdash \overline{x}\langle v \rangle.P}$$

この型付け規則によれば，$\Gamma, \mathsf{lin!}S.S \vdash \overline{x}\langle x \rangle.P$ という型判定が $x = v$ と $T = S$ のもとで合法になる．

第6章　結論

6.1　本論文のまとめ

分散システムの開発においては，通信記述を解析できる計算モデルに基づきソフトウェアを構成することが望ましい．さらに，既存の柔軟かつ信頼性の高い静的検査をもつ既存のプログラミング言語を利用できれば，利便性と通信の信頼性を両立した枠組みを構築できる．そこで我々はプロセス計算の一種であるπ計算を基礎とした２種類の型システムに基づく通信記述の枠組みをそれぞれ提案した．まず，通信の非同期性と計算資源の局所性に着目し，非同期局所化π計算（ALπ）を基礎とした，Haskell のモナドを用いた通信記述の枠組みを与えた．次に，通信プロトコルを表現できるセッション型を Haskell に導入した．さらに，セッション型システムの主部簡約性を定式化し，型安全性を証明した．

ALπ は，分散システムにおける非同期性と局所性を捉えたπ計算の部分体系である．この性質を利用し，π計算の名前渡しによる表現能力を備えつつ，軽量な実装をもつ通信記述の枠組みを提案した．このフレームワークは，実装においてπ計算のラベル付き遷移系を忠実にエンコードしている．そのため，ネットワークプログラムの解析にπ計算の解析技法がそのまま扱えるのが利点である．それまでのπ計算に基づくプログラミングの枠組みはプログラミング言語の構文などを一から設計したものである一方で，本手法においては Haskell の抽象化能力と静的型付けを活用し，実装を軽量なものとしたことは新規な点である．Haskell へのエンコーディングにおいては，まず既存の Haskell プログラミングの枠組みであるモナドの形で，非同期π計算の意味を忠実に実装する方法を議論した．さらに，型により通信の局所性を表現するために重要な役割を果たすサブタイプ関係を，Haskell の型クラスによりエンコードした．非同期局所π計算はπ計算に制限を加えた体系であるが，名前渡しの機能により分散システムの通信記述に十分な能力を備えている．これを示すため，インスタントメッセンジャーの実装例を挙げた．

セッション型は通信手順を表現できる型システムであり，２つのプロセス間の異種のメッセージ通信の系列が安全に進行することを型検査により静的に判定でき

る．従来からセッション型を Haskell に導入する様々な枠組みが提案されてきたが，複数の通信路を同時に使用する枠組みは十分に整備されていなかった．我々は de Bruijn レベルを用いて名前を型レベルにエンコードすることで，セッション型のプログラミングにおいて型注釈の必要がない枠組みを構築した．この利点を示すため，SMTP プロトコルで型付けされた通信路と，メールの情報を伝達する通信路の２つを用いたプログラミングの例を示した．Pucella と Tov の実装と比較して，スタック操作が必要なく，より見通しがよい枠組みであることを示した．この方法により，セッション型や線形型のような，文脈に依存して通信能力が変わる多様な型システムを Haskell に埋め込めるようになる．

本田らのオリジナルのセッション型システムは π 計算の動作意味で扱うときに主部簡約性が成立しないことが過去に指摘されていた．主部簡約性が成立しない型システムでは型安全性も成立せず，型システムとして不適当である．我々はセッション型システムに修正を加え，主部簡約を定式化し証明した．従来のセッション型システムでは，プロセスの簡約により，ある名前において通信が整合せず型付けできない項が生成されることがあった．そのような名前に `0dd` 型という通信の不整合を表す型を割り当てた．ここで `0dd` 型は将来の不整合の可能性を示す型であるものの，簡約によっては到達されないことを主部簡約として定式化し，これを証明した．さらに，$s!\langle s\rangle; P$ の形で，通信路の能力をそれ自身を介して送信するという，より自由なセッション型の型付け規則を発見した．5 章のまとめに記したように，この型付けは他のセッション型システムにも適用可能である．

分散システムの開発において，本論文で提案した手法により Haskell を用いた分散プログラムの通信記述を π 計算の技法を用いて解析できるようになる．特に，セッション型が Haskell に統合されたことにより，プログラムの通信記述が全体として整合したものとなる．本手法により，分散システムの正しさを保証でき，信頼性が向上することが期待できる．

6.2　関連研究

6.2.1　並行計算の関数型プログラミングへの応用

Nepi[KMK01, MMK+05] は Lisp 処理系上に実装された，同期 π 計算に基づくプログラミング言語である．関数型言語上に構築された点において本研究との類似がある．しかしながら Lisp は型システムを提供しないため，Nepi には型システム

の概念がない．一方，Nepi は同期通信や非決定的選択プリミティブを提供し，より制御構造を見通し良く記述できるとしている．

並行モナドを構成する手法は [Cla99] でも述べられているが，通信プリミティブを導入していない．3 章と 4 章の手法では並行プリミティブに非同期 π 計算のものを導入し，さらにチャネル型による安全な通信が期待できる．実装の手法としては，状態モナドを組み合わせることで名前生成を可能にしている．

6.2.2 極性とセッション型システム

吉田と Vasconcelos は [YV07] で極性をもつセッション型システムを提案した. そのセッション型システムは通信路の両端点を構文的に異なるかたちで表現する．極性による注釈をつけることで,「転送者」プロセス

$$fwd?(t);t?(x:\mathsf{int});s!\langle x\rangle;\mathbf{0} \tag{6.1}$$

は次の通りになる：

$$fwd^{+}?(t);t?(x:\mathsf{int});s^{+}!\langle x\rangle;\mathbf{0} \tag{6.2}$$

すべての通信路 s は極性の注釈がつき s^{+} や s^{-} と表記される. s^{+} とは反対側の端点である s^{-} を受信した場合， このプロセスは次の通りになる：

$$s^{-}?(x:\mathsf{int});s^{+}!\langle x\rangle;\mathbf{0} \tag{6.3}$$

極性をもたない π 計算と異なり, 両端点に異なる型を付与できるため，型が変化するという問題は起こらない．

Giunti ら [GHVY09] はセッション型システムの型安全性の別な証明を与えている．この証明は *double binder language* とよばれる別の計算体系への変換を介して与えられている．その証明は，変換の健全性と変換後のシステムにおける型安全性を必要とするために，やや複雑なものとなっている．

6.3 今後の課題

今後の課題は次の通りである．

計算体系と実装の対応関係　型付き ALπ のラベル付き遷移関係と，3 章で示した PiMonad の実行の間には，直観的には双模倣の対応があると予想されるが，形式的な証明を与えられていない．そのような証明を与えることは今後の課題である．さらに，異常終了や通信の切断等のにおける振る舞いは ALπ では定式化されておらず，そのような場合の動作は実装依存になっている．π 計算に例外処理を加えた体系は Carbone らによって提案されており [CYH09]，そのような体系に基づいた実装を与えることは興味深い方向である．

de Bruijn レベルの技法を用いた多様な型システムの実装　de Bruijn レベルを使った技法は線形型 [KPT99] や multiparty のセッション型 [HYC08] などの他の π 計算の型システムの埋め込みにも使えると予想される．特に，multiparty のセッション型のエンコーディングは有望である．multiparty のセッション型とは，通信路に３つ以上の端点をもつ計算体系におけるセッション型である．[HYC08] における端点のセッション型は [HVK98] とよく似ており，我々の手法が効果的に適用できる. ただし，適用には非同期性に起因する微妙な扱いが必要になると予想される．multiparty のセッション型においては，通信は非同期的になるため，あるセッション型 $k\langle U\rangle; k'\langle U'\rangle; T$ について，$k'\langle U'\rangle; k\langle U\rangle; T$ などと順序が入れ替わった型も等価とみなす場合がある．Haskell の型推論は単一化による進行するため，型に複数の表現を持たせられず，何らかの一意な表現が必要になる．ここで de Bruijn エンコーディングが重要な役割を果たす可能性がある. この k はスコープをもつ通信路を表すため，de Bruijn レベルにより自然数を割り当てることができ, その数の順で並べ替えることにより一意な型表現を得ることができる．

依存型言語へのセッション型システムの埋め込み　Dependent ML[Xi07] に代表される依存型言語はプログラムの性質に関する証明を型で表現可能なプログラミング言語である．このようなプログラミング言語において，4 章で述べたセッション型の埋め込み技法がどのようにエンコードできるのかは興味深い課題である．しかしながら，注意すべきは Haskell などで使われている型レベルプログラミングは型推論を利用した計算であるという点である．型推論は，実行時のデータの流れによらず項の構造のみで型を計算する枠組みであり，単一化を利用して計算の向きをある程度まで自由に与えることができる．とくに関数型プログラミング言語の Hindley-Milner 型推論 [DM82] は，ある項の型を決めるのに部分項だけでなく文脈の型も利用できるため，Haskell の型レベルプログラミングにおいては通常の計

算とは逆向きに進行する計算を利用できる．依存型言語は λ 計算の β 簡約により型レベル計算が進行するため，単一化のような双方向性を利用できず，いくつかの技法は使えないおそれがある．

より柔軟なセッション型システムの模索　セッション型は平易な型付け規則により通信プロトコルを表現できる点で型システムとして良い性質を備えている．今後の課題の 1 つは，より多くのプロセスに型を与える柔軟なセッション型システムを構築することである．そのヒントは [Ex-Thr] 規則にあると考えられる．例えば，

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s : \bot}{\Gamma \vdash s!\langle e \rangle; P \triangleright \Delta \cdot s : ![S]; \mathsf{End}} \tag{6.4}$$

という型付け規則は，通信の一方の端点 s における通信路の再利用を可能にする．通信が完了したのち（この場合，何かの値が通信路に送られたあと），その通信路をふただび並列に動作するプロセス間の通信に使用できるようになる．そのような型のクラスは五十嵐と小林による Generic Type System [IK04] の文脈で調べられるだろう．

謝辞

本研究を進めるにあたり，多岐に渡って熱心な御指導御鞭撻を頂いた名古屋大学大学院情報科学研究科 結縁祥治教授 ならびに 同 阿草清滋教授に心より感謝致します．本論文の構成および内容に関して有益な御意見を頂いた名古屋大学大学院情報科学研究科 坂部俊樹教授に深く感謝致します．結縁祥治教授には本論文の執筆以外にも研究の様々な局面において数々の適切な御助言を頂いたことに深く感謝致します．

有限会社 IT プランニング 小笠原啓常務 ならびに 同 今井宜洋氏には，日頃から多大なるご支援を頂き，また，業務が多忙にもかかわらず研究の遂行にご理解を示して下さったことに深く感謝致します．業務と並行して研究を遂行することを認めて下さいました，有限会社 IT プランニング 大嶽成治社長 ならびに 同 大嶽政彦専務に深く感謝致します．

また，著者が名古屋大学に在籍している期間，熱心に議論し，時には精神的にご支援を頂いた愛知県立大学情報科学部 山本晋一郎教授，名古屋大学大学院情報科学研究科 小林隆志准教授, 同 濱口毅助教，同 渥美紀寿研究員， 立命館大学 情報理工学部 情報システム学科 桑原寛明講師をはじめ，名古屋大学阿草・結縁研究室の関係者諸氏，同期，先輩，後輩，および，名古屋大学情報科学研究科附属組込みシステム研究センターの皆様に感謝致します．

陽に陰に日々の生活を通して著者を支えてくれた妻 友里恵に心より感謝致します．最後に，長期にわたり暖かく見守って下さった両親に感謝致します．

参考文献

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In *Theorem Proving in Higher Order Logics*, Vol. 3603 of *Lecture Notes in Computer Science*, pp. 50–65. Springer-Verlag, 2005.

[BHA+04] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. Student paper: HaskellDB improved. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pp. 108–115. ACM, 2004.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, Vol. 195, No. 2, pp. 205–226, 1998.

[Bou92] Gerard Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, Sophia-Antipolis, 1992.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[CDCY07] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object Oriented Languages. In *Formal Methods for Open Object-Based Distributed Systems*, Vol. 4468 of *Lecture Notes in Computer Science*, pp. 1–31. Springer, 2007.

[CdY08] Mariangiola D. Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In Gilles Barthe and Cédric Fournet, editors, *TGC'07 : the 3rd conference on Trustworthy global com-*

puting, Vol. 4912 of *Lecture Notes in Computer Science*, pp. 257–275. Springer, 2008.

[Cla99] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, Vol. 9, No. 3, pp. 313–323, May 1999.

[CV09] Luís Caires and Hugo T. Vieira. Conversation Types. In *ESOP 2009: 18th European Symposium on Programming*, Vol. 5502 of *Lecture Notes in Computer Science*, pp. 285–300. Springer, 2009.

[CYH09] Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous session types: Exceptions and multiparty interactions. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services*, Vol. 5569 of *Lecture Notes in Computer Science*, chapter 5, pp. 187–212. Springer, 2009.

[DB72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, Vol. 75, No. 5, pp. 381–392, 1972.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, New York, NY, USA, 1982. ACM.

[DO02] Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *Journal of Functional Programming*, Vol. 12, No. 02, pp. 133–158, 2002.

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pp. 263–273. ACM, 1997.

[EJ09] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, Vol. 42, No. 4, pp. 42–52, 2009.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[GH05] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, Vol. 42, No. 2, pp. 191–225, 2005.

[ghc] The Glasgow Haskell Compiler . Available from `http://www.haskell.org/ghc/`.

[GHVY09] Marco Giunti, Kohei Honda, Vasco T. Vasconcelos, and Nobuko Yoshida. Session-Based Type Discipline for Pi Calculus with Matching. In *PLACES '09: the preproceedings of Programming Language Approaches to Concurrency and Communication-cEntric Software*, March 2009. Available at `http://places09.di.fc.ul.pt/`.

[GV10] Marco Giunti and Vasco T. Vasconcelos. A Linear Account of Session Types in the Pi Calculus. In Paul Gastin and François Laroussinie, editors, *CONCUR'10: 21st international conference on Concurrency theory*, Vol. 6269 of *Lecture Notes in Computer Science*, pp. 432–446. Springer, 2010.

[Has10] *Haskell 2010 Language Report*, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, Vol. 512 of *Lecture Notes in Computer Science*, pp. 133–147. Springer, 1991.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In *Object-Based Concurrent Computing*, Vol. 612 of *Lecture Notes in Computer Science*, pp. 21–51. Springer, 1992.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the 7th European Symposium on Pro-*

gramming, Vol. 1381 of *Lecture Notes in Computer Science*, pp. 122–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *SIGPLAN Notices*, Vol. 43, No. 1, pp. 273–284, 2008.

[IK04] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *Theoretical Computer Science*, Vol. 311, No. 1-3, pp. 121–163, 2004.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 3, pp. 396–450, 2001.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp. 295–308. ACM, 1996.

[Jon00] Mark P. Jones. Type Classes with Functional Dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pp. 230–244. Springer, 2000.

[Kle08] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pp. 96–107. ACM, 2004.

[KMK01] Yoshinobu Kawabe, Ken Mano, and Kiyoshi Kogure. The Nepi 2 programming system: A π-calculus-based approach to agent-based programming. In *FAABS 2000: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems*, Vol. 1871 of *Lecture Notes in Computer Science*, pp. 90–102, 2001.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, pp. 914–947, 1999.

[KS08] Oleg Kiselyov and Chung C. Shan. Lightweight Monadic Regions. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pp. 1–12. ACM, 2008.

[KSS00] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An Implicitly-Typed Deadlock-Free process calculus. In Catuscia Palamidessi, editor, *CONCUR 2000: 11th international conference on Concurrency theory*, Vol. 1877 of *Lecture Notes in Computer Science*, pp. 489–504. Springer, 2000.

[Lan05] Saunders Mac Lane. 圏論の基礎. シュプリンガー・フェアラーク東京, 2005.

[Ler11] Xavier Leroy. The OCaml system release 3.12 Documentation and user's manual, 2011. With Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Departement of Computer Science, Universiteit Utrecht, 2001.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[Mcb02] Conor Mcbride. Faking it : Simulating Dependent Types in haskell. *Journal of Functional Programming*, Vol. 12, No. 5, pp. 375–392, July 2002.

[Mer00] Massimo Merro. *Locality in the π-calculus and Applications to Object-Oriented Languages*. PhD thesis, Ecole des Mines de Paris, 2000.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, Vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, Vol. 2, No. 2, pp. 119–141, 1992.

[Mil93] Robin Milner. *The polyadic π-calculus: a tutorial*. Springer, 1993. Available at `http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/`.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[MMK+05] Atsushi Mizuno, Ken Mano, Yoshinobu Kawabe, Hiroaki Kuwabara, Kiyoshi Agusa, and Shoji Yuen. Name-passing style GUI programming in the π-calculus-based language Nepi. *ENTCS*, Vol. 139, No. 1, pp. 145–168, 2005.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, Vol. 93, pp. 55–92, 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, Vol. 100, pp. 1–77, 1992.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. 1997.

[Nes00] U. Nestmann. What is a "Good" Encoding of Guarded Choice? *Information and Computation*, Vol. 156, No. 1-2, pp. 287–319, 2000.

[NP00] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Information and Computation*, Vol. 163, No. 1, pp. 1–59, 2000.

[NT04] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL'04 : Practical Aspects of Declarative Languages*, Vol. 3057 of *Lecture Notes in Computer Science*, pp. 56–70. Springer, 2004.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 1st edition, 2008.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical Structures in Computer Science*, Vol. 13, No. 5, pp. 685–719, 2003.

[PE88] Frank Pfenning and Conal Elliot. Higher-Order Abstract Syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pp. 199–208. ACM, 1988.

[PH06] Frédéric Peschanski and Samuel Hym. A stackless runtime environment for a Pi-calculus. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pp. 57–67. ACM, 2006.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, Vol. 6, No. 5, pp. 409–454, 1996.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-calculus. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell Session Types with (Almost) No Class. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*. ACM, 2008.

[San96] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, Vol. 33, pp. 69–97, 1996.

[SE08] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, Imperial College London, June 2008. Available at `http://pubs.doc.ic.ac.uk/session-types-in-haskell/`.

[SJCS08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type Checking with Open Type Functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pp. 51–62. ACM, 2008.

[SPJM97] Mark Jones Simon Peyton Jones and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Second Haskell Workshop*, June 1997.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[SWU10] Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. Nomadic Pict: Programming Languages, Communication Infrastructure Overlays, and Semantics for Mobile Computation. *ACM Transactions on Programming Languages and Systems*, Vol. 32, No. 4, pp. 12:1–12:63, 2010.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94 - Parallel Architectures and Languages Europe*, Vol. 817 of *Lecture Notes in Computer Science*, pp. 398–413. Springer, 1994.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp. 1–14. ACM, 1992.

[Wal95] David Walker. Objects in the π-Calculus. *Information and Computation*, Vol. 116, No. 2, pp. 253–271, 1995.

[Xi] Hongwei Xi. The ATS Programming Language. Available from `http://www.ats-lang.org/`.

[Xi07] Hongwei Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, Vol. 17, pp. 215–286, 2007.

[Yos02] Nobuko Yoshida. Minimality and Separation Results on Asynchronous Mobile Processes - Representability Theorems by Concurrent Combinators. *Theoretical Computer Science*, Vol. 274, No. 1-2, pp. 231–276, 2002.

[YV07] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. In *SecReT 2006:*

Proceedings of the First International Workshop on Security and Rewriting Techniques, Vol. 171 of *Electronic Notes in Theoretical Computer Science*, pp. 73–93. Elsevier, 2007.

[河辺 02] 河辺義信, 真野健, 堀田英一, 小暮潔. π-計算の名前制限の名前生成による実装の正しさ. 電子情報通信学会論文誌 D, Vol. Vol.J85-D1, No. 3, pp. 249–261, 2002.

発表文献

論文誌

[IYA06] 今井 敬吾, 結縁 祥治, 阿草 清滋. Haskell のための非同期局所化 π 計算に基づくネットワークプログラミングフレームワーク. 情報処理学会論文誌：プログラミング, Vol.47, No. SIG 16 (PRO 31), pp.10-28 (2006)

[IYA11] Keigo Imai, Shoji Yuen, Kiyoshi Agusa. Session Type Inference in Haskell. Electronic Proceedings in Theoretical Computer Science, Vol. 69, pp. 74-91 (2011)

国際会議

[IYA10a] Keigo Imai, Shoji Yuen, Kiyoshi Agusa. Session Type Inference in Haskell. Preproceedings of Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES), pp. 43-52, Paphos, Cyprus, March, 2010. （この論文は [IYA11] の会議録版である。会議の後さらに改訂したものを投稿し、査読を経た後に [IYA11] が採録された。）

研究会/シンポジウム

[IYA07] 今井 敬吾, 結縁 祥治, 阿草 清滋. 高信頼なネットワークアプリケーションのための Haskell 言語による型付き通信ライブラリ. 第 5 回ディペンダブルシステムワークショップ, 北海道函館市 (2007 年 7 月)

[IYA08] 今井 敬吾, 結縁 祥治, 阿草 清滋. セッション型に基づく高信頼ネットワークプログラムの関数型言語による実装手法. 情報処理学会論文誌：プログラミング, Vol.49, No. SIG 3 (PRO 36), p.57 (2008)

[IYA09] Keigo Imai, Shoji Yuen, Kiyoshi Agusa. A full implementation of Session Types in Haskell. 第 11 回プログラミングおよびプログラミング言語ワークショップ論文集 (PPL2009), pp. 44-56 (2009)

[IYA10b] 今井 敬吾, 結縁 祥治, 阿草 清滋. 極性をもたないセッション型システム. 第 12 回プログラミングおよびプログラミング言語ワークショップ論文集 (PPL2010), pp. 16-30 (2010)

付 録A　Haskell型レベルプログラミングによるセッション型推論の実装

第 4 章の手法で実装したセッション型推論の実装 `full-sessions` を示す．これらのソースコードは， The Glasgow Haskell Compiler 7.0.3 [ghc] でコンパイルできることを確認済みである．最新のソースコードは `http://hackage.haskell.org/package/full-sessions/` から取得できる．

モジュールの構成　実装の全体は モジュール `Control.FullSession` に統合される．その他のモジュールの役割は次の通りである．

FullSession.Base 実装の全体にわたって使用する型レベルプログラミングの基礎的な処理を提供する．型レベルにおける整数の加減算と，型レベルのリストを含む．

FullSession.TypeEq 型変数どうしの構文的な比較を提供する．再帰型の folding/unfolding でのみ扱い，他では使用しない．

FullSession.Types セッション型および通信路型を提供する．

FullSession.TypeAlgebra 型の双対性を提供する．

FullSession.Ended 終了した型環境に関する操作を提供する．

FullSession.Recursion 再帰型の folding/unfolding 操作を提供する．

FullSession.SMonad `Session` モナドを提供する．

FullSession.FullSession セッション型付きの π 計算に基づく通信プリミティブを定義する．

FullSession.NwSession ネットワーク通信に関するプリミティブを定義する.

FullSession.Incoherent 関数依存を用いて宣言した型クラスについて，incoherent なインスタンスをまとめる.

FullSession.DeferredInstances ライブラリをコンパイルする時の型クラスの展開を抑制するため，一部の型クラスのインスタンス宣言をこのモジュールに置いている.

smtp.hs 4.4 節で示した SMTP クライアントの実装の全体.

ソースコード A.1: モジュール Control.FullSession

```
1  -- | Pi-calculus style communication and concurrency primitives which come with session types.
2  module Control.Concurrent.FullSession (
3  -- * Type level construcs
4  -- ** Type level numbers and booleans
5      Z, S, P, Nat, T, F,
6  -- * Session types (protocol types)
7      Send, Recv, Throw, Catch, Select, Offer, End, Bot, Rec, Var, Close, SelectN, OfferN,
8  -- * Session type environments
9      (:>), Nil,
10 -- * The Session monad
11     Session, (>>>=), (>>>), ireturn, runS,
12 -- * Communication and concurrency primitives
13 -- ** Channel types
14     Channel, Service,
15 -- ** General communication
16     close, send, recv, sendS, recvS, sel1, sel2, ifSelect, offer, new, newService, connect, connectRunS,
17     accept, acceptRunS,
18 -- ** Network communication
19 -- *** Primitives
20     connectNw, connectNw2, acceptOneNw2, sel1N, sel2N, ifSelectN, offerN,
21     dualNw, dualNw2, mkNwService, mkNwService2,
22 -- *** Type class for messages
23     Message (parseMessage, showMessage),
24 -- ** Thread creation
25     forkIOs, forkOSs,
26 -- ** Interfacing with the IO monad
27     io, io_,
28 -- ** Exception handling
29     finallys,
30 -- ** Recursive protocol support
31     unwind0, unwind1, unwind2, recur1, recur2,
32 -- * Utility functions for type inferene
33     channeltype1, channeltype2, typecheck1, typecheck2,
34 -- * Type classes for type-level operations
35 -- ** Type level arithmetics and boolean operators
36     EqNat, Sub, SubT, And,
37 -- ** Operations on type level lists
38     SList, Pickup, PickupR, Update, UpdateR,
39 -- ** Type classes for ended type environments (1)
40     Ended, IsEnded, IsEndedST,
41 -- ** Duality of session types
42     Dual,
43 -- ** Parallel composition of session types
```

```
    Comp, Par, Par',
-- ** Type classes for ended type environments (2)
    EndedWithout, EndedWithout', EndedWithout2, EndedWithout2', AppendEnd, AppendEnd', Diff, Diff',
-- ** Restrictions on session types for network communication
    NwService, NwService2, NwSender, NwReceiver, NwSession, NwDual, NwSendOnly, NwReceiveOnly,
-- ** Recursive protocol
    RecFold, RecFoldCont, RecFold2, RecFoldCont2, RecUnfold, RecUnfoldCont,
-- ** Type equality
    TypeEq, TypeEq', TypeEq'',
  ) where

import FullSession.Base
import FullSession.TypeEq
import FullSession.Types
import FullSession.TypeAlgebra
import FullSession.Ended
import FullSession.Recursion
import FullSession.SMonad
import FullSession.FullSession
import FullSession.NwSession
import FullSession.Incoherent
import FullSession.DeferredInstances
```

ソースコード A.2: モジュール FullSession.Base

```
{-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeFamilies #-}

module FullSession.Base where

-- ** Type level numbers

-- | Type level zero.
data Z = Z deriving Show
-- | Type level successor. @'S' n@ denotes @(n+1)@.
data S n = S n deriving Show
-- | Type level predecessor (only for internal use). @'P' n@ denotes @(n-1)@.
data P n = P n

-- | The class which covers type-level natural numbers.
class Nat n where
  nat :: n
instance Nat Z where
  nat = Z
instance Nat n => Nat (S n) where
  nat = S nat

-- | Type level @True@.
data T = T
-- | Type level @False@.
data F = F

-- | type-level '&&'
class And b1 b2 b | b1 b2 -> b


-- | Equality on type-level natural numbers.  @b ~ 'T'@ if @x == y@. Otherwise @b ~ F@.
class EqNat x y b | x y -> b
instance EqNat Z Z T
instance EqNat (S n) Z F
```

```
38 instance EqNat Z (S n) F
39 instance EqNat n n' b => EqNat (S n) (S n') b
40
41 -- | Computes subtraction of @n@ by @n'@
42 class Sub n n' where
43   sub :: n -> n' -> SubT n n'
44   sub_ :: Either n n' -> SubT n n'
45 instance Sub n n where { sub _ _ = Z; sub_ _ = Z }
46 instance Sub (S n) n where { sub _ _ = S Z; sub_ _ = S Z }
47 instance Sub (S (S n)) n where { sub _ _ = S (S Z); sub_ _ = S (S Z) }
48 instance Sub (S (S (S n))) n where { sub _ _ = S (S (S Z)); sub_ _ = S (S (S Z)) }
49 instance Sub (S (S (S (S n)))) n where { sub _ _ = S (S (S (S Z))); sub_ _ = S (S (S (S Z))) }
50 instance Sub (S (S (S (S (S n))))) n where
51   sub _ _ = S (S (S (S (S Z)))); sub_ _ = S (S (S (S (S Z))))
52 instance Sub (S (S (S (S (S (S n)))))) n where
53   sub _ _ = S (S (S (S (S (S Z))))); sub_ _ = S (S (S (S (S (S Z)))))
54 instance Sub (S (S (S (S (S (S (S n))))))) n where
55   sub _ _ = S (S (S (S (S (S (S Z)))))); sub_ _ = S (S (S (S (S (S (S Z))))))
56 instance Sub (S (S (S (S (S (S (S (S n)))))))) n where
57   sub _ _ = S (S (S (S (S (S (S (S Z)))))); sub_ _ = S (S (S (S (S (S (S (S Z))))))
58 instance Sub (S (S (S (S (S (S (S (S (S n))))))))) n where
59   sub _ _ = S (S (S (S (S (S (S (S (S Z))))))); sub_ _ = S (S (S (S (S (S (S (S (S Z)))))))
60 instance Sub n (S n) where { sub _ _ = P Z; sub_ _ = P Z }
61 instance Sub n (S (S n)) where { sub _ _ = P (P Z); sub_ _ = P (P Z) }
62 instance Sub n (S (S (S n))) where { sub _ _ = P (P (P Z)); sub_ _ = P (P (P Z)) }
63 instance Sub n (S (S (S (S n)))) where { sub _ _ = P (P (P (P Z))); sub_ _ = P (P (P (P Z))) }
64 instance Sub n (S (S (S (S (S n))))) where
65   sub _ _ = P (P (P (P (P Z)))); sub_ _ = P (P (P (P (P Z))))
66 instance Sub n (S (S (S (S (S (S n)))))) where
67   sub _ _ = P (P (P (P (P (P Z))))); sub_ _ = P (P (P (P (P (P Z)))))
68 instance Sub n (S (S (S (S (S (S (S n))))))) where
69   sub _ _ = P (P (P (P (P (P (P Z)))))); sub_ _ = P (P (P (P (P (P (P Z))))))
70 instance Sub n (S (S (S (S (S (S (S (S n)))))))) where
71   sub _ _ = P (P (P (P (P (P (P (P Z)))))); sub_ _ = P (P (P (P (P (P (P (P Z))))))))
72 instance Sub n (S (S (S (S (S (S (S (S (S n))))))))) where
73   sub _ _ = P (P (P (P (P (P (P (P (P Z)))))); sub_ _ = P (P (P (P (P (P (P (P (P Z))))))))
74
75 -- | Computes subtraction of @n@ by @n'@
76 type family SubT n n' :: *
77 type instance SubT n n = Z
78 type instance SubT (S n) n = S Z
79 type instance SubT (S (S n)) n = S (S Z)
80 type instance SubT (S (S (S n))) n = S (S (S Z))
81 type instance SubT (S (S (S (S n)))) n = S (S (S (S Z)))
82 type instance SubT (S (S (S (S (S n))))) n = S (S (S (S (S Z))))
83 type instance SubT (S (S (S (S (S (S n)))))) n = S (S (S (S (S (S Z)))))
84 type instance SubT (S (S (S (S (S (S (S n))))))) n = S (S (S (S (S (S (S Z))))))
85 type instance SubT (S (S (S (S (S (S (S (S n)))))))) n = S (S (S (S (S (S (S (S Z)))))))
86 type instance SubT (S (S (S (S (S (S (S (S (S n))))))))) n = S (S (S (S (S (S (S (S (S Z))))))))
87 type instance SubT n (S n) = P Z
88 type instance SubT n (S (S n)) = P (P Z)
89 type instance SubT n (S (S (S n))) = P (P (P Z))
90 type instance SubT n (S (S (S (S n)))) = P (P (P (P Z)))
91 type instance SubT n (S (S (S (S (S n))))) = P (P (P (P (P Z))))
92 type instance SubT n (S (S (S (S (S (S n)))))) = P (P (P (P (P (P Z)))))
93 type instance SubT n (S (S (S (S (S (S (S n))))))) = P (P (P (P (P (P (P Z))))))
94 type instance SubT n (S (S (S (S (S (S (S (S n)))))))) = P (P (P (P (P (P (P (P Z)))))))
95 type instance SubT n (S (S (S (S (S (S (S (S (S n))))))))) = P (P (P (P (P (P (P (P (P Z))))))))
96
97 -- | Type-level snoc (reversed version of cons @(:)@). @ss :> s@ denotes a list @ss@ with @s@ on its end.
98 data ss :> s = ss :> s
99 -- | Type-level empty list (@[]@).
100 data Nil = Nil
```

```
101
102
103 -- ** Operations on type level lists
104
105
106 -- | The class which covers session-type environments. The second parameter of the class denotes the
107 --   length of the list.
108 class SList ss l | ss -> l where
109   len_ :: ss -> l
110 instance SList ss l => SList (ss :> s) (S l) where
111   len_ ~(ss :> s) = S (len_ ss)
112 instance SList Nil Z where
113   len_ _ = Z
114
115 -- | @'Pickup' ss n s@ denotes that the @n@-th element of the list @ss@ is @s@.
116 --   This type class plays an important role in session-type inference.
117 --
118 --   Formally, @'Pickup' ss n s@ if @s = pickup ss n@ where @pickup@ is:
119 --
120 --   @
121 --   pickup ss n = pickupR ss (len ss - (n+1))
122 --     where pickupR (ss:>s) Z     = s
123 --           pickupR (ss:>s) (S n) = pickupR ss n
124 --           len Nil     = 0
125 --           len (ss:>s) = (len ss) + 1
126 --   @
127 --
128 --   For example, @Pickup (End :> Bot :> Send Int End) Z t)@ is an instance of @Pickup@, and @t@ is
129 --   unified with @Bot@.
130 --
131 --   Note that the list counts from left to right.
132 --   For example, The @0@-th element of the list @((Nil :> End) :> Bot) :> Send Int End@ is @End@.
133 --
134 --   Usually the list is accessed from the right end.
135 --   The context
136 --
137 --   @
138 --     'SList' ss (S n), 'Pickup' (ss:>Bot:>Recv Char End) n s
139 --   @
140 --
141 --   is expanded into
142 --
143 --   @
144 --     'SList' ss (S n), 'PickupR' (ss:>Bot:>Recv Char End) ('SubT' (S n) (S n)) s, 'Sub' (S n) (S n)
145 --   @
146 --
147 --   since @'SubT' ('S' n) ('S' n) ~ Z@,  it will be reduced to
148 --
149 --   @
150 --     'PickupR' (ss:>Bot:>Recv Char End) Z s
151 --   @
152 --
153 --   and then @s@ is unified with @Recv Char End@.
154 class -- (SList ss l, Sub l (S n), PickupR ss (SubT l (S n)) s) =>
155       Pickup ss n s | ss n -> s where
156   pickup :: ss -> n -> s
157
158 -- | The reversed version of 'Pickup' which accesses lists in reversed order (counts from right to left).
159 --   I.e., @'PickupR' (End :> Bot :> Send Int End) Z (Send Int End)@ is an instance of 'PickupR'.
160 class PickupR ss n s | ss n -> s where
161   pickupR :: ss -> n -> s
162 instance ss ~ (ss':>t) => PickupR ss Z t where
163   pickupR (_ :> t) _ = t
```

```
instance (PickupR ss' n t, ss ~ (ss':>s')) => PickupR ss (S n) t where
  pickupR (ss' :> _) (S n) = pickupR ss' n

-- | @'Update' ss n t ss'@ denotes that @ss'@ is same as @ss@ except that its @n@-th element is @t@.
--   Formally, @'Update' ss n t ss'@ if @ss' = update ss n t@ where @update@ is:
--
-- @
--   update ss n t = updateR ss (len ss - (n+1)) t
--     where updateR (ss:>_) Z     t = ss :> t
--           updateR (ss:>s) (S n) t = updateR ss n t :> s
--           len Nil     = 0
--           len (ss:>s) = (len ss) + 1
-- @
--
--   In other words, @'Update' (End :> Bot :> Send Int End) Z End (End :> Bot :> End))@ is an instance
--   of @Update@.
--
--   Note that the list counts from left to right, as in the case of @Pickup@.
--
class -- (SList ss l, Sub l (S n), UpdateR ss (SubT l (S n)) t ss') =>
    Update ss n t ss' | ss n t -> ss' where
  update :: ss -> n -> t -> ss'

-- | The reversed version of 'Update'.
class UpdateR ss n t ss' | ss n t -> ss' where
  updateR ::  ss -> n -> t -> ss'
instance UpdateR (ss:>s) Z t (ss:>t) where
  updateR (ss:>_) _ t = ss :> t
instance UpdateR ss n t ss' => UpdateR (ss:>s) (S n) t (ss':>s) where
  updateR (ss:>s) (S n) t = updateR ss n t :> s
```

ソースコード A.3: モジュール FullSession.TypeEq

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE OverlappingInstances #-}

module FullSession.TypeEq where

import FullSession.Base

class TypeEq' () x y b => TypeEq x y b | x y -> b where
  type'eq :: x -> y -> b
  type'eq _ _ = undefined::b
class TypeEq' q x y b | q x y -> b
class TypeEq'' q x y b | q x y -> b
instance TypeEq' () x y b => TypeEq x y b
instance b ~ T => TypeEq' () x x b -- redundunt
instance TypeEq'' q x y b => TypeEq' q x y b
instance EqNat x y b => TypeEq'' () x y b
```

ソースコード A.4: モジュール FullSession.Types

```
{-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE Rank2Types #-}

```

```
7  module FullSession.Types where
8
9  import FullSession.Base
10
11
12
13
14
15 -- | @‘Send‘ v u@ denotes a protocol to emit a value of type @v@ followed by a behavior of type @u@.
16 -- Use of @‘send‘@ on a channel changes its session type from @‘Send‘ v u@ into @u@.
17 data Send v u = Send (v -> IO ()) u
18 -- | @‘Recv‘ v u@ denotes a protocol of receiving a value of type @v@ followed by a behavior of type @u@.
19 -- Use of @‘recv‘@ on a channel changes its session type from @‘Recv‘ v u@ into @u@.
20 data Recv v u = Recv (IO v) u
21 -- | @‘Throw‘ u1 u2@ denotes a behavior to output of a channel with session type @u1@ followed by a
22 --   behavior of type @u2@.
23 -- Use of @‘sendS‘@ on a channel changes its session type from @‘Throw‘ u1 u2@ into @u2@.
24 data Throw u' u = Throw (u' -> IO ()) u
25 -- | @‘Catch‘ u1 u2@ is the input of a channel with session type @u1@ followed by a behavior of type @u2@.
26 -- Use of @‘recvS‘@ on a channel changes its session type from @‘Catch‘ u1 u2@ into @u2@.
27 data Catch u' u = Catch (IO u') u
28 -- | @‘Select‘ u1 u2@ denotes to be either behavior of type @u1@ or type @u2@ after emitting a
29 --   corresponding label @1@ or @2@.
30 --   Use of @‘sel1‘@ or @‘sel2‘@ on a channel changes its session type from @‘Select‘ u1 u2@ into @u1@
31 --   or @u2@, respectively.
32 data Select u1 u2 = Select (Bool -> IO ()) u1 u2
33 -- | @‘Offer‘ u1 u2@ denotes a behavior like either @u1@ or @u2@ according to the incoming label.
34 data Offer u1 u2 = Offer (IO Bool) u1 u2
35 -- | ‘Bot‘ is the type for a channel whose both endpoints are already engaged by two processes, so that no
36 --   further processes can own that channel.
37 --   For example,  in @forkIO (send k e) >>> recv k@, @k@ has type ‘Bot‘.
38 data Bot = Bot
39 -- | ‘End‘ denotes a terminated session. Further communication along a channel with type ‘End‘ cannot
40 --   take place.
41 data End = End
42 -- | @‘Rec‘ m r@ denotes recursive session, where @m@ represents the binder of recursion variable.
43 -- a type-level natural numer (like @‘S‘ ‘Z‘@). nesting level of @‘Rec‘@, and
44 -- @r@ is the body of the recursion which may contain @‘Var‘ m@.
45 data Rec m r = Rec m r deriving Show
46 -- | Recursion variable.
47 data Var n = Var n deriving Show
48 -- | @‘Close‘@ denotes a session that can do nothing but closing it.
49 data Close = Close (IO ())
50 data SelectN u1 u2 = SelectN u1 u2
51 data OfferN u1 u2 = OfferN (IO (Maybe Bool)) u1 u2
52
53
54 -- | The channel type. The type-level number @n@ points to the session-type in type environments. For
55 --  example, in the type @Session t (Nil:>Send Int End) (Nil:>End) ()@,
56 -- the usage of the channel @c :: Channel t Z@ is @Send Int End@ in pretype and @End@ in posttype.
57 newtype Channel t n = C n
58
59
60 class Diff xx yy zz | xx yy -> zz where
61   diff :: Either xx yy -> zz
62 instance (SList xx lx, SList yy ly, Diff' (SubT lx ly) xx yy zz, Sub lx ly) => Diff xx yy zz where
63   diff e  = (\sub_ diff' ->
64     case e of
65       Left xx -> fst (diff' (sub_ (Left (len_ xx)))) xx
66       Right yy -> snd (diff' (sub_ (Right (len_ yy)))) yy
67     ) sub_ diff'
68
69 class Diff' n xx yy zz | n xx yy -> zz where
```

```
  diff' :: n -> (xx -> zz, yy -> zz)
instance (xx ~ zz, yy ~ zz) => Diff' Z xx yy zz where
  diff' _ = (id, id)
instance (Diff' n xx' yy zz', xx ~ (xx' :> End), zz ~ (zz' :> End)) => Diff' (S n) xx yy zz where
  diff' (S n) = (\f -> (\(xx' :> End) -> fst f xx' :> End, \yy -> snd f yy :> End)) (diff' n)
instance (Diff' n xx yy' zz', yy ~ (yy' :> End), zz ~ (zz' :> End)) => Diff' (P n) xx yy zz where
  diff' (P n) = (\f -> (\xx -> fst f xx :> End, \(yy' :> End) -> snd f yy' :> End)) (diff' n)
```

ソースコード A.5: モジュール FullSession.TypeAlgebra

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeFamilies #-}

module FullSession.TypeAlgebra where

import Control.Concurrent

import FullSession.Base
import FullSession.Types

-- |duality
class Dual n s t | s -> t, t -> s where
  dual :: n -> IO (s, t)
instance Dual n End End where
  dual _ = return (End, End)
instance Dual n Close Close where
  dual _ = return (Close (return ()), Close (return ()))
instance (Dual n u u', t ~ t') => Dual n (Send t u) (Recv t' u') where
  dual n = do m <- newEmptyMVar; (u, u') <- dual n; return (Send (putMVar m) u, Recv (takeMVar m) u')
instance (Dual n u' u, t ~ t') => Dual n (Recv t' u') (Send t u) where
  dual n = do m <- newEmptyMVar; (u, u') <- dual n; return (Recv (takeMVar m) u, Send (putMVar m) u')
instance (Dual n u1 u1', Dual n u2 u2') => Dual n (Select u1 u2) (Offer u1' u2') where
  dual n = do m <- newEmptyMVar;
              (u1, u1') <- dual n; (u2, u2') <- dual n;
              return (Select (putMVar m) u1 u2, Offer (takeMVar m) u1' u2')
instance (Dual n u1 u1', Dual n u2 u2') => Dual n (Offer u1 u2) (Select u1' u2') where
  dual n = do m <- newEmptyMVar;
              (u1, u1') <- dual n; (u2, u2') <- dual n;
              return (Offer (takeMVar m) u1 u2, Select (putMVar m) u1' u2')
instance (Dual n u u', v ~ v') => Dual n (Throw v u) (Catch v' u') where
  dual n = do m <- newEmptyMVar; (u, u') <- dual n; return (Throw (putMVar m) u, Catch (takeMVar m) u')
instance (Dual n u u', v ~ v') => Dual n (Catch v u) (Throw v' u') where
  dual n = do m <- newEmptyMVar; (u, u') <- dual n; return (Catch (takeMVar m) u, Throw (putMVar m) u')
instance (Dual (S n) r r', n ~ m, n ~ m') => Dual n (Rec m r) (Rec m' r') where
  dual n = do (r, r') <- dual (S n); return (Rec n r, Rec n r')
instance (Nat v, v ~ v') => Dual n (Var v) (Var v') where
  dual _ = return (Var nat, Var nat)


-- |sesion type algebra - compose two dual types into Bot, and use End as a Zero-element
--  where both @Comp s End s@ and @Comp End s s@ holds.
class Comp s t u | s u -> t, t u -> s, s t -> u where
  decomp :: u -> IO (s, t)
instance (Dual Z u u', t ~ t') => Comp (Send t u) (Recv t' u') Bot where
  decomp _ = do (s,t) <- dual Z; return (s, t)
instance (Dual Z u u' , t ~ t') => Comp (Recv t u) (Send t' u') Bot where
  decomp _ = do (s,t) <- dual Z; return (s, t)
instance (Dual Z u1 u1', Dual Z u2 u2') => Comp (Select u1 u2) (Offer u1' u2') Bot where
  decomp _ = do (s,t) <- dual Z; return (s, t)
instance (Dual Z u1 u1', Dual Z u2 u2') => Comp (Offer u1 u2) (Select u1' u2') Bot where
  decomp _ = do (s,t) <- dual Z; return (s, t)
```

```
53  instance (Dual Z u u' , t ~ t') => Comp (Throw t u) (Catch t' u') Bot where
54    decomp _ = do (s,t) <- dual Z; return (s, t)
55  instance (Dual Z u u' , t ~ t') => Comp (Catch t u) (Throw t' u') Bot where
56    decomp _ = do (s,t) <- dual Z; return (s, t)
57  instance (Dual (S Z) r r', m ~ Z, m' ~ Z) => Comp (Rec m r) (Rec m' r') Bot where
58    decomp _ = do (s,t) <- dual (S Z); return (Rec Z s, Rec Z t)
59
60
61  instance (v ~ v'', u ~ u'') => Comp (Send v u) End (Send v'' u'') where
62    decomp c = return (c, End)
63  instance (v ~ v'', u ~ u'') => Comp (Recv v u) End (Recv v'' u'') where
64    decomp c = return (c, End)
65  instance (u1 ~ u1'', u ~ u'') => Comp (Throw u1 u) End (Throw u1'' u'') where
66    decomp c = return (c, End)
67  instance (u1 ~ u1'', u ~ u'') => Comp (Catch u1 u) End (Catch u1'' u'') where
68    decomp c = return (c, End)
69  instance (u1 ~ u1'', u2 ~ u2'') => Comp (Select u1 u2) End (Select u1'' u2'') where
70    decomp c = return (c, End)
71  instance (u1 ~ u1'', u2 ~ u2'') => Comp (Offer u1 u2) End (Offer u1'' u2'') where
72    decomp c = return (c, End)
73  instance (u1 ~ u1'', u2 ~ u2'') => Comp (SelectN u1 u2) End (SelectN u1'' u2'') where
74    decomp c = return (c, End)
75  instance (u1 ~ u1'', u2 ~ u2'') => Comp (OfferN u1 u2) End (OfferN u1'' u2'') where
76    decomp c = return (c, End)
77  instance (u ~ u'', m ~ Z, m' ~ Z) => Comp (Rec m u) End (Rec m' u'') where
78    decomp c = return (c, End)
79
80
81  instance (v ~ v'', u ~ u'') => Comp End (Send v u) (Send v'' u'') where
82    decomp c = return (End, c)
83  instance (v ~ v'', u ~ u'') => Comp End (Recv v u) (Recv v'' u'') where
84    decomp c = return (End, c)
85  instance (u1 ~ u1'', u ~ u'') => Comp End (Throw u1 u) (Throw u1'' u'') where
86    decomp c = return (End, c)
87  instance (u1 ~ u1'', u ~ u'') => Comp End (Catch u1 u) (Catch u1'' u'') where
88    decomp c = return (End, c)
89  instance (u1 ~ u1'', u2 ~ u2'') => Comp End (Select u1 u2) (Select u1'' u2'') where
90    decomp c = return (End, c)
91  instance (u1 ~ u1'', u2 ~ u2'') => Comp End (Offer u1 u2) (Offer u1'' u2'') where
92    decomp c = return (End, c)
93  instance (u1 ~ u1'', u2 ~ u2'') => Comp End (SelectN u1 u2) (SelectN u1'' u2'') where
94    decomp c = return (End, c)
95  instance (u1 ~ u1'', u2 ~ u2'') => Comp End (OfferN u1 u2) (OfferN u1'' u2'') where
96    decomp c = return (End, c)
97  instance (u ~ u'', m ~ Z, m' ~ Z) => Comp End (Rec m u) (Rec m' u'') where
98    decomp c = return (End, c)
99
100 instance Comp End Close Close where
101   decomp c = return (End, c)
102 instance Comp Close End Close where
103   decomp c = return (c,End)
104
105 instance Comp End Bot Bot where
106   decomp c = return (End, c)
107 instance Comp Bot End Bot where
108   decomp c = return (c, End)
109
110 instance Comp End End End where
111   decomp c = return (c, c)
```

ソースコード A.6: モジュール FullSession.Ended

```haskell
{-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeFamilies #-}

module FullSession.Ended where

import FullSession.Base
import FullSession.Types

-- | @'Ended' n ss@ denotes that the session-type environment @ss@ (the length of it is @n@) is Ended.
--  The all elements in an Ended type environments are @'End'@.
class (SList ss n, IsEnded ss T) => Ended n ss | n -> ss where
  ended :: n -> ss
instance Ended Z Nil where
  ended _ = Nil
instance Ended n ss => Ended (S n) (ss :> End) where
  ended ~(S n) = ended n :> End

-- | @'IsEnded' ss b@ denotes that b ~ T if @ss@ is Ended, otherwise @b ~ F@.  In other words, @b ~ T@
--   if the all elements of ss are End
class IsEnded ss b | ss -> b
instance IsEnded Nil T
instance (IsEnded ss b) => IsEnded (ss:>End) b

class IsEndedST s b | s -> b
instance IsEndedST End T
instance IsEndedST (Send x y) F
instance IsEndedST (Recv x y) F
instance IsEndedST (Throw x y) F
instance IsEndedST (Catch x y) F
instance IsEndedST (Select x y) F
instance IsEndedST (Offer x y) F
instance IsEndedST (SelectN x y) F
instance IsEndedST (OfferN x y) F
instance IsEndedST (Bot) F
instance IsEndedST (Close) F
instance IsEndedST (Rec n r) F
instance IsEndedST (Var v) F


class EndedWithout n s ss | n s -> ss
instance (SList ss l, EndedWithout' (SubT l (S n)) s l ss) => EndedWithout n s ss
class EndedWithout' n s l ss  | n s l -> ss
instance (ss ~ (ss':>s), l ~ S l', Ended l'  ss', IsEnded ss' T) => EndedWithout' Z s l ss
instance (ss ~ (ss':>End), l ~ S l', EndedWithout' n s l' ss') => EndedWithout' (S n) s l ss

class EndedWithout2 n m s t ss  | n s m t -> ss
instance (SList ss l, EndedWithout2' (SubT l (S n)) (SubT l (S m)) s t l ss) => EndedWithout2 n m s t ss
class EndedWithout2' n m s t l ss  | n m s t l -> ss
instance (ss ~ (ss':>s), l ~ S l', EndedWithout' m t l' ss') => EndedWithout2' Z (S m) s t l ss
instance (ss ~ (ss':>t), l ~ S l', EndedWithout' n s l' ss') => EndedWithout2' (S n) Z s t l ss
instance (ss ~ (ss':>End), l ~ S l', EndedWithout2' n m s t l' ss') => EndedWithout2' (S n) (S m) s t l ss


class AppendEnd ss ss' where
  appendEnd :: ss -> ss'
  deleteEnd :: ss' -> ss
instance (SList ss l, SList ss' l', Sub l' l, AppendEnd' (SubT l' l) ss ss') => AppendEnd ss ss' where
  appendEnd ss  = let d = sub (len_ ss') (len_ ss); ss' = appendEnd' d ss in ss'
  deleteEnd ss' = let d = sub (len_ ss') (len_ ss); ss = deleteEnd' d ss' in ss
class AppendEnd' n ss ss' | n ss -> ss', n ss' -> ss where
  appendEnd' :: n -> ss -> ss'
```

```
64    deleteEnd' :: n -> ss' -> ss
65  instance ss ~ ss' => AppendEnd' Z ss ss' where
66    appendEnd' _ ss = ss
67    deleteEnd' _ ss = ss
68  instance (AppendEnd' n ss ss'', ss' ~ (ss'':>End)) => AppendEnd' (S n) ss ss' where
69    appendEnd' (S n) ss = appendEnd' n ss :> End
70    deleteEnd' (S n) (ss:>_) = deleteEnd' n ss
```

ソースコード A.7: モジュール FullSession.Recursion

```
1   {-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
2   {-# LANGUAGE UndecidableInstances #-}
3   {-# LANGUAGE TypeFamilies #-}
4
5   module FullSession.Recursion where
6
7   import FullSession.TypeEq
8   import FullSession.Base
9   import FullSession.Types
10
11  -- folding
12  class RecFold m u s r | m u -> r  where
13    fold' :: m -> u -> s -> r
14  instance RecFold m (Var n) s (Var n) where
15    fold' _ v _ = v
16  instance (TypeEq m n b, RecFoldCont b m n a s r) => RecFold m (Rec n a) s r where -- fold if m = n
17    fold' m (Rec n a) s = fold'cont (type'eq m n) m n a s
18
19  class RecFoldCont b m n a s r | b m n a -> r where
20     fold'cont :: b -> m -> n -> a -> s -> r
21  instance a ~ s => RecFoldCont T m n a s (Var m) where -- fold
22     fold'cont _ m _ _ _ = Var m
23  instance (RecFold2 n s s', RecUnfold n s' xx s, RecFold m a s' r)
24        => RecFoldCont F m n a s (Rec n r) where -- do not fold
25     fold'cont b m n a s = Rec n (fold' m a (fold2' n s))
26
27  -- folding
28  class RecFold2 m u r | m u -> r  where
29    fold2' :: m -> u -> r
30  instance RecFold2 m (Var n) (Var n) where
31    fold2' _ v = v
32  instance (TypeEq m n b, RecFoldCont2 b m n a r) => RecFold2 m (Rec n a) r where -- fold if m = n
33    fold2' m (Rec n a) = fold2'cont (type'eq m n) m n a
34
35  class RecFoldCont2 b m n a r | b m n a -> r where
36     fold2'cont :: b -> m -> n -> a -> r
37  instance RecFoldCont2 T m n a (Var m) where -- fold
38     fold2'cont _ m _ _ = Var m
39  instance (RecFold2 m a r)
40        => RecFoldCont2 F m n a (Rec n r) where -- do not fold
41     fold2'cont b m n a = Rec n (fold2' m a)
42
43  -- unfolding
44  class RecUnfold m r s u | m r s -> u where
45    unfold' :: m -> r -> s -> u
46  instance (RecFold2 n s s', RecUnfold m a s' a') => RecUnfold m (Rec n a) s (Rec n a') where
47    unfold' m (Rec n a) s = Rec n (unfold' m a (fold2' n s))
48  instance (TypeEq m n b, RecUnfoldCont b m n s a) => RecUnfold m (Var n) s a where -- unfold if m = n
49    unfold' m (Var n) s = unfoldCont (type'eq m n) m n s
50
51  class RecUnfoldCont b m n s a | b m n s -> a where
52    unfoldCont :: b -> m -> n -> s -> a
```

```
53 instance RecUnfoldCont T m n s (Rec m s) where -- unfold
54   unfoldCont _ m _ s = Rec m s
55 instance RecUnfoldCont F m n s (Var n) where -- do not unfold
56   unfoldCont _ _ n _ = Var n
57
58 -- RecUnfold ensures that our fold/unfold is isomorphic
59 unfold :: (RecFold m u r r, RecUnfold m r r u) => Rec m r -> u
60 unfold (Rec m r) = unfold' m r r
61
62 unfold0 :: (RecFold Z u r r, RecUnfold Z r r u) => Rec Z r -> u
63 unfold0 (Rec m r) = unfold' m r r
64
65 unfold1 :: (RecFold (S Z) u r r, RecUnfold (S Z) r r u) => Rec (S Z) r -> u
66 unfold1 (Rec m r) = unfold' m r r
67
68 unfold2 :: (RecFold (S (S Z)) u r r, RecUnfold (S (S Z)) r r u) => Rec (S (S Z)) r -> u
69 unfold2 (Rec m r) = unfold' m r r
70
71
72 instance (RecUnfold m u s u', v ~ v') => RecUnfold m (Send v u) s (Send v' u') where
73   unfold' m (Send c u) s = Send c (unfold' m u s)
74 instance (RecUnfold m u s u', v ~ v') => RecUnfold m (Recv v u) s (Recv v' u') where
75   unfold' m (Recv c u) s = Recv c (unfold' m u s)
76 instance (RecUnfold m u s u', v ~ v') => RecUnfold m (Throw v u) s (Throw v' u') where
77   unfold' m (Throw c u) s = Throw c (unfold' m u s)
78 instance (RecUnfold m u s u', v ~ v') => RecUnfold m (Catch v u) s (Catch v' u') where
79   unfold' m (Catch c u) s = Catch c (unfold' m u s)
80 instance (RecUnfold m u1 s u1', RecUnfold m u2 s u2')
81   => RecUnfold m (Select u1 u2) s (Select u1' u2') where
82   unfold' m (Select c u1 u2) s = Select c (unfold' m u1 s) (unfold' m u2 s)
83 instance (RecUnfold m u1 s u1', RecUnfold m u2 s u2')
84   => RecUnfold m (Offer u1 u2) s (Offer u1' u2') where
85   unfold' m (Offer c u1 u2) s = Offer c (unfold' m u1 s) (unfold' m u2 s)
86 instance (RecUnfold m u1 s u1', RecUnfold m u2 s u2')
87   => RecUnfold m (SelectN u1 u2) s (SelectN u1' u2') where
88   unfold' m (SelectN u1 u2) s = SelectN (unfold' m u1 s) (unfold' m u2 s)
89 instance (RecUnfold m u1 s u1', RecUnfold m u2 s u2')
90   => RecUnfold m (OfferN u1 u2) s (OfferN u1' u2') where
91   unfold' m (OfferN c u1 u2) s = OfferN c (unfold' m u1 s) (unfold' m u2 s)
92 instance RecUnfold m End s End where
93   unfold' m End s = End
94 instance RecUnfold m Close s Close where
95   unfold' m (Close c) s = Close c
96
97 instance (RecFold m u s u', v ~ v') => RecFold m (Send v u) s (Send v' u') where
98   fold' m (Send c u) s = Send c (fold' m u s)
99 instance (RecFold m u s u', v ~ v') => RecFold m (Recv v u) s (Recv v' u') where
100  fold' m (Recv c u) s = Recv c (fold' m u s)
101 instance (RecFold m u s u', v ~ v') => RecFold m (Throw v u) s (Throw v' u') where
102  fold' m (Throw c u) s = Throw c (fold' m u s)
103 instance (RecFold m u s u', v ~ v') => RecFold m (Catch v u) s (Catch v' u') where
104  fold' m (Catch c u) s = Catch c (fold' m u s)
105 instance (RecFold m u1 s u1', RecFold m u2 s u2')
106  => RecFold m (Select u1 u2) s (Select u1' u2') where
107  fold' m (Select c u1 u2) s = Select c (fold' m u1 s) (fold' m u2 s)
108 instance (RecFold m u1 s u1', RecFold m u2 s u2')
109  => RecFold m (Offer u1 u2) s (Offer u1' u2') where
110  fold' m (Offer c u1 u2) s = Offer c (fold' m u1 s) (fold' m u2 s)
111 instance (RecFold m u1 s u1', RecFold m u2 s u2')
112  => RecFold m (SelectN u1 u2) s (SelectN u1' u2') where
113  fold' m (SelectN u1 u2) s = SelectN (fold' m u1 s) (fold' m u2 s)
114 instance (RecFold m u1 s u1', RecFold m u2 s u2')
115  => RecFold m (OfferN u1 u2) s (OfferN u1' u2') where
```

```
116   fold' m (OfferN c u1 u2) s = OfferN c (fold' m u1 s) (fold' m u2 s)
117 instance RecFold m End s End where
118   fold' m End _ = End
119 instance RecFold m Close s Close where
120   fold' m (Close c) _ = Close c
121
122 instance (RecFold2 m u u', v ~ v') => RecFold2 m (Send v u) (Send v' u') where
123   fold2' m (Send c u) = Send c (fold2' m u)
124 instance (RecFold2 m u u', v ~ v') => RecFold2 m (Recv v u) (Recv v' u') where
125   fold2' m (Recv c u) = Recv c (fold2' m u)
126 instance (RecFold2 m u u', v ~ v') => RecFold2 m (Throw v u) (Throw v' u') where
127   fold2' m (Throw c u) = Throw c (fold2' m u)
128 instance (RecFold2 m u u', v ~ v') => RecFold2 m (Catch v u) (Catch v' u') where
129   fold2' m (Catch c u) = Catch c (fold2' m u)
130 instance (RecFold2 m u1 u1', RecFold2 m u2 u2') => RecFold2 m (Select u1 u2) (Select u1' u2') where
131   fold2' m (Select c u1 u2) = Select c (fold2' m u1) (fold2' m u2)
132 instance (RecFold2 m u1 u1', RecFold2 m u2 u2') => RecFold2 m (Offer u1 u2) (Offer u1' u2') where
133   fold2' m (Offer c u1 u2) = Offer c (fold2' m u1) (fold2' m u2)
134 instance (RecFold2 m u1 u1', RecFold2 m u2 u2') => RecFold2 m (SelectN u1 u2) (SelectN u1' u2') where
135   fold2' m (SelectN u1 u2) = SelectN (fold2' m u1) (fold2' m u2)
136 instance (RecFold2 m u1 u1', RecFold2 m u2 u2') => RecFold2 m (OfferN u1 u2) (OfferN u1' u2') where
137   fold2' m (OfferN c u1 u2) = OfferN c (fold2' m u1) (fold2' m u2)
138 instance RecFold2 m End End where
139   fold2' m End = End
140 instance RecFold2 m Close Close where
141   fold2' m (Close c) = Close c
```

ソースコード A.8: モジュール FullSession.SMonad

```
1  {-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
2  {-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
3  {-# LANGUAGE UndecidableInstances #-}
4  {-# LANGUAGE Rank2Types #-}
5
6  module FullSession.SMonad where
7
8  import FullSession.Base
9  import FullSession.Types
10 import FullSession.Ended
11
12 -- | The @Session@ monad. @ss@ and @tt@ denotes the /usage/ of channels.
13 --
14 --   * @ss@ denotes /pre-type/, which denotes the type-level list of session types /required/ to run
15 --     the session.
16 --
17 --   * @tt@ denotes /post-type/, which denotes the type-level lists of session types /produced/ by
18 --     the session.
19 --
20 --   @t@ denotes a /type-tag/, which prevents abuse of use of channels. For detail, see 'runS'.
21 --
22 newtype Session t ss tt a = Session { session :: ss -> IO (tt,a) }
23
24 -- | Bind (a.k.a @>>=@) operation for 'Session' monad.
25 (>>>=) :: Session t ss tt a -> (a -> Session t tt uu b) -> Session t ss uu b
26 (Session m) >>>= f = Session (\ss -> m ss >>= \(tt,x) -> session (f x) tt)
27
28 (>>>) :: Session t ss tt a -> Session t tt uu b -> Session t ss uu b
29 (Session m) >>> (Session n) = Session (\ss -> m ss >>= \(tt,_) -> n tt)
30
31 -- | Unit (a.k.a @return@) operation for 'Session' monad.
32 ireturn :: a -> Session t ss ss a
33 ireturn a = Session (\ss -> return (ss, a))
```

```
34
35  -- | 'runS' runs the 'Session'. The pretype (see 'Session') must be 'Nil'.
36  --   The posttype must be 'Ended', i.e. all channels must be 'End'.
37  --
38  --   Forall'd type variable @t@ prevents abuse of use of channels inside different run.
39  --   For example, @new >>>= \c -> 'io_' (runS ( ... send c ...) )@ is rejected by the Haskell typechecker
40  --   with error @Inferred type is less polymorphic than expected@.
41  runS :: Ended n ss => forall a n. (forall t. Session t Nil ss a) -> IO a
42  runS s = case s of Session m -> m Nil >>= \(_,a) -> return a
43
44
45  channeltype1 :: Pickup ss' Z s' => (Channel t Z -> Session t (Nil:>s) ss' a) -> (s, s')
46  channeltype1 = error "ERROR - channeltype1 is for type checking purpose only!"
47  channeltype2 :: (Pickup ss' Z s', Pickup ss' (S Z) t')
48    => (Channel t Z -> Channel t (S Z) -> Session t (Nil:>s:>t) ss' a)
49    -> ((s, s'), (t, t'))
50  channeltype2 = error "ERROR - channeltype2 is for type checking purpose only!"
51
52  typecheck1
53    :: (SList ss l) =>
54       (Channel t l -> Session t (ss:>s1) ss' a)
55       -> Session t (ss:>s1) ss' a
56  typecheck1 = error "ERROR - typecheck2 is for type checking purpose only!"
57
58  typecheck2
59    :: (SList ss l) =>
60       (Channel t l -> Channel t (S l) -> Session t (ss:>s1:>s2) ss' a)
61       -> Session t (ss:>s1:>s2) ss' a
62  typecheck2 = error "ERROR - typecheck2 is for type checking purpose only!"
63
64  inspect1 :: SList tt l
65    => (Channel t l -> Session t ss ss () -> Session t (tt:>t1) tt' a)
66    -> (ss, Session t (tt:>t1) tt' a)
67  inspect1 = undefined
```

ソースコード A.9: モジュール FullSession.FullSession

```
1   {-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
2   {-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
3   {-# LANGUAGE UndecidableInstances #-}
4   {-# LANGUAGE NoMonomorphismRestriction #-}
5   {-# LANGUAGE TypeFamilies #-}
6   {-# LANGUAGE Rank2Types #-}
7
8
9   module FullSession.FullSession where
10
11  import Control.Concurrent
12
13  import FullSession.Base
14  import FullSession.TypeEq
15  import FullSession.Types
16  import FullSession.TypeAlgebra
17  import FullSession.Ended
18  import FullSession.SMonad
19  import FullSession.Recursion
20
21  close :: (Pickup ss n Close, Update ss n End ss', IsEnded ss F)
22        => Channel t n -> Session t ss ss' ()
23  close (C n) = Session (\ss -> case pickup ss n of Close c -> c >> (return (update ss n End, ())))
24
25  send :: (Pickup ss n (Send v a), Update ss n a ss', IsEnded ss F)
```

```
      => Channel t n -> v -> Session t ss ss' ()
send (C n) v =
  Session (\ss -> case pickup ss n of (Send outp u) -> outp v >> (return (update ss n u, ())))

recv :: (Pickup ss n (Recv v u), Update ss n u ss', IsEnded ss F)
      => Channel t n -> Session t ss ss' v
recv (C n) =
  Session (\ss -> case pickup ss n of (Recv inp u) -> inp >>= \x -> (return (update ss n u, x)))

sendS :: (Pickup ss n1 (Throw c1 u1),
          Update ss n1 u1 ss',
          Pickup ss' n2 c1,
          Update ss' n2 End ss'',
          IsEnded ss F)
       => Channel t n1 -> Channel t n2 -> Session t ss ss'' ()
sendS (C n1) (C n2) = Session (\ss ->
    case pickup ss n1 of
      (Throw outp u1) ->
        let ss' = update ss n1 (u1) in
          outp (pickup ss' n2) >> return (update ss' n2 (End), ()))

recvS :: (Pickup ss n1 (Catch c1 u1),
          Update ss n1 u1 ss',
          SList ss' l,
          IsEnded ss F)
       => Channel t n1 -> Session t ss (ss':>c1) (Channel t l)
recvS (C n1) = Session (\ss ->
    case pickup ss n1 of
      (Catch inp u1) ->
        let ss' = update ss n1 (u1) in
          inp >>= \c -> return (ss' :> c, C (len_ ss')))

-- |output a label `1'
sel1 :: (Pickup ss n (Select s x), Update ss n s tt, IsEnded ss F) => Channel t n -> Session t ss tt ()
sel1 (C n) = Session (\ss -> case pickup ss n of Select c s _ -> c True >> return (update ss n s, ()))

-- |output a label `2'
sel2 :: (Pickup ss n (Select x s), Update ss n s tt, IsEnded ss F) => Channel t n -> Session t ss tt ()
sel2 (C n) = Session (\ss -> case pickup ss n of Select c _ s -> c False >> return (update ss n s, ()))

ifSelect :: (Pickup ss n (Select x y), Update ss n x sx, Update ss n y sy, Diff xx yy zz, IsEnded ss F)
         => Channel t n
         -> Bool
         -> Session t sx xx a
         -> Session t sy yy a
         -> Session t ss zz a
ifSelect (C n) b (Session s) (Session t) = Session $ \ss -> case pickup ss n of
  (Select outp x y) -> outp b >> (\diff ->
    if b then s (update ss n x) >>= \(xx,a) -> return (diff (Left xx), a)
         else t (update ss n y) >>= \(yy,a) -> return (diff (Right yy), a)) diff

offer :: (Pickup ss n (Offer x y), Update ss n x sx, Update ss n y sy, Diff xx yy zz, IsEnded ss F)
      => Channel t n
      -> Session t sx xx a
      -> Session t sy yy a
      -> Session t ss zz a
offer (C n) (Session s) (Session t) = Session $ \ss -> case pickup ss n of
  (Offer inp x y) -> inp >>= \b -> (\diff ->
    if b then s (update ss n x) >>= \(xx,a) -> return (diff (Left xx), a)
         else t (update ss n y) >>= \(yy,a) -> return (diff (Right yy), a)) diff

-- |pointwise extension of `Comp'
class Par ss tt' tt | ss tt' -> tt, ss tt -> tt', tt tt' -> ss where
```

```
   split :: tt -> IO (ss, tt')
instance Par Nil Nil Nil where
   split _ = return (Nil, Nil)
instance (Comp s t' t, IsEnded ss b, Par' b ss tt' tt) => Par (ss:>s) (tt':>t') (tt:>t) where
   split (uu':>u) = do (ss',tt') <- split' uu'; (s,t) <- decomp u; return (ss':>s,tt':>t)

-- |the specialized case for 'ended' ss
class Par' t ss tt' tt | t ss tt' -> tt, t ss tt-> tt', t tt tt' -> ss where
  split' :: IsEnded ss t => tt -> IO (ss, tt')
instance (SList tt n, Ended n ss, IsEnded ss T, tt' ~ tt) => Par' T ss tt' tt where
  split' x = return (ended (len_ x),x)
instance (IsEnded ss F, Par ss tt' tt) => Par' F ss tt' tt where
  split' u = do (ss,tt) <- split u; return (ss, tt)

-- |start a new thread
forkIOs, forkOSs :: (SList ss l, SList tt' l, SList tt l, Ended l' ss', IsEnded ss b, Par' b ss tt' tt)
                 => Session t ss ss' () -> Session t tt tt' ThreadId
forkIOs (Session f) =
  Session (\tt -> do (ss, tt') <- split' tt; tid <- forkIO (f ss >> return ()); return (tt', tid))
forkOSs (Session f) =
  Session (\tt -> do (ss, tt') <- split' tt; tid <- forkOS (f ss >> return ()); return (tt', tid))

io :: IO a -> Session t ss ss a
io m = Session (\ss -> m >>= \x -> return (ss, x))

io_ :: IO a -> Session t ss ss ()
io_ m = Session (\ss -> m >> return (ss, ()))

new :: SList ss l => Session t ss (ss:>Bot) (Channel t l)
new = Session (\ss -> return (ss:>Bot, C (len_ ss)))

unwind :: (RecFold m u r r, RecUnfold m r r u,
           Pickup ss n (Rec m r),
           Update ss n u tt,
           IsEnded ss F)
        => Channel t n -> Session t ss tt ()
unwind (C n) = Session $ \ss -> return (update ss n (unfold (pickup ss n)), ())

unwind0 :: (RecFold Z u r r, RecUnfold Z r r u,
           Pickup ss n (Rec Z r),
           Update ss n u tt,
           IsEnded ss F)
        => Channel t n -> Session t ss tt ()
unwind0 (C n) = Session $ \ss -> return (update ss n (unfold0 (pickup ss n)), ())

unwind1 :: (RecFold (S Z) u r r, RecUnfold (S Z) r r u,
           Pickup ss n (Rec (S Z) r),
           Update ss n u tt,
           IsEnded ss F)
        => Channel t n -> Session t ss tt ()
unwind1 (C n) = Session $ \ss -> return (update ss n (unfold1 (pickup ss n)), ())

unwind2 :: (RecFold (S (S Z)) u r r, RecUnfold (S (S Z)) r r u,
           Pickup ss n (Rec (S (S Z)) r),
           Update ss n u tt,
           IsEnded ss F)
        => Channel t n -> Session t ss tt ()
unwind2 (C n) = Session $ \ss -> return (update ss n (unfold2 (pickup ss n)), ())

recur1 :: (EndedWithout n s ss, AppendEnd ss ss', SList ss' l, Ended l tt) =>
          (Channel t n -> Session t ss tt ()) ->
          Channel t n -> Session t ss' tt ()
recur1 f c =
```

```
  case f c of Session m -> Session (\ss' -> m (deleteEnd ss') >> return (ended (len_ ss'), ()))

recur2 :: (EndedWithout2 n m s s' ss, AppendEnd ss ss', SList ss' l, Ended l tt) =>
          (Channel t n -> Channel t m -> Session t ss tt ()) ->
          Channel t n -> Channel t m -> Session t ss' tt ()
recur2 f c d =
  case f c d of Session m -> Session (\ss' -> m (deleteEnd ss') >> return (ended (len_ ss'), ()))


newtype Service u = Service (u -> IO (), IO u)

newService :: Dual Z u u' => IO (Service u)
newService = do mv <- newEmptyMVar; return (Service (putMVar mv, takeMVar mv))

connect :: (Dual Z u u', SList ss l) => Service u -> Session t ss (ss:>u') (Channel t l)
connect (Service (put,_)) = Session (\ss -> do (u,u') <- dual Z; put u; return (ss:>u', C (len_ ss)))

connectRunS :: (Dual Z u u', Ended xs n)
  => Service u -> (forall t. Channel t Z -> Session t (Nil:>u') xs a) -> IO a
connectRunS (Service (put,_)) f =
  do (u,u') <- dual Z; put u; (_,a) <- session (f (C Z)) (Nil:>u'); return a

accept :: (Dual Z u u', SList ss l) => Service u -> Session t ss (ss:>u) (Channel t l)
accept (Service (_,get)) = Session (\ss -> do u <- get; return (ss:>u, C (len_ ss)))

acceptRunS :: (Dual Z u u', Ended xs n)
  => Service u -> (forall t. Channel t Z -> Session t (Nil:>u) xs a) -> IO a
acceptRunS (Service (_,get)) f = do u <- get; (_,a) <- session (f (C Z)) (Nil:>u); return a
```

ソースコード A.10: モジュール FullSession.NwSession

```
{-# OPTIONS -fcontext-stack=100 #-}
{-# LANGUAGE TypeOperators, KindSignatures, ScopedTypeVariables, EmptyDataDecls #-}
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE Rank2Types #-}

module FullSession.NwSession where

import qualified Control.Exception as E
import System.IO.Error (mkIOError, userErrorType)
import Prelude hiding (catch)
import System.IO
import Data.IORef
import qualified Network.Socket as N
import Network.BSD
import Control.Monad (liftM)
import Control.Concurrent
import System.IO.Unsafe (unsafePerformIO)
import System.Exit

import FullSession.Base
import FullSession.Types
import FullSession.TypeAlgebra
import FullSession.Ended
import FullSession.SMonad


errorExit :: String -> a
errorExit str = E.throw (mkIOError userErrorType str Nothing Nothing)
```

```
32
33
34  finish :: (Ended ss n, IsEnded ss T) => String -> Session t ss tt ()
35  finish str = Session (\_ -> errorExit str)
36
37  connectTo :: String -> Int -> IO (Handle, IORef String)
38  connectTo host port_ = do
39        let port = toEnum port_
40        sock <- N.socket N.AF_INET N.Stream 0
41        addrs <- liftM hostAddresses $ getHostByName host
42        if null addrs then errorExit $ "no such host : " ++ host else return ()
43        N.connect sock $ N.SockAddrInet port (head addrs)
44        handle <- N.socketToHandle sock ReadWriteMode
45        str <- hGetContents handle
46        ref <- newIORef str
47        return (handle, ref)
48
49  listenAt :: Int -> IO (Handle, IORef String)
50  listenAt port_ = do
51        let port = toEnum port_
52        lsock <- N.socket N.AF_INET N.Stream 0
53        N.bindSocket lsock $ N.SockAddrInet port N.iNADDR_ANY
54        N.listen lsock 1 -- listen only ONCE
55        (sock,N.SockAddrInet _ _) <- N.accept lsock -- accept only ONCE
56        N.sClose lsock
57        handle <- N.socketToHandle sock ReadWriteMode
58        str <- hGetContents handle
59        ref <- newIORef str
60        return (handle, ref)
61
62
63  newtype NwService u = NwService (String,Int)
64
65  mkNwService :: NwSession u => String -> Int -> u -> NwService u
66  mkNwService str port _ = NwService (str,port)
67
68  connectNw :: (SList ss l, NwSession u) => NwService u -> Session t ss (ss:>u) (Channel t l)
69  connectNw (NwService (host,port)) = Session $ \ss -> do
70        (handle, ref) <- connectTo host port
71        let u = genSession ref handle
72        return (ss:>u, C (len_ ss))
73
74
75  newtype NwService2 u u' = NwService2 (String,Int)
76
77  mkNwService2 :: (NwSendOnly u, NwReceiveOnly u') => String -> Int -> u -> u' -> NwService2 u u'
78  mkNwService2 str port _ _ = NwService2 (str,port)
79
80  connectNw2 :: (SList ss l, NwSendOnly u, NwReceiveOnly u')
81    => NwService2 u u' -> Session t ss (ss:>u:>u') (Channel t l, Channel t (S l))
82  connectNw2 (NwService2 (host,port)) = Session $ \ss -> do
83        (handle, ref) <- connectTo host port
84        let u  = genSession ref handle
85            u' = genSession ref handle
86        return (ss:>u:>u', (C (len_ ss), C (S (len_ ss))))
87
88  acceptOneNw2 :: (SList ss l, NwSendOnly u, NwReceiveOnly u')
89    => NwService2 u u' -> Session t ss (ss:>u:>u') (Channel t l, Channel t (S l))
90  acceptOneNw2 (NwService2 (_,port)) = Session $ \ss -> do
91        (handle, ref) <- listenAt port
92        let u  = genSession ref handle
93            u' = genSession ref handle
94        return (ss:>u:>u', (C (len_ ss), C (S (len_ ss))))
```

```
95
96  dualNw :: NwDual u u' => NwService u -> NwService u'
97  dualNw (NwService (host,port)) = NwService (host,port)
98
99  dualNw2 :: (NwDual u1 u1', NwDual u2 u2') => NwService2 u1 u2 -> NwService2 u1' u2'
100 dualNw2 (NwService2 (host,port)) = NwService2 (host,port)
101
102 class Message mes where
103   parseMessage :: String -> Maybe (mes,String)
104   showMessage :: mes -> String
105
106 class NwSession u => NwSender u
107 instance (NwSession u, Message v) => NwSender (Send v u)
108 instance (NwSender u1, NwSender u2) => NwSender (SelectN u1 u2)
109
110 class NwSession u => NwReceiver u where
111   tryParse :: u -> String -> Bool
112 instance (NwSession u, Message v) => NwReceiver (Recv v u) where
113   tryParse _ str = maybe False (const True) (parseMessage str::Maybe (v,String))
114 instance (NwReceiver u1, NwReceiver u2) => NwReceiver (OfferN u1 u2) where
115   tryParse (OfferN _ u1 u2) str = tryParse u1 str || tryParse u2 str
116
117 class NwSession u where
118   genSession :: IORef String -> Handle -> u
119 instance (Message v, NwSession u) => NwSession (Send v u) where
120   genSession str h = Send (\v -> do hPutStr h (showMessage v); hFlush h) (genSession str h)
121 instance (Message v, NwSession u) => NwSession (Recv v u) where
122   genSession ref h =
123     Recv (do str <- readIORef ref;
124              case parseMessage str of
125                Just (v, rest) -> do writeIORef ref rest; return v;
126                Nothing -> errorExit ("no parse : "++str)
127         ) (genSession ref h)
128 instance NwSession Close where
129   genSession _ h = Close (putStrLn "closing connection.." >> hFlush h >> hClose h)
130 instance (NwSender u1, NwSender u2) => NwSession (SelectN u1 u2) where
131   genSession str h = SelectN (genSession str h) (genSession str h)
132 instance (NwReceiver u1, NwReceiver u2) => NwSession (OfferN u1 u2) where
133   genSession ref h = offer (genSession ref h) (genSession ref h)
134     where
135       offer l r =
136         OfferN
137           (do str <- readIORef ref;
138               if tryParse l str
139                  then return (Just True)
140                  else if tryParse r str
141                         then return (Just False)
142                         else return Nothing)
143           l r
144 instance (NwSession u, Nat m) => NwSession (Rec m u) where
145   genSession str h = Rec nat (genSession str h)
146 instance Nat n => NwSession (Var n) where
147   genSession str h = Var nat
148
149 -- |output a label '1'
150 sel1N :: (Pickup ss n (SelectN s x), Update ss n s tt) => Channel t n -> Session t ss tt ()
151 sel1N (C n) = Session (\ss ->
152   case pickup ss n of SelectN u1 _ -> return (update ss n u1, ()))
153
154 sel2N :: (Pickup ss n (SelectN x s), Update ss n s tt) => Channel t n -> Session t ss tt ()
155 sel2N (C n) = Session (\ss ->
156   case pickup ss n of SelectN _ u2 -> return (update ss n u2, ()))
157
```

```
ifSelectN :: (Pickup ss n (SelectN x y), Update ss n x sx, Update ss n y sy, Diff xx yy zz, IsEnded ss F)
         => Channel t n
         -> Bool
         -> Session t sx xx a
         -> Session t sy yy a
         -> Session t ss zz a
ifSelectN (C n) b (Session s) (Session t) = Session $ \ss -> case pickup ss n of
  (SelectN x y) -> (\diff ->
    if b then s (update ss n x) >>= \(xx,a) -> return (diff (Left xx), a)
         else t (update ss n y) >>= \(yy,a) -> return (diff (Right yy), a)) diff


offerN :: (NwReceiver x, NwReceiver y,
          Pickup ss n (OfferN x y), Update ss n x sx, Update ss n y sy, Diff xx yy zz, IsEnded ss F)
      => Channel t n
      -> Session t sx xx a
      -> Session t sy yy a
      -> Session t ss zz a
offerN (C n) (Session s) (Session t) = Session $ \ss -> case pickup ss n of
  (OfferN test x y) -> test >>= \m -> (\diff ->
    case m of
      Just True  -> s (update ss n x) >>= \(xx,a) -> return (diff (Left xx), a)
      Just False -> t (update ss n y) >>= \(yy,a) -> return (diff (Right yy), a)
      Nothing    -> errorExit "no parse"
    ) diff


class (NwSession s, NwSession t) => NwDual s t | s -> t, t -> s
instance NwDual Close Close
instance (Message t, Message t', NwDual u u', t ~ t') => NwDual (Send t u) (Recv t' u')
instance (Message t, Message t', NwDual u' u, t ~ t') => NwDual (Recv t' u') (Send t u)
instance (NwSender u1, NwReceiver u1', NwSender u2, NwReceiver u2', NwDual u1 u1', NwDual u2 u2')
  => NwDual (SelectN u1 u2) (OfferN u1' u2')
instance (NwReceiver u1, NwSender u1', NwReceiver u2, NwSender u2', NwDual u1 u1', NwDual u2 u2')
  => NwDual (OfferN u1 u2) (SelectN u1' u2')
instance (NwDual r r', Nat m, m ~ m') => NwDual (Rec m r) (Rec m' r')
instance (Nat v, v ~ v') => NwDual (Var v) (Var v')

class NwSession u => NwSendOnly u
instance (NwSendOnly u, Message v) => NwSendOnly (Send v u)
instance (NwSendOnly u1, NwSendOnly u2, NwSender u1, NwSender u2) => NwSendOnly (SelectN u1 u2)
instance NwSendOnly Close
instance (NwSendOnly u, Nat m) => NwSendOnly (Rec m u)
instance Nat n => NwSendOnly (Var n)

class NwSession u => NwReceiveOnly u
instance (NwReceiveOnly u, Message v) => NwReceiveOnly (Recv v u)
instance (NwReceiveOnly u1, NwReceiveOnly u2, NwReceiver u1, NwReceiver u2)
  => NwReceiveOnly (OfferN u1 u2)
instance NwReceiveOnly Close
instance (NwReceiveOnly u, Nat m) => NwReceiveOnly (Rec m u)
instance Nat n => NwReceiveOnly (Var n)

finallys :: Session t ss tt () -> IO () -> Session t ss tt ()
finallys (Session f) m = Session (\tt -> E.finally (f tt) m)
```

ソースコード A.11: モジュール FullSession.Incoherent

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, UndecidableInstances #-}
{-# LANGUAGE TypeOperators, OverlappingInstances, IncoherentInstances #-}

module FullSession.Incoherent where
```

```
5
6  import FullSession.Base
7  import FullSession.TypeEq
8  import FullSession.Types
9  import FullSession.TypeAlgebra
10 import FullSession.Ended
11 import FullSession.Recursion
12 import FullSession.SMonad
13 import FullSession.FullSession
14 import FullSession.NwSession
15
16
17 instance IsEnded (ss:>Send x y) F
18 instance IsEnded (ss:>Recv x y) F
19 instance IsEnded (ss:>Throw x y) F
20 instance IsEnded (ss:>Catch x y) F
21 instance IsEnded (ss:>Select x y) F
22 instance IsEnded (ss:>Offer x y) F
23 instance IsEnded (ss:>SelectN x y) F
24 instance IsEnded (ss:>OfferN x y) F
25 instance IsEnded (ss:>Bot) F
26 instance IsEnded (ss:>Close) F
27 instance IsEnded (ss:>Rec n r) F
28 instance IsEnded (ss:>Var v) F
29 instance (IsEnded ss b1, IsEndedST s b2, And b1 b2 b) => IsEnded (ss:>s) b
30
31 instance And T T T
32 instance And b F F
33 instance And F b F
```

ソースコード A.12: モジュール FullSession.DeferredInstances

```
1  {-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, FlexibleContexts, UndecidableInstances #-}
2  module FullSession.DeferredInstances where
3
4  import FullSession.Base
5
6  instance (SList ss l, Sub l (S n), PickupR ss (SubT l (S n)) s)
7      => Pickup ss n s where
8    pickup ss n = pickupR ss (sub (len_ ss) (S n))
9  instance (SList ss l, Sub l (S n), UpdateR ss (SubT l (S n)) t ss')
10  => Update ss n t ss' where
11   update ss n t = updateR ss (sub (len_ ss) (S n)) t
```

ソースコード A.13: SMTP クライアント smtp.hs

```
1  {-# OPTIONS_GHC -F -pgmF ixdopp -fcontext-stack=120 #-}
2
3  -- a very simple SMTP client.
4
5  import Control.Concurrent.FullSession
6
7  {-
8  $ ghci examples/smtp-client.hs
9
10 -- case 1.
11 *Main> main
12 host name of SMTP server:
13 xxx.yyy.zzz.www
14 port number:
```

```
25
from:
sender@example.com
to:
recipient@example.com
msg:
Hello!
complete.
*Main>

-- case 2.
*Main> main
host name of SMTP server:
xxx.yyy.zzz.www
port number:
25
from:
sender@example.com
to:
nobodyxxxx
msg:
Hello!
["550 <nobodyxxxx>: Recipient address rejected: User unknown in local recipient table\r"]
*Main>
-}


-- Types for SMTP commands.
newtype EHLO = EHLO String; newtype MAIL = MAIL String; newtype RCPT = RCPT String
data    DATA = DATA;        data    QUIT = QUIT;
newtype MailBody = MailBody [String]

-- Types for SMTP server replies (200 OK, 500 error and 354 start mail input)
newtype R2yz = R2yz [String]; newtype R5yz = R5yz [String]; newtype R354 = R354 String

-- auxiliary functions
send_receive_200 c mes = ixdo send c mes; (R2yz _) <- recv c; ireturn ()
send_receive_354 c mes = ixdo send c mes; (R354 _) <- recv c; ireturn ()

sendMail c d =
 ixdo         -- the body of our SMTP client
  (R2yz _) <- recv c                    -- receive 220
  send_receive_200 c (EHLO "mydomain") -- send EHLO, then receive 250
  unwind0 c; sel1N c                    -- (annotation) branch to send 'MAIL FROM'
  from <- recv d                           -- (1) input the sender's address on d
  send_receive_200 c (MAIL from)        -- send 'MAIL FROM', then receive 250
  unwind1 c; sel1N c                    -- (annotation) branch to send 'RCPT TO'
  to <- recv d                             -- (2) input the recipient's address on d
  send c (RCPT to)                      -- send 'RCPT TO'
  --
  offerN c (ixdo                        -- branch the session according to the reply
    (R2yz _) <- recv c                  -- if 250 OK is offered
    sel1 d; mail <- recv d                -- (3) input the content of the mail on d
    unwind1 c; sel2N c                  -- (annotation) branch to send 'DATA'
    send_receive_354 c DATA             -- send 'DATA' and receive 354
    send_receive_200 c (MailBody mail) -- send the content of the mail
    unwind0 c; sel2N c                  -- (annotation) branch to send 'QUIT'
    send c QUIT; close c                -- send 'QUIT' and close the connection
   ) (ixdo
    (R5yz errmsg) <- recv c;            -- if 500 ERROR is offered
    sel2 d; send d errmsg;                -- (4)
    send c QUIT; close c)               -- send 'QUIT' and close the connection
  close d
```

```
78
79
80  start host port from to msg = ixdo
81    d <- new
82    c <- connectNw (mkNwService host port undefined)
83    forkIOs (sendMail c d)
84    send d from
85    send d to
86    offer d (send d [msg] >>> io_ (putStrLn "complete.")) (recv d >>>= \errmsg -> io_ (print errmsg))
87    close d
88
89  main = do
90    putStrLn "host name of SMTP server:"
91    host <- getLine
92    putStrLn "port number:"
93    portstr <- getLine
94    putStrLn "from:"
95    from <- getLine
96    putStrLn "to:"
97    to <- getLine
98    putStrLn "msg:"
99    msg <- getLine
100   runS $ start host (read portstr) from to msg
101
102
103
104 instance Message EHLO where
105   showMessage (EHLO domain) = "EHLO " ++ domain ++ "\r\n"
106   parseMessage = undefined
107
108 instance Message MAIL where
109   showMessage (MAIL revpath) = "MAIL FROM:<"++revpath++">\r\n"
110   parseMessage = undefined
111
112 instance Message R2yz where
113   showMessage = undefined
114   parseMessage r = case readRes r of
115     (ls@(('2':_):_),r') -> Just (R2yz ls,r')
116     _ -> Nothing
117
118 instance Message R5yz where
119   showMessage = undefined
120   parseMessage r = case readRes r of
121     (ls@(('5':_):_),r') -> Just (R5yz ls,r')
122     _ -> Nothing
123
124 instance Message R354 where
125   showMessage = undefined
126   parseMessage r = case readRes r of
127     ([l@('3':'5':'4':_)],r') -> Just (R354 l,r')
128     _ -> Nothing
129
130 instance Message RCPT where
131   showMessage (RCPT addr) = "RCPT TO:<"++addr++">\r\n"
132   parseMessage = undefined
133
134 instance Message DATA where
135   showMessage DATA = "DATA\r\n"
136   parseMessage = undefined
137
138 instance Message MailBody where
139   showMessage (MailBody ls) = mailBody ls
140   parseMessage = undefined
```

```
141
142 instance Message QUIT where
143   showMessage QUIT = "QUIT\r\n"
144   parseMessage = undefined
145
146 cut :: Char -> String -> (String, String)
147 cut c s = let (l,r) = span (/=c) s in (l,safetail r)
148   where
149     safetail (_:s) = s
150     safetail []    = []
151
152 line :: String -> (String, String)
153 line = cut '\n'
154
155 code_line s = splitAt 3 s
156
157 readRes :: String -> ([String],String)
158 readRes s =
159       let (l, r)       = line s
160           (code,l')    = code_line l
161       in
162         case head l' of
163            '-' -> let (ls,r') = readRes r in (l:ls,r')
164            _   -> ([l],r)
165 {-
166 http://tools.ietf.org/html/rfc2821#section-4.5.2
167 4.5.2 Transparency
168
169
170    Without some provision for data transparency, the character sequence
171    "<CRLF>.<CRLF>" ends the mail text and cannot be sent by the user.
172    In general, users are not aware of such "forbidden" sequences.  To
173    allow all user composed text to be transmitted transparently, the
174    following procedures are used:
175
176    -  Before sending a line of mail text, the SMTP client checks the
177       first character of the line.  If it is a period, one additional
178       period is inserted at the beginning of the line.
179
180    -  When a line of mail text is received by the SMTP server, it checks
181       the line.  If the line is composed of a single period, it is
182       treated as the end of mail indicator.  If the first character is a
183       period and there are other characters on the line, the first
184       character is deleted.
185 -}
186 -- each line should be 7 bit strings
187 mailBody :: [String] -> String
188 mailBody ls = mb ls []
189   where
190     mb (l@('.':_):ls) s = '.':l ++ "\r\n"++mb ls s
191     mb (l:ls) s   = l ++ "\r\n" ++ mb ls s
192     mb [] s = ".\r\n" ++ s
```