平成29年度 修士論文

プログラミング言語の操作的意味論に基づく拡張可能な形式化の研究

東京工業大学 情報理工学院 数理・計算科学系
学籍番号 16M30094
奥河 諒

指導教員
増原 英彦教授

平成30年2月27日

概要

プログラミング言語の型安全性とは，正しく型付けされたプログラムは異常終了しない，という数学的性質 (定理) である．プログラムを実行することなく異常終了することがわかればバグの早期発見に有用である．近年，定理証明支援器を用いた定理証明 (以下機械証明と呼ぶ) が盛んである．定理証明支援器とはプログラミング言語の一種である．型と論理・プログラムと証明の対応関係に則って，定理と証明をプログラムとして記述する．定理証明支援器におけるプログラムの実行は証明の検査に対応する．言い換えると，機械証明では証明の正しさが機械的に保証される．しかし機械証明でプログラミング言語の性質を証明することは難しい．プログラミング言語は算術式・ラムダ計算式・条件文などの複数の要素からなるが，機械証明では要素ごとに型安全性を与え，それらを組合せて言語全体の型安全性を与えることが出来ないためである．言い換えると，プログラミング言語に新しい言語要素を追加する度に，機械証明では一から型安全性の証明をやり直す必要がある．本研究では構文，評価・型付け規則，型安全性の証明がそれぞれ拡張可能な枠組みを提案し，実際にラムダ計算と自然数の足し算を混ぜた言語について型安全性を証明する．言語の意味論は A.Charguéraud によって提案された pretty big-step semantics[12] で与える．構文の拡張可能な定義方法には，T. Sheard による 2 層型 (2 level-types)[28] を利用した．拡張可能な評価規則には P. Mosses の modular structural operational semantics(MSOS)[20] を応用した．型安全性の拡張可能な証明には B. Delaware らの proof algebra[16] を応用した．MSOS は small-step semantics における評価規則の定義方法として提案されたものである．本研究では MSOS のアイデアを pretty big-step semantics における評価規則の定義へ応用した．Proof algebra は項の構造に関する帰納法のための枠組みで，表示的意味論でプログラミング言語の意味を与えることを前提としてｓいる．pretty big-step semantics では項の評価の途中で一時項を生成するために，proof algebra を単純に利用することができない．本研究では生成した一時項についての型安全性を項全体の型安全性の証明のための帰納法から独立して証明するによってこの問題を解決した．

謝辞

本修士論文は，筆者が東京工業大学　情報理工学院 数理・計算科学系 数理・計算科学コース修士課程在学中において行った研究をまとめたものです．

本研究を進めるにあたり、ご指導を頂いた卒業論文指導教員の本学増原英彦教授に心より感謝致します．また本論文を何度もご精読頂き多大なる有用なコメントを頂きました本学青谷知幸助教に深謝致します．

最後となりますが Coq に関する数多くの助言を下さった朝倉氏を始め，ここまで一緒に頑張って来た研究室の皆様に心より感謝しております．ありがとうございました．

目 次

図 目 次

表 目 次

第1章　序論

本研究の目的は構文や評価・型付け規則，型安全性の証明を操作的意味論で拡張可能な機械証明の枠組みを提案することである．

1.1　機械証明

数学的な性質の証明を，Coq[2, 14, 7, 11], Isabelle[24], Agda[22] などの証明支援器と呼ばれる証明の正当性を確認してくれる機械を用いて与えることを機械証明と呼ぶ．機械証明を使うメリットは人が紙とペンを使って与える証明よりも信頼性の高い証明を与えることを可能にする点にある．実際，機械証明は数学の分野では四色定理の証明にも利用されている [5]．

型安全性とはプログラミング言語における数学的性質の一つで，正しく型付けされたプログラムは異常終了しない，という性質である．プログラムが異常終了すること，あるいはしないことがプログラムを実行することなくわかることでバグの早期発見に役立ち，正しいプログラムを早く組み上げることに貢献する．近年，この型安全性という性質も機械証明によって与えられることが増えている．

プログラミング言語における数学的性質の証明が数学の分野における証明と違う点は，証明に拡張性が求められることである．例えば Java7 について型安全性の証明がされているとき，Java7 にラムダ式を追加した Java8 についても，Java7 の型安全性の証明を再利用して Java8 の型安全性の証明を与えたい．既存の言語に対する定義や証明を再利用して，その言語に別の言語要素を追加した言語に対する定義や証明を構成することを，本研究では拡張と言うことにする．特に互いに異なる言語要素を持つ言語同士を組み合わせた言語に対する定義や証明を，元になった言語の定義・証明を再利用して構成することを，本研究では定義・証明をモジュラーに与えると定義する．

1.2　差分による言語拡張

本節では単純型付ラムダ計算に自然数と加算を表す言語を追加することを例として差分による言語拡張を説明する．

1.2.1　単純型付ラムダ計算

拡張元の言語である単純型付ラムダ計算の定義をする．単純型付ラムダ計算の構文を以下表 1.1 に示す．

t	:=	x | λx:T.t | t t
v	:=	(λx:T.t, Ξ v)
T	:=	T → T
Ξ a	:=	[x × a]
Γ	:=	Ξ T

表 1.1: 単純型付ラムダ計算の構文

項 t は変数参照，ラムダ抽象，ラムダ適用からなる．値はクロージャである．[...] はリストを表しており，Ξ a は要素の型が a の連想配列である．Γ は型環境であり，要素が型の連想配列として表す．単純型付ラムダ計算に対する大ステップ意味論での評価規則と型付け規則はそれぞれ表 1.2，表 1.3 のようになる．

t $_{env} \Longrightarrow$ v : 評価規則

$$\frac{env(x) = v}{x_{env} \Longrightarrow v}\ (\mathrm{E-Var}) \qquad \frac{}{\lambda x : T.t_{env} \Longrightarrow (\lambda x : T.t,\ env)}\ (\mathrm{E-Lam})$$

$$\frac{t_1\ _{env} \Longrightarrow (\lambda x : T.t,\ env') \quad t_2\ _{env} \Longrightarrow v_2 \quad t_{env'[x \mapsto v_2]} \Longrightarrow v}{t_1\ t_{2env} \Longrightarrow v}\ (\mathrm{E-App})$$

表 1.2: 単純型付ラムダ計算の評価規則（大ステップ意味論）

$\Gamma \vdash t : T$: 型付け規則

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\mathrm{T-Var}) \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t\ :\ T_1 \to T_2}\ (\mathrm{T-Lam})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ (\mathrm{T-App})$$

表 1.3: 単純型付ラムダ計算の型付け規則

表 1.2 に示した大ステップ意味論の評価規則は $t_{env} \Longrightarrow v$ のように書き，項 t を環境 env で評価すると値 v になると読む．E-Var は変数参照の規則であり，環境 env の x に束縛されている要素を参照する．E-Lam はラムダ抽象の規則であり，ラムダ抽象と評価時の環境を組にしてクロージャとする．E-App はラムダ適用の規則である．左の部分項を評価した結果がクロージャであり右の部分項が v_2 に評価できるとき，クロージャのラムダ抽象の中身の項を，クロージャが持つ環境の x に v_2 を束縛した環境で v に評価される場合，ラムダ適用はその v に評価される．

表 1.3 に示した型付け規則は $\Gamma \vdash t : T$ のように書き，型環境 Γ の下で項 t は型 T に型付けされると読む．T-Var は変数参照に対する規則であり，型環境 Γ の x が束縛されていればその型に型付けする．T-Lam はラムダ抽象に対する規則であり，Γ の x に T_1 を束縛した状態で t が T_2 に型付けされるならば，クロージャ $\lambda x : T_1.t$ は $T_1 \to T_2$ に型付けされる．T-App がラムダ適用に対す

る型付けである．t_1 が $T_1 \to T_2$ に型付けされ，t_2 が T_1 に型付けされるとき，ラムダ適用 $t_1\ t_2$ は T_2 に型付けされる．

型安全性とは正しく型付けされたプログラムは異常終了しないという性質であったが，その具体的な定理を以下に示す．

定理 1.2.1 (型安全性)

$$(\Gamma \vdash t : T\ \wedge\ \Gamma \models env\ \wedge\ t_{env} \Longrightarrow v) \Rightarrow \Gamma \vdash v : T$$

証明は評価規則に関する帰納法による．項 t が変数参照とラムダ抽象の場合は自明である [25]．ラムダ適用 $t_1\ t_2$ について値 v に評価できるとき，値にも項と同じ型が付いていることを証明する．

1.2.2 拡張の定義

本節では前節で定義した単純型付け言語に自然数と加算の言語を拡張することで言語拡張のために必要なことを挙げる．

前節の単純型付ラムダ計算を基に言語を拡張する場合，差分として定義できることが望ましい．自然数と加算の言語は以下表 1.4 のように単純型付ラムダ計算との差分として定義したい．

t	:=	... | n | t + t
v	:=	... | n
T	:=	... | `Nat`

表 1.4: 自然数と加算の言語の構文を差分として定義

自然数と加算の言語の項 t は自然数と加算からなる．値は自然数である．型 Nat は自然数に対する型である．この言語の大ステップ意味論における評価規則，型付け規則をそれぞれ表 1.5，表 1.6 に示す．

$t \Longrightarrow v$: 評価規則

$$\frac{}{n \Longrightarrow n}\ (\mathrm{E-Num}) \qquad \frac{e1 \Longrightarrow n1 \quad e2 \Longrightarrow n2 \quad n1 + n2 = n}{e1 + e2 \Longrightarrow n}\ (\mathrm{E-Add})$$

表 1.5: 自然数の加算の評価規則（大ステップ）

表 1.5 に示した E-Num は自然数に対する評価規則である．自然数は既に値であるので受け取った数をそのまま返す．E-Add は加算に対する評価規則である．両方の自然数を評価した結果が共に自然数である場合，それらを足し合わせた結果を返す．

$\Gamma \vdash t : T$: 型付け規則
$\dfrac{}{\Gamma \vdash n : Nat}\ (\mathrm{T}-\mathrm{Num})$　$\dfrac{\Gamma \vdash e1 : Nat \quad \Gamma \vdash e2 : Nat}{\Gamma \vdash e1 + e2 : Nat}\ (\mathrm{T}-\mathrm{Add})$

表 1.6: 自然数の加算の型付け規則

表 1.6 に示した T-Num は自然数に対する型付け規則である．どんな自然数でも値になり得るので任意の自然数に対して型を付ける．T-Add は加算に対する型付け規則である．両方の部分項が自然数に型付けされるとき，加算もまた Nat に型付けされる．

型安全性の証明は単純型付ラムダ計算の証明を以下の二つの場合で拡張する．

- `(e1+e2)`：帰納法の仮定を使って証明
- `n`：値であるため自明

このように構文を一つ拡張するだけでも言語の構成要素を横断的に変更を加える必要がある．

1.3　機械証明の拡張

本節では機械証明を行うときの例を用いて機械証明では拡張が困難であることを説明する．

前節では差分として定義することで拡張をした．差分として定義するとは再利用が可能な定義をするということである．機械証明で再利用が可能な構文定義や評価・型付け関数定義，型安全性の証明は素朴には不可能であることを説明する．

先ほど例示した単純型付ラムダ計算言語の Lambda，自然数と加算の言語 Arith を実際に合成することで再利用可能な定義が難しく拡張が困難であることを述べる．それぞれの言語の構文を表現するデータ型を図 1.1 に，評価規則を Coq の関数として定義したものが図 1.2 である．評価規則や型付け規則を関数として定義したものを本研究では以降それぞれ評価関数，型付け関数と呼ぶ．

```
1 data Arith = num Int | add Arith Arith
2 data vArith = vnum Int
3 data Lambda = var Id | lam Id T Lambda | app Lambda Lambda
4 data vLambda = vlam Env Id T Lambda
```

図 1.1: 素朴な方法で定義した加算と単純型付ラムダ計算をそれぞれ表すデータ型

Arith と Lambda 両方の構文を含む新しい言語 AL の構文を素朴に定義すると図 1.3 のようになる．ここで注意すべきなのは add の部分項や app の部分項の再帰が Arith のみ，Lambda のみに留まらず Arith と Lambda を合成したデータ型全体を参照していることである．そのため Arith と Lambda それぞれの構文を表現したデータ型の定義をそのまま使うことができず，必ず再帰部の書き換えを行うことが必要である．元の定義を書き換えずに再帰部の定義を変更するというこ

```
evalArith :: Arith -> vArith
evalArith (num n) = vnum n
evalArith (add e1 e2) = case (evalArith e1, evalArith e2) of
    (vnum n1, vnum n2) -> vnum (n1 + n2)

evalLambda :: Env vLambda -> Lambda -> vLambda
evalLambda env (var x) = case lookup env x of
    Some v -> v
evalLambda env (lam x dt e) = vlam env x dt e
evalLambda env (app e1 e2) = case evalLambda env e1 of
    vlam env' x e -> let v2 := evalLambda env e2 in
    evalLambda (update env' x v2) e
```

図 1.2: 素朴な方法で定義したそれぞれの評価関数

```
data AL = num Int | add AL AL
  | var Id | lam Id T AL | app AL
data vAL = vnum Int | vlam Env Id T AL
```

図 1.3: Arith と Lambda を組み合わせたデータ型

とはできない．従って，Arith と Lambda を合成した言語の定義のために構文を定義するデータ型を新たに定義し直すことが必須である．

Arith と Lambda を合成した言語に対する評価関数は図 1.4 と表現できる．これについても部分項を評価するための再帰関数が Arith と Lambda を合成したデータ型に対する評価関数にする必要があるため，元の評価関数を再利用することはできない．

更に環境の取り扱いが問題となる，Arith では評価の際に環境が必要なかったが Lambda には必要である．従って合成言語の評価には必要になる．従って Arith 由来の構文の評価について，環境を受け取って再帰呼びの際に渡す必要があるため，再帰部に関する再利用が実現したとしてもまだ不十分である．

```
evalAL :: Env   vAL -> AL -> vAL
evalAL env (num n) = vnum n
evalAL env (add e1 e2) = case (evalAL env e1, evalAL env e2) of
    (vnum n1, vnum n2) -> vnum (n1 + n2)
evalAL env (var x) = case lookup env x of
    Some v -> v
evalAL env (lam x dt e) = vlam env x dt e
evalAL env (app e1 e2) = case evalAL env e1 of
    vlam env' x e -> let v2 := evalAL env e2 in
        evalAL (update env' x v2) e
```

図 1.4: Arith と Lambda を組み合わせたデータ型に対する評価関数

1.4 既存研究

本節ではこれまで為されてきた拡張に対する既存の取り組みを紹介し，本研究の方針と立ち位置を述べる．

プログラミング言語の拡張については古くから the expression problem[32] として知られていた．その解決策も現在までに複数発表されている．関数型言語に対する拡張としては W. Swierstra の Data-Types à la Carte(DTC)[29] があり，この研究では T. Sheard の two-level types[28] の手法を応用してデータ構造とその評価・型付け規則を拡張可能にする枠組みを提供することで the expression problem の解決としている．オブジェクト指向言語に対する言語拡張としては，object algebra を利用した B. Oliveira の Extensibility for the Masses や [23] や generics を用いて拡張可能にする M. Torgersen の The Expression Problem Revisited[30] がある．

近年プログラミング言語の型安全性を機械証明で与える研究が盛んである．B. Delaware らの Product Lines of Theorems[15] は拡張の可能性をあらかじめ制限することで A. Igarashi の提案する Featherweight Java[18] のモジュラーな構築を可能にしている．K. Nakata の Trace-Based Coinductive Operational Semantics for While[21] はトレースという仕組みを使い，While 言語の型安全性の証明と式の等価性の証明を小ステップ意味論と大ステップ意味論の両方で与えたものである．

言語拡張という視点で the expression problem に対する解決法であるを機械証明の型安全性の証明に応用するのは自然である．DTC の手法を構文や評価・型付け規則に応用して，Coq で証明を拡張する手法を提案した B. Delaware らの Meta-Theory à la Carte(MTC)[16] や，Agda で証明を拡張する手法を提案した C. Schwaab の Modular Type-Safety Proofs in Agda[27] が一例である．

本研究は MTC を応用し，操作的意味論で評価規則を定義し型安全性を証明する．MTC は表示的意味論で議論がされている，また本研究では証明支援器 Coq を使って機械証明を与える．C. Schwaab の手法は Agda と Coq と設計の違いから定義に利用することができない．Coq は Agda と比較して型クラスの仕組みがあることからモジュラーな拡張のために向いていると考える．

MTC の応用として，B. Delaware らの Modular Monadic Meta-Theory(3MT)[17] がある．この研究は MTC が表示的意味論で実装していることを利用してモナドを使った意味論の拡張方法と型安全性の証明を提案した研究である．P. Torrini の Reasoning about modular datatypes with Mendler induction[31] は項の構造として相互再帰をする手法を提案したものである．S. Keuchel の Generic Datatypes à La Carte は MTC が抱えていた Coq の一貫性を一部崩してしまっているという問題を M. Abbott の Containers: Constructing strictly positive types[1] の手法で解決したものである．

操作的意味論でも意味論の拡張を可能にする手法として P. Mosses の Modular Structual Operational Semantics(MSOS)[20] がある．これは圏論の積圏と恒等射を利用し，小ステップ操作的意味論で意味論の拡張する方法を提案する手法である．C. Poulsen の Deriving Pretty-Big-Step Semantics from Small-Step Semantics[6] は MSOS で書かれた小ステップ操作的意味論の拡張可

能な定義を A. Charguéraud の Pretty-big-step 意味論 [12] に書き換える手法を提案している．

本研究では操作的意味論に pretty-big-step 意味論を採用する．理由として第一に MSOS の小ステップ操作的意味論は拡張可能であるが書き換えられた pretty-big-step 意味論が拡張可能であるかはわかっていない．しかしながら書き換えられた pretty-big-step 意味論は拡張可能な定義が持つべき性質を持っているように見える．第二に部分項の評価順が不定である大ステップ意味論と比較して，部分項の評価順の決まっている pretty-big-step 意味論では参照等の評価順が重要になる拡張が自然にできると考えられるからである．第三に MTC のフレームワークの表示的意味論において，評価規則は項から値領域に移す．Pretty-big-step 意味論も同様に項から値に移すため相性が良いと考えられる．大ステップ意味論から引き継いでしまっている不停止のプログラムと型の付かないプログラムの区別は，J. Reynolds の Definitional Interpreter [26] の評価木の高さを制限し，証明の際に任意の高さの停止するプログラムについて言及することで N. Amin[3] によって解決済である．

1.5 本研究の提案

本研究では構文，評価・型付け規則，型安全性の証明に MTC の proof algebra の手法を，意味論の拡張に P. Mosses の Modular Structual Operational Semantics(MSOS) [20] の手法を応用する．操作的意味論には A. Charguéraud の Pretty-Big-Step 意味論 [12] を採用する．構文定義や型付け規則の定義は操作的意味論は表示的意味論とは独立であるが，評価規則に関しては pretty-big-step 意味論として MTC の枠組みで定義する手法を本研究で提案する．証明についても意味論が変わるため一部表示的意味論と違うことを証明する必要があるので値と型の関係に工夫を行うことで証明を可能にする手法を提案する．事例研究として自然数と加算言語と単純型付ラムダ計算言語をそれぞれ定義・証明することで，型安全性の証明がモジュラーに書くために必要なことを示す．貢献として操作的意味論にすることによって MTC のサンプルコードと比較して，型安全性の証明を拡張可能に定義するために必要な行数が削減できたことを示す．

1.6 本論文の構成

本論文の構成は以下の通りである．2 章では本論文を読むにあたり必要な前提知識を定義する．MTC のフレームワーク，pretty-big-step 意味論，そして MSOS についてが定義されている．3 章では MSOS の手法で定義した pretty-big-step 意味論を MTC の枠組みで拡張可能にするに当たって生じる問題とその解決法を議論する．4 章では実際に本研究の手法で単純型付単純型付ラムダ計算と自然数と加算言語の，拡張可能な型安全性の証明を構成方法を提案する．5 章では関連研究について述べる．6 章では結論として本研究の貢献と新たに明らかになった問題を述べる．

第2章　定義

本章では本研究を理解する上で必要となる前提を定義する．

2.1　構文と評価・型付け規則の拡張

構文や評価・型付け規則の拡張は MTC の手法を応用する．この手法では項や値，型といった構文を Functor と呼ばれる穴あきのデータ構造で定義する．前章で例に用いたデータ型を Functor で定義するには表 2.1 のようにする．

```
1 data Arith a = num nat | add a a
2 data vArith a = vnum nat
```

図 2.1: Functor で定義した Arith

ここで a は型変数であり，加算を表すデータ型の部分項が不定であることがわかる．この型変数に unit を与えた型 `Arith unit` は自然数を表現する葉のみを表現できるデータ型である．`Arith` にこの `Arith unit` を与えた `Arith (Arith unit)` は高さ 1 までの 2 分木を表現できるデータ型になる．このように任意の高さの二分木を表現できるようにするためには型変数として開けてある穴に自身のデータ型を埋めることで実現ができる．この「穴を自身でを埋める」ためには不動点を取る操作をする．つまり，`Fix Arith = Arith (Fix Arith)` となるような不動点演算子 `Fix` を用いることで型変数に自身を埋めた，任意の高さの二分木を表現できるデータ型を構築できる．不動点を取る操作と不動点を取ることで構成されたデータ型に対する操作は以下のように与えられる．具体的な実装は省略する．

```
1 Fix F = F (Fix F)
2 in_t :: F (Fix F) -> Fix F
3 out_t :: Fix F -> F (Fix F)
```

このように構築したデータ型に対する評価規則や型付け規則を定義するためには Algebra と呼ばれる型を持つ規則を定義し，再帰的なデータ構造に対して再帰的に適用する．Algebra は以下のような型で表現される．

```
1 type Algebra F a = F a -> a
2 type r => MAlgebra F a = (r -> a) -> F r -> a
```

図 2.2: Algebra の定義

F は任意の Functor，a は遷移先の型である．値を表すデータ型 Val や型を表すデータ型 Typ を用いて，Arith に対する評価関数ならば Algebra Arith Val，型付け関数ならば Algebra Arith Typ のように書ける．本研究では algebra を使って表現した評価関数を評価 algebra，型付け関数を型付け algebra と定義する．

併記されている MAlgebra は mendler-style の Algebra と呼ばれるデータ型で，評価・型付け対象の f の部分項の型が違うのが特徴である．通常の Algebra であれば F a を a に移すので部分項が既に評価済み，あるいは型付け済みのデータを受け取り，評価や型付けをする方法を表すが，MAlgebra は部分項の型の指定がないため評価や型付けのされていないデータを受け取って評価や型付けする方法を表す．このとき部分項の型 r を a に移す「部分項評価（型付け）関数」と呼ぶべき関数を受け取る必要がある．

評価規則 MAlgebra Arith Val は上の定義に照らし合わせると ∀r.(r → Val) → Arith r → Val と書ける．この第一引数は部分項評価関数，第二引数は部分項の型が部分項評価関数の定義域である評価対象である．これを踏まえて MAlgebra Arith Val を定義する (図 2.3).

```
1 evalArith :: MAlgebra Arith Val
2 evalArith rec (num n) = inj (vnum n)
3 evalArith rec (e1 + e2) = case (prj (rec e1), prj (rec e2)) of
4     (Some (vnum n1), Some (vnum n2)) -> inj (vnum (n1 + n2))
```

図 2.3: Arith の評価 MAlgebra

特筆すべきは値の構文を拡張可能にしている inj 関数と prj 関数である．表 1.4 で定義したように，値についても項と同様に拡張として定義されているはずである．従って num を Val に埋め込む必要があり，その役割を果たすのが inj 関数である．逆に足し算を計算するために部分項評価関数によって飛ばされた Val から num を取り出す役割を果たすのが prj 関数である．これらの関数が正しく働くためには vArith ⊂ Val のような関係が暗黙的に定義されている必要がある．この包含関係は次節で定義する．またこの関係から inj は全域関数であるが prj は部分関数となることが明らかであり，prj (rec e1) が Some (num n1) に移ることがわかる．このように埋め込み・取り出しの操作を明示的に評価関数に組み込むことで値の構文の拡張が可能になる．

2.2　定義の合成

定義を合成するために先ほど包含関係として紹介したデータ型の順序関係を定義する．データ型 F と G が存在し，F が G に内包されているとき F :<:　G と書くことにする．前章で挙げた例を使うと，Arith :<:　AL のように記述することになる．そこでデータ型の結合演算子:+:を導入する．データ型の結合には一般的な和型を使う．構文の結合の定義は以下の通りである（図 2.4）.

また F と G が順序関係にあるとき，以下（図 2.5）の演算が使えるものとする．

inj は小さい方から大きい方に埋め込む操作，prj は大きい方から小さい方への射影である．

またこの結合によって以下の順序関係が自然に導入できる．

```
data a => (:+:) F G a = inl (F a) | inr (G a)
```

図 2.4: 和型の定義

```
class a => (:<:) F G a where
    inj :: F a -> G a
    prj :: G a -> option (F a)
```

図 2.5: 順序関係

- F :<: F
- F :<: G ⟶ F :<: (G :+: H)
- F :<: H ⟶ F :<: (G :+: H)

結合したデータ型に対する操作も以下のように単純に合成できる．

```
class F, G, (falg :: MAlgebra F A)(galg :: MAlgebra G A) =>
   alg_plus :: MAlgebra (F :+: G) A where
   alg_plus rec (inl fa) = falg rec fa
   alg_plus rec (inr ga) = galg rec ga
```

この操作の結合により，図 2.3 と同様のスタイルで単純型付ラムダ計算についても評価 algebra を evalLambda として与えられれば，2 つの評価関数を結合し，AL，つまり Arith :+: Lambda に対する操作を以下のように自動的に生成できる．

```
evalArithBool :: MAlgebra (Arith :+: Bool) Val
evalArithBool rec (inl a) = evalArith rec a
evalArithBool rec (inr a) = evalLambda rec a
```

また結合したデータ型の Algebra に対して well-formed であるという性質がある．これは Algebra の定義域が disjoint であるということであり，型安全性を証明する際に必要な性質である．Well-formed であるということは以下の性質が成り立つということである．

```
F :<: G => falg rec e = galg rec (inj e)
```

上式の e の型は F r，rec は r -> a，falg は MAlgebra F A，galg は MAlgebra G A である．従って上式の意味するところは，F r 型である e を F に対する Algebra で操作しても G に埋め込んだ先の Algebra で操作しても結果が変わらないということである．

2.3 型安全性の証明の拡張

証明の拡張には proof algebra と呼ばれる手法を応用する．それぞれの構成要素について項の構造に関する帰納法の場合分けに相当する証明をすることで，それらを組み合わせた言語に対する証明を与えるという手法である．

拡張されたデータ型を F と置き，以下を仮定する．

$$\mathtt{F} = \mathtt{G}_1 : + : : + : \mathtt{G}_2 : + : \cdots : + : \mathtt{G}_n$$
$$\text{かつ } \forall e : \mathtt{F\ a}, \exists e' : \mathtt{G}_i\ \mathtt{a}, e = \mathtt{inj}\ e' \text{の } i \text{ が一意}$$

この仮定は F が G_1，G_2，$\cdots$，G_n の和でできており，それぞれの G_i は共通部分が存在しないということを言っている．このとき proof algebra は以下のように定義される．

```
1 Soundness :: (Fix F) -> Prop
2 G :<: F => inject :: Algebra G (Fix F)
3     inject e = in_t (inj e)
4
5 FPAlgebra G = forall e: Fix G, Soundness (fold inject e)
```

図 2.6: Proof algebra の定義

図 2.6 の 5 行目に定義した FPAlgebra が意味するところとは，

どんな Fix G 型の e に対しても，fold inject e は *Soundness* を満たす

というものである．1 行目の Soundness は型安全性の命題である．具体的な定義は 4 章で与える．2，3 行目の inject 関数は G (Fix F) 型の値を Fix F 型に移す関数である．5 行目の FPAlgebra が F の証明をモジュラーに与えるためにそれぞれの G_i について与えるべき命題となる．各 G_i について FPAlgebra $\mathtt{G}_i$ が与えられている場合に任意の Fix F 型の値 e について *Soundness* e が成立することを示す．

F に対する仮定より，任意の Fix F 型の値 e に対して一意に決まる i が存在して，inject a が e と等しくなるような $\mathtt{G}_i$ (Fix F) 型の値 a が存在する．仮定より各 i について，FPAlgebra G が証明済みなので FPAlgebra の定義より以下が言える．

$$\forall e' : \mathtt{Fix\ G}_i, \mathtt{Soundness}\ (\mathtt{fold\ inject}\ e')$$

従って，どんな a についてもある $\mathtt{e}'$ が存在し，inject a=fold inject $\mathtt{e}'$ となることを仮定すれば，任意の Fix F 型の値に対して P が成立することが示せる．この仮定の証明は省略する．

2.4 Pretty-big-step

本節では大ステップの考えを基に発展させた pretty-big-step 意味論 [12] を紹介する．

Pretty-big-step では中間項と呼ばれる項を導入し，必ず 1 か所の部分項に着目する点が大ステップの意味論と違う点であり，従って評価順が決定される．大ステップ，小ステップ，pretty-big-step で自然数と加算言語の意味論はそれぞれ図 2.7，図 2.8，図 2.9 のように表される．

$t \Longrightarrow v$: 大ステップの評価規則

$$\frac{}{\texttt{num}\ n \Longrightarrow \texttt{vnum}\ n} \qquad \frac{t_1 \Longrightarrow \texttt{vnum}\ n_1 \quad t_2 \Longrightarrow \texttt{vnum}\ n_2 \quad n_1 + n_2 = n}{\texttt{add}\ t_1\ t_2 \Longrightarrow \texttt{vnum}\ n}$$

図 2.7: 大ステップ意味論

$t \longrightarrow t'$: 小ステップの評価規則

$$\frac{t_1 \longrightarrow t_1'}{\texttt{add}\ t_1\ t_2 \longrightarrow \texttt{add}\ t_1'\ t_2} \qquad \frac{t_2 \longrightarrow t_2'}{\texttt{add}\ v_1\ t_2 \longrightarrow \texttt{add}\ v_1\ t_2'}$$

$$\frac{n_1 + n_2 = n}{\texttt{add}\ (\texttt{num}\ n_1)\ (\texttt{num}\ n_2) \longrightarrow \texttt{num}\ n}$$

図 2.8: 小ステップ意味論

中間項は図 2.9 の PB-Add に出現する `add1`，PB-Add1 に出現する `add1` 及び `add2`，そして PB-Add2 に出現する `add2` である．構文規則にこれらは存在せず，評価のために生成され，評価の途中にだけ現れる構造である．

それぞれの特徴を簡単にまとめると，大ステップと pretty-big-step は項から値に移しているのに対し，小ステップでは項から項に移している．また小ステップと pretty-big-step では `add` の左側の中間項から評価を進めることが決定的なのに対し，大ステップでは依存関係がないため左右のどちらから評価を進めるかが非決定的である．

2.5 小ステップ意味論の MSOS

本節では小ステップ操作的意味論を拡張可能にする MSOS について説明する．MSOS ではそれぞれの意味論の評価に必要な構成要素をラベルと呼び，評価規則を表す矢印記号の両端に下付きで表記することで評価前後のラベルを表現する．例えば $t_\sigma \longrightarrow_{\sigma'} t'$ と書いた場合は t を評価するのに σ を受け取り，評価後の t' と共に σ' を返す，という意味となる．ラベルはレコード型であり，$\sigma.env$ と書くことでラベルから環境を取り出して読むことができ，$\sigma.env = x$ と書くことでラベルの環境に対して x を書き込むことができるとする．単純型付ラムダ計算のラベル付き小ステップ意味論を図 2.10 に示す．

$t \Downarrow v$: Pretty-big-step の評価規則

$$\frac{}{\texttt{num}\ n \Downarrow \texttt{vnum}\ n}\ (\mathrm{PB-Num})$$

$$\frac{e_1 \Downarrow v_1 \quad \texttt{add1}\ v_1\ e_2 \Downarrow v}{\texttt{add}\ e_1\ e_2 \Downarrow v}\ (\mathrm{PB-Add}) \qquad \frac{e_2 \Downarrow v_2 \quad \texttt{add2}\ v_1\ v_2 \Downarrow v}{\texttt{add1}\ v_1\ e_2 \Downarrow v}\ (\mathrm{PB-Add1})$$

$$\frac{n_1 + n_2 = n}{\texttt{add2}\ (\texttt{vnum}\ n_1)\ (\texttt{vnum}\ n_2) \Downarrow \texttt{vnum}\ n}\ (\mathrm{PB-Add2})$$

図 2.9: Pretty-big-step 意味論

$t_\sigma \longrightarrow_{\sigma'} t'$: ラベル付き小ステップ意味論

$$\frac{\sigma.env(x) = v}{\texttt{var}\ x_\sigma \longrightarrow_\sigma v}\ (\mathrm{SL-Var}) \qquad \frac{}{\texttt{lam}\ x\ T\ t_\sigma \longrightarrow_\sigma \texttt{lam}\ x\ T\ t\ \sigma.env}\ (\mathrm{SL-Lam})$$

$$\frac{t_1\ _\sigma \longrightarrow_{\sigma'} t_1'}{\texttt{app}\ t_1\ t_2\ _\sigma \longrightarrow_{\sigma'} \texttt{app}\ t_1'\ t_2}\ (\mathrm{SL-App1}) \qquad \frac{t_2\ _\sigma \longrightarrow_{\sigma'} t_2'}{\texttt{app}\ v_1\ t_2\ _\sigma \longrightarrow_{\sigma'} \texttt{app}\ v_1\ t_2'}\ (\mathrm{SL-App2})$$

$$\frac{t_{\sigma.env=env[x\mapsto v_2]} \longrightarrow_{\sigma'} t'}{\texttt{app}\ (\texttt{lam}\ x\ T\ t\ env)\ v_2\ _\sigma \longrightarrow_{\sigma'.env=\sigma.env} (\texttt{lam}\ x\ T\ t'\ env)}\ (\mathrm{SL-App3})$$

$$\frac{}{\texttt{app}\ (\texttt{lam}\ x\ T\ v\ env)\ v_2\ _\sigma \longrightarrow_\sigma v}\ (\mathrm{SL-App4})$$

図 2.10: ラベル付き小ステップ意味論

変数参照に対する規則 SL-Var では評価時に受け取るラベルから環境を取り出し変数を参照している．ラムダ抽象に対する規則 SL-Lam ではラムダ抽象に評価時に受け取るラベルから環境を取り出しクロージャにしている．ラムダ適用に対する規則ではまずオペレータを評価 (SL-App1) し，次にオペランドを評価 (SL-App2) する．両方が値になったらオペレータの中身を評価 (SL-App3) する．SL-App3 でオペレータのクロージャの中身 t を評価する際には受け取ったラベルの環境に対し，クロージャが持つ環境の x に v_2 を束縛したものを書き込む．t を評価して返ってきたラベルの環境を返す前に評価前の環境で上書きしてから返す．オペレータのクロージャの中身が値まで評価できたら中身を取り出す (SL-App4).

2.6 Pretty-big-step 意味論の MSOS

B. Casper によって小ステップ意味論で定義された MSOS を pretty-big-step 意味論に変換する手法 [6] が発表されている．この研究で変換された自然数と加算言語と単純型付ラムダ計算に対するラベル付きの pretty-big-step 意味論はそれぞれ図 2.11, 図 2.12 と表現できる．

$t_\sigma \Downarrow_{\sigma'} v$: ラベル付き pretty-big-step 意味論

$$\frac{}{\texttt{num}\ n\ _\sigma \Downarrow_\sigma \texttt{vnum}\ n}\ (\mathrm{PBL-Num})$$

$$\frac{t_1\ _\sigma \Downarrow_{\sigma'} v_1 \quad \texttt{add1}\ v_1\ t_2\ _{\sigma'} \Downarrow_{\sigma''} v}{\texttt{add}\ t_1\ t_2\ _\sigma \Downarrow_{\sigma''} v}\ (\mathrm{PBL-Add}) \qquad \frac{t_2\ _\sigma \Downarrow_{\sigma'} v_2 \quad \texttt{add2}\ v_1\ v_2\ _{\sigma'} \Downarrow_{\sigma''} v}{\texttt{add1}\ v_1\ t_2\ _\sigma \Downarrow_{\sigma''} v}\ (\mathrm{PBL-Add1})$$

$$\frac{n_1 + n_2 = n}{\texttt{add2}\ (\texttt{vnum}\ n_1)\ (\texttt{vnum}\ n_2)\ _\sigma \Downarrow_\sigma \texttt{vnum}\ n}\ (\mathrm{PBL-Add2})$$

図 2.11: 自然数と加算言語のラベル付き pretty-big-step 意味論

$t_\sigma \Downarrow_{\sigma'} v$: ラベル付き pretty-big-step 意味論

$$\frac{\sigma.env(x) = v}{\texttt{var}\ x\ _\sigma \Downarrow_\sigma v}\ (\text{PBL} - \text{Var}) \qquad \frac{}{\texttt{lam}\ x\ T\ t\ _\sigma \Downarrow_\sigma \texttt{vlam}\ x\ T\ t\ \sigma.env}\ (\text{PBL} - \text{Lam})$$

$$\frac{t_1\ _\sigma \Downarrow_{\sigma'} v_1 \quad \texttt{app1}\ v_1\ t_2\ _{\sigma'} \Downarrow_{\sigma''} v}{\texttt{app}\ t_1\ t_2\ _\sigma \Downarrow_{\sigma''} v}\ (\text{PBL} - \text{App}) \qquad \frac{t_2\ _\sigma \Downarrow_{\sigma'} v_2 \quad \texttt{app2}\ v_1\ v_2\ _{\sigma'} \Downarrow_{\sigma''} v}{\texttt{app1}\ v_1\ t_2\ _\sigma \Downarrow_{\sigma''} v}\ (\text{PBL} - \text{App1})$$

$$\frac{t\ _{\sigma.env=(env[x\mapsto v_2])} \Downarrow_{\sigma'} v}{\texttt{app2}\ (\texttt{vlam}\ x\ T\ t\ env)\ v_2\ _\sigma \Downarrow_{\sigma'.env=\sigma.env} v}\ (\text{PBL} - \text{App2})$$

図 2.12: 単純型付ラムダ計算のラベル付き pretty-big-step 意味論

第3章　問題点と解決法

本章ではMSOSの手法で定義したpretty-big-step意味論をMTCで拡張可能にするにあたって生じる問題点とその解決法を述べる．

3.1　問題点

3.1.1　Pretty-big-stepの中間項

MTCのフレームワークでpretty-big-stepの中間項を導入するには項の部分に穴を開けて以下のようにFunctorを定義する．但しValは値のデータ型である

```
1 data iArith a = add1 Val a | add2 Val Val
2 data iLambda a = app1 Val a | app2 Val Val
```

評価関数や型付け関数を従来のように`Arith`に対してでなく，`Arith :+: iArith`に対して定義すれば良さそうであるが，2つ問題が発生する．一つは評価関数の再帰について，もう一つは型付け関数と型安全性の証明についてである．

評価関数の再帰

MTCのフレームワークを単純にpretty-big-stepに対して用いると評価関数は以下のようになる．

```
1 evalArithPB_first :: MAlgebra (Arith :+: iArith) Val
2 evalArithPB_first rec (inl (num n)) = inj (vnum n)
3 evalArithPB_first rec (inl (add e1 e2)) =
4     rec (add1 (rec e1) e2)
5 evalArithPB_first rec (inr (add1 v1 e2)) =
6     rec (add2 v1 (rec e2))
7 evalArithPB_first rec (inr (add2 v1 v2)) = case (prj v1, prj v2) of
8     (Some (vnum n1), Some (vnum n2)) -> inj (vnum (n1 + n2))
```

ところがこの定義では型エラーが発生する．なぜならばMAlgebraの定義（図2.2）より，部分項評価関数である`rec`の型は$r \to Val$であり，このrはArithやiArithの部分項の型に等しい．この型変数rはiArithとは無関係である．従って`rec`を使って，`add1`や`add2`といった中間項の評価を進めることができない．

型付け規則と型安全性

中間項に対して型付け規則を定義する際にも問題がある．なぜならば MTC のフレームワークでは項の型付けは関数として定義するのに対し，値に対する型付けは関係として定義するためである．Coq では関数定義の中に関係を含めることはできないが，逆の関係定義に関数を含めることはできる．そのため部分構造に項と値の両方を持つ中間項に対しては必然的に関係的に型付けを定義する必要がある．すると関数定義に関係を利用できないことから，中間項についての型付け結果を使うためには項に対しても型付けを関係で定義する必要がある．MTC のフレームワークにおいては関係で定義された型付け規則は，一意性を明示的に証明しなければ型安全性の証明ができない．一方で関数で定義された型付け規則は一意性が自明なので証明する必要はない．従ってできる限り関数で定義することが記述量を減らす観点から望ましいため，項についての型付け規則を関数的に与えながら型安全性の証明をするという課題が存在する．

3.1.2　項の構造に関する帰納法

Pretty-big-step 意味論における型安全性の定理は定理 (3.1.1) になる．

定理 3.1.1 (ラベル付き pretty-big-step 意味論における型安全性)

$$(\Gamma \vdash t : T \ \wedge \ t \ {}_{\sigma} \Downarrow_{\sigma'} v \ \wedge \ \Gamma \models \sigma) \Rightarrow (\Gamma \vdash v : T \ \wedge \ \Gamma \models \sigma')$$

単純型付ラムダ計算の型安全性の証明を proof algebra により拡張可能にするために項の構造に関する帰納法で与えることを試みる．項がラムダ適用の場合の帰納法の仮定は以下 (3.1)(3.2) である．

$$(\Gamma \vdash t_1 : T_1 \ \wedge \ t_{1\sigma} \Downarrow_{\sigma'} v_1 \ \wedge \ \Gamma \models \sigma) \Rightarrow (\Gamma \vdash v_1 : T_1 \ \wedge \ \Gamma \models \sigma') \tag{3.1}$$

$$(\Gamma \vdash t_2 : T_2 \ \wedge \ t_{2\sigma} \Downarrow_{\sigma'} v_2 \ \wedge \ \Gamma \models \sigma) \Rightarrow (\Gamma \vdash v_2 : T_2 \ \wedge \ \Gamma \models \sigma') \tag{3.2}$$

また証明のために与えられる前提は以下．

$$\Gamma \vdash t_1 \ t_2 : T \tag{3.3}$$

$$t_1 \ t_{2\sigma} \Downarrow_{\sigma'} v \tag{3.4}$$

$$\Gamma \models \sigma \tag{3.5}$$

これらが与えられた上で以下．

$$\Gamma \vdash v : T \ \wedge \ \Gamma \models \sigma'$$

を証明する．まず (3.3) より型付け規則 (T-App) から以下は明らか．

$$\Gamma \vdash t_1 : T_1 \to T \tag{3.6}$$

$$\Gamma \vdash t_2 : T_1 \tag{3.7}$$

また (3.4) より評価規則 (PBL-App) から以下もわかる．

$$t_{1\sigma} \Downarrow_{\sigma_1} \lambda x : T_1.t \tag{3.8}$$

$$t_{2\sigma_1} \Downarrow_{\sigma_2} v_2 \tag{3.9}$$

よって帰納法の仮定を使って (3.1)(3.5)(3.6)(3.8) より (3.10)(3.11) が，(3.2)(3.7)(3.9)(3.11) より (3.12)(3.13) がわかる．

$$\Gamma \vdash \lambda x : T_1.t : T_1 \to T \tag{3.10}$$

$$\Gamma \models \sigma_1 \tag{3.11}$$

$$\Gamma \vdash v_2 : T_1 \tag{3.12}$$

$$\Gamma \models \sigma_2 \tag{3.13}$$

ところが証明はこれ以上進めることができない．なぜならばラムダ抽象の中身 (e) に対する帰納法の仮定がないので，ラムダ抽象の中を評価した結果に型が付くことを示すことができないからである．

型安全性の証明は通常であれば [25] 評価規則に関する帰納法による．評価規則に関する帰納法であれば評価木に現れる評価関係に対して帰納法の仮定が与えられる．ラムダ適用の評価であればオペレータの評価，オペランドの評価，そしてラムダ抽象の中身それぞれに対して帰納法の仮定を使え，ラムダ抽象の評価では評価が起こらないので帰納法の仮定は与えられない．

項の構造に関する帰納法では部分項に対して帰納法の仮定が与えられる．ラムダ適用の評価ではオペレータとオペランドである．ラムダ抽象の中身はラムダ適用の部分項でないため帰納法の仮定が使えない．一方ラムダ抽象の評価ではラムダ抽象の中身に対して帰納法の仮定が使える．しかし型安全性のラムダ抽象の場合の証明にラムダ抽象の中身の帰納法の仮定は不必要である．

このようにラムダ抽象の中身に対する帰納法の仮定の受け取れるタイミングがちぐはぐになってしまっているのを解決しなければ証明ができないのである．

3.1.3　ラベルの形式化

MSOS では意味論を拡張するためにラベルというデータ構造を利用する．ラベルとはレコード型として定義されており，それぞれの意味論が必要とする情報をそれぞれのレコードに保持する．型安全性の証明のためには他の意味論のためのデータを無視していることを保証する必要がある．つまり単純型付ラムダ計算で参照のためのデータを参照したり書き換えないことを保証する必要がある．レコードには型クラスを使って操作のためのインターフェースを提供する．レコード型のモジュラーな拡張は難しい上に，評価関数にレコード型としてラベルを渡すと他の意味論のためのデータを無視していることが保証できない．型クラスを使って操作のインターフェースを提供する方法であれば，他の意味論のためのデータに関与しないようにすることは容易である．また型クラスを使った実装であればデータの実体には興味がなくなるので，モジュラーな拡張が可能になる．単純型付ラムダ計算と参照を合成した言語に対するラベルのデータ型は，環境に対する操作を定義した型クラスと参照に対する型クラスの両方の実装があるものであれば何でもよいことになる．具体的には以下の Env が環境に対する操作，Ref が参照に対する操作を表す型クラスである．

```
1 class Env C a where
2     lookup :: C a -> id -> option a
3     update :: C a -> id -> a -> C a
4     remove :: C a -> id -> C a
5
6 class Ref C a where
7     create :: C a -> a -> (C a, id)
8     ref :: C a -> id -> a -> C a
9     deref :: C a -> id -> option a
```

これらの型クラスを利用して，次の 1 行目に示す label を宣言することで Env と Ref の両方の操作が可能なデータ型を宣言できる（ref に対するインスタンスは省略）．

```
1  data label a b = {env : [id * a], ref : [b]}
2  lookup' :: [id * a] -> id -> option a
3  lookup' nil x = None
4  lookup' ([y,e]:t) x = if x == y then Some e else lookup' t x
5  update' :: [id * a] -> id -> a -> [id * a]
6  update' lst x e = [x,e] : lst
7  remove' :: [id * a] -> id -> [id * a]
8  remove' nil x = nil
9  remove' ([y,e]:t) x = if x == y then t else [y,e] : remove' t x
10 instance Env id (a => label a b) a where
11     lookup l x = {env = lookup' l.env x, ref = l.ref}
12     update l x e = {env = update' l.env x e, ref = l.ref}
13     remove l x = {env = remove' l.env x, ref = l.ref}
```

label は env フィールドと ref フィールドからなるレコード型である．このラベルを受け取る評価のための Algebra は図 3.1 のようになる．

```
1  L => evalArithL :: MAlgebra Arith (L -> Val)
2  evalArithL rec (num n) l = (inj (vnum n), l)
3  evalArithL rec (add e1 e2) l = case (prj (rec e1 l), prj (rec e2 l)) of
4      (Some (vnum n1), Some (vnum n2)) -> inj (vnum (n1 + n2))
5
6  L, Env id L Val => evalLambdaL :: MAlgebra Lambda (L -> Val)
7  evalLambdaL rec (var x) l = case (lookup l x) of
8      Some v => v
9  evalLambdaL rec (lam x e) l = inj (vlam l x e)
10 evalLambdaL rec (app e1 e2) l = case prj (rec e1 l) of
11     Some (vlam l' x e) -> let v := rec e2 l in
12     rec e (update l' x v)
```

図 3.1: ラベルを用いた評価関数

1 行目，MAlgebra の第二引数が `L -> Val` になっているのはラベルを新たに受け取ることができるためである．MAlgebra の定義は 2.2 の通りなので `evalArithL` の型は以下のようになる．

```
1  L, r => (r -> L -> Val) -> Arith r -> L -> Val
```

これによりラベルを次々と受け取るようにすることで単純型付ラムダ計算の環境のような要素は実装できるが，参照のように読み書きが可能なものは実装できない．これは評価順が一意でないと正しく実装できない．

3.2 解決法

3.2.1 評価関数の再帰

部分項評価関数 `rec` が中間項を評価できない問題に対し，`rec` を使わない再帰呼び出しをすることで解決した．Coq で再帰呼び出しをする際には停止性を保証する必要がある．しかしこれは評価木の高さを制限したことで解決した．

制限した高さで評価できなかったことを表すデータを追加した以下の Val' を使って，

```
1  data Val' = TIMEOUT | Completed Val
```

再帰呼び出しを使い pretty-big-step で実装した Arith の評価関数は以下のようになる．

```
1  evalArithPB :: MAlgebra (Arith :+: iArith) (nat -> Val')
2  evalArithPB rec _ 0 = TIMEOUT
3  evalArithPB rec (inl (num n)) (m+1) = Completed (inj (vnum n))
4  evalArithPB rec (inl (add e1 e2)) (m+1) =
5      evalArithPB rec (inr (add1 (rec e1 m) e2)) m
6  evalArithPB rec (inr (add1 v1 e2)) (m+1) =
7      evalArithPB rec (inr (add2 v1 (rec e2 m))) m
8  evalArithPB rec (inr (add2 v1 v2)) (m+1) =
9      case (v1, v2) of
10     (Completed v1', Completed v2') -> case (prj v1', prj v2') of
```

```
11        (Some (vnum n1), Some (vnum n2)) -> Completed (inj (vnum (n1 + n2)))
```

2行目が制限された高さで評価が終わらない場合の分岐である．5, 7行目でrecを使わずevalArithPB自身を再帰呼び出しすることで中間項を評価している．

3.2.2 型安全性の証明

本研究では中間項に型を付けないことで項に対する型付け規則を関数で定義しながら型安全性の証明をすることとした．つまり，型安全性の定理（1.2.1）の e が動く範囲を中間項の含まない項とすれば十分である．

実際，中間項は評価の途中に現れるだけで与えられる抽象構文木に出現することは想定していない．以下に Arith に対する型安全性の証明を，中間項に対して型付けを行うことなく項の構造に関する帰納法で示す．

$e = num\ n$ の場合は省略．

$e = \mathtt{add}\ e1\ e2$ の場合，与えられる帰納法の仮定は以下．

$$\Gamma \vdash e1 : T \land e1 \Downarrow v1 \Gamma \vdash v1 : T \tag{3.14}$$

$$\Gamma \vdash e2 : T \land e2 \Downarrow v2 \Gamma \vdash v2 : T \tag{3.15}$$

また受け取れる前提は以下．

$$\Gamma \vdash \mathtt{add}\ e1\ e2 : T \tag{3.16}$$

$$\mathtt{add}\ e1\ e2 \Downarrow v \tag{3.17}$$

また，以下の値の型付けに対する逆も仮定する（証明は省略）．

$$\Gamma \vdash v : \mathtt{Int} \Rightarrow \exists n. v = \mathtt{vnum}\ n \tag{3.18}$$

示すべき命題は以下．

$$\Gamma \vdash v : T$$

図 3.2 が示すべき証明木である．

以上により中間項に対しては型を付けなくとも型安全性の証明ができることが示せた．

同様の議論を単純型付ラムダ計算でも行いたいが仮定が足りないことは前述した．そこで値に対する型付けの条件としてラムダ抽象の帰納法の仮定を埋め込むことにした．その定義は図 3.3 に

$$\dfrac{\dfrac{\dfrac{}{\texttt{add}\ e1\ e2 \Downarrow v}\ 3.17}{e1 \Downarrow v1} \quad \dfrac{\dfrac{\dfrac{}{\texttt{add}\ e1\ e2 \Downarrow v}\ 3.16}{e1 \Downarrow v1} \quad \dfrac{\dfrac{}{\Gamma \vdash \texttt{add}\ e1\ e2 : \texttt{Int}}\ 3.17}{\Gamma \vdash e1 : \texttt{Int}}}{\Gamma \vdash v1 : \texttt{Int}}\ 3.14}{e1 \Downarrow \texttt{vnum}\ n1 \cdots (a)}\ 3.18$$

$$\dfrac{e1 \text{ と同様}}{e2 \Downarrow \texttt{vnum} n2 \cdots (b)}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{vnum}(n1+n2) : \texttt{Int}} \quad \dfrac{\dfrac{\dfrac{\vdots\ (a)}{e1 \Downarrow \texttt{vnum}\ n1} \quad \dfrac{\dfrac{\vdots\ (b)}{e2 \Downarrow \texttt{vnum}\ n2} \quad \dfrac{}{\texttt{add2}\ (\texttt{vnum}\ n1)\ (\texttt{vnum}\ n2) \Downarrow \texttt{vnum}(n1+n2)}}{\texttt{add1}\ (\texttt{vnum}\ n1)\ e2 \Downarrow \texttt{vnum}(n1+n2)}\ \text{評価規則の展開}}{\texttt{add}\ e1\ e2 \Downarrow \texttt{vnum}(n1+n2)}\ 3.17}{\Gamma \vdash v : \texttt{Int}}$$

図 3.2: Arith の型安全性の証明木

示した.

$$\begin{aligned}
&(e_{env[x \mapsto v']} \Downarrow_{l'} v \wedge \Gamma \models env \wedge \Gamma \vdash v' : dt \Rightarrow \\
&\Gamma \vdash v : t2 \wedge \Gamma \models l') \Rightarrow \\
&\Gamma \vdash \lambda x : dt.e\ env\ \ : dt \to t2
\end{aligned}$$

図 3.3: クロージャに対する型付けの条件

図 3.3 及びこの逆（以下）を示せば (3.10)(3.12) から証明を進めることができ，単純型付ラムダ計算に対しても型安全性の証明が定義できる.

$$\begin{aligned}
&\Gamma \vdash \lambda x : dt.e\ env\ \ : dt \to t2 \Rightarrow \\
&(e_{env[x \mapsto v']} \Downarrow_{l'} v \wedge \Gamma \models env \wedge \Gamma \vdash v' : dt \Rightarrow \\
&\Gamma \vdash v : t2 \wedge \Gamma \models l')
\end{aligned}$$

3.2.3 読み書き可能なラベル

単純型付ラムダ計算に対する評価規則の型シグネチャを以下の 1 行目ようにすることで，入出力が可能なラベルを実装することが可能になる．2, 3 行目は 1 行目を展開したものである．引数と返り値の両方にラベルを表すデータ型 L が出現しており，ラベルの入力，出力の両方ができるようになっていることが確認できる.

```
1 MAlgebra (Lambda :+: iLambda) (nat -> L -> (Val', L))
```

```
2 (r -> nat -> L -> (Val', L)) -> (Lambda :+: iLambda) r -> nat -> L ->
3     (Val', L)
```

具体的な単純型付ラムダ計算の評価関数は図 3.4 のように表現できる.

```
1  L, Env id L Val' => evalLambdaPB ::
2  MAlgebra (Lambda :+: Lambda_i Val') (nat -> L -> (Val' * L))
3  evalLambdaPB _ _ 0 l =  (TIMEOUT, l)
4  evalLambdaPB rec (inl (var x)) n l = case lookup l x of
5    Some v => (v, l)
6  evalLambdaPB rec (inl (lam x t)) n l = (Completed (inj (vlam env x t)), l)
7  evalLambdaPB rec (inl (app t1 t2)) n l = let (v1,l1) := rec t1 (n-1) l in
8    evalLambdaPB rec (inr (app1 v1 t2)) (n-1) l1
9  evalLambdaPB rec (inr (app1 v1 t2)) n l = let (v2 ,l2) := rec t2 (n-1) l in
10   evalLambdaPB rec (inr (app2 v1 v2)) (n-1) l2
11 evalLambdaPB rec (inr (app2 v1 v2)) n l = case v1 of
12   Completed v1' ->
13     case prj v1' of
14     Some (vlam l' x t) -> let (v, l'') := rec t (n-1) (update l' x v2) in
15         (v, remove l'' x)
16   TIMEOUT -> (TIMEOUT, l)
```

図 3.4: Pretty-big-step での Lambda の評価関数

Pretty-big-step になって評価順が明確になったことでラムダ適用の左側を先に評価し（7 行目），その評価で返ってきたラベルを使って右側の項を評価する（9 行目），という評価の流れが実現できている.

第4章　事例研究

本章では Arith，Lambda の定義と型安全性の証明を行い，本研究の提案するフレームワークの使い方を述べる．

4.1　Arith の定義と型安全性の証明

4.1.1　構文定義

Arith の項，中間項，値，型の構文定義に必要な構文の定義は以下のようになる．

```
1 data Arith a = num nat | add a a
2 data iArith v a = add1 v a | add2 v v
3 data vArith a = vnum nat
4 data TArith a = Nat
```

1 行目 Arith が項の構文，2 行目の iArith が中間項の構文，3 行目 vArith が値の構文，4 行目の TArith は型の構文定義である．再帰部は型変数 a としておき，中間項の定義に現れる型変数 v には値の型が入る．

4.1.2　型付け規則

型付け algebra は以下のように定義できる．前章で述べたように型付けは項にのみ与える．

```
1 TArith :<: DType =>
2   typeofArith :: MAlgebra Arith (Gamma -> option DType)
3 typeofArith rec (num nat) gamma = Some (inj Nat)
4 typeofArith rec (add e1 e2) gamma = case (rec e1 gamma, rec e2 gamma) of
5     (Some t1, Some t2) -> case (prj t1, prj t2) of
6         (Some Nat, Some Nat) -> Some (inj Nat)
7         _ -> None
8     _ -> None
```

1 行目に仮定した TArith を包含する DType は拡張をした全体の言語の型の構文を表すデータ型になる．Arith と Lambda を拡張した言語であれば，TArith:+:TLambda(TLambda は TArith 同様 Lambda に対する型を表すデータ型) が入る．

3 行目が T-Num に対応する分岐，4 行目から 8 行目までが T-Add に対応する分岐である．4 行目で部分項型付け関数を使い部分項を型付けする．4 行目の case of に対応するのが 5 行目と 8 行

目である．T-Add に従い部分項両方が型付けできていて，かつ両方が Nat に型付けされている場合に Nat に型付けする．それ以外の場合では型付けをしない．

4.1.3　評価規則

Pretty-big-step でラベルを用いた評価 algebra は以下の通り．

```
1  vArith :<: V, stuck :<: V, (Arith :+: iArith (Fix V)) :<: F', L =>
2    evalArith :: MAlgebra (Arith :+: iArith (Fix V)) (nat -> L -> (Val', L))
3  evalArith _ _ 0 l = (TIMEOUT, l)
4  evalArith rec (inl (num n0)) n l = (Completed (inj (vnum n0)), l)
5  evalArith rec (inl (add e1 e2)) (n+1) l = let (v1, l1) := (rec e1 n l) in
6      evalArith rec (inr (add1 v1 e2)) n l1
7  evalArith rec (inr (add1 v1 e2)) (n+1) l = let (v2, l2) := (rec e2 n l) in
8      evalArith rec (inr (add2 v1 v2)) n l2
9  evalArith rec (inr (add2 v1 v2)) n l = case (v1, v2) of
10     (Completed v1', Completed v2') -> case (prj v1', prj v2') of
11         (Some (vnum n1), Some (vnum n2)) ->
12             (Completed (inj (vnum (n1+n2))), l)
13         _ -> (Completed stuck, l)
14     _ -> (TIMEOUT, l)
```

V が拡張後の値を表すデータ型，F' が拡張後の項と中間項を表すデータ型，L がラベルである．stuck は行き詰まり状態を表すデータ型である．「$(\lambda x : T.x) + 3$」のように値でないのに評価すべき規則がないような項は stuck に評価される．stuck は決して型付けされない．

3 行目が制限された評価木の高さで評価が終わらない場合である．4 行目が PBL-Num に対応する場合である．5，6 行目が PBL-Add に対応する場合である．加算の左側を部分項評価関数によって評価をし，再帰呼び出しで中間項の評価をしている．7，8 行目が PBL-Add1 に対応する場合である．9 行目から 14 行目が PBL-Add2 に対応する場合である．add2 の持つ値の両方が評価完了してる場合で (10 行目から 13 行目)，共に vnum に評価されている場合 (11，12 行目)，それらを足し合わせて評価が完了したことを示す Completed でラップして返す．add2 の値が両方評価完了してかつ vnum に評価されていないものがある場合 (13 行目)，評価すべき規則が存在しないので stuck とする．評価完了していないものがある場合 (14 行目)，add2 自体の評価が終わらなかったことにする．

証明と再利用のため，この評価規則を中間項を含まない形でも定義する．つまり evalArith 関数は MAlgebra (Arith :+: iArith) Val という型シグネチャをしているが MAlgebra Arith Val のような型シグネチャを持つ関数も定義する必要がある．その定義は以下のようになる．

```
1  vArith :<: V, stuck :<: V Arith :<: F, L =>
2      evalArith' :: MAlgebra Arith (nat -> L -> (Val', L))
3  evalArith' rec e = evalArith rec (inj e)
```

4.1.4 値と型に対する Well-Formed 性の定義

値と型が well-formed であるという関係を定義する．well-formed 性とは値に型が付くことの必要十分条件である．Arith の値と型に対する well-formed 性は以下のようになる．

定理 4.1.1 (Arith の値と型に対する well-formed 性)

$$\Gamma \vdash v : Nat \Leftrightarrow \exists n. v = \texttt{vnum}\ n$$

v が Nat に型付けされているとき，かつその時に限りある n が存在し，v が vnum n と表せる，というものである．この規則を Coq で定義すると以下のようになる．

```
1 Inductive WFValue_Arith_v (WFV : WFValue_i -> Prop) : WFValue_i -> Prop :=
2 | WFV_v_num : forall n,  WFValue_Arith_v WFV (mk_WFValue_i (vnum n) tnat).
```

この定義では well-formed 性の定理の ($\Leftarrow$) を公理として定義していることになる．($\Rightarrow$) に関しての証明は自明.

4.1.5 ラベルと型環境に対する Well-Formed 性の定義

ラベルと型環境の well-formed 性を定義する．ラベルと型環境の well-formed 性とは，型環境の下でラベルが健全であることの必要十分条件である．しかし Arith の評価には固有のラベルを使用しないので，任意のラベルについて成り立つ関係となる．つまり以下のように書ける．

定理 4.1.2

$$\Gamma \models \sigma \Leftrightarrow True$$

これを Coq で定義すると以下のようになる．

```
1 Inductive WFLabel_Arith (WFL : WFLabel_i -> Prop) :  WFLabel_i -> Prop :=
2 | WFL_ari : forall l g, WFLabel_Arith WFL (mk_WFLabel_i l g).
```

この定義でも値と型の well-formed 性と同様に ($\Leftarrow$) のみを公理として定義しているが，($\Rightarrow$) は自明なので以上により型安全性のために必要な定義が完了する．

4.1.6 型安全性の証明

証明すべき型安全性の命題は以下のようになる．

型安全性の証明は図 2.6 に示したように proof algebra を使って `PAlgebra Arith` を与えることで，Arith を拡張した言語に対する証明を構築する．図 2.6 の Soundness は実際には以下のように与えられる．

定理 4.1.3 (PAlgebra で示すべき型安全性の命題)

$$t : (\texttt{Fix}\ F) \to (\Gamma \vdash t : T\ \wedge\ t_\sigma \Downarrow_{\sigma'} v\ \wedge \Gamma \models \sigma\ \Rightarrow\ \Gamma \vdash v : T\ \wedge\ \Gamma \models \sigma')$$

これを項の構造に関する帰納法で与えばよい．証明の概略は前章で述べたので省略する．

4.2　Lambda の定義と証明

4.2.1　構文規則

Arith 同様に項，中間項，値、型に対して構文規則を定義する．

```
1 data Lambda a = var id | lam id DType a | app a a
2 data Lambda_i v a = app1 v a | app2 v v
3 data Env L a, Lambda :<: F => vLambda a = clo id DType (Fix F)  (L a)
4 data TLambda a = TArrow a a
```

2 行目の中間項，4 行目の型については Arith のときとほとんど同じである．1 行目の項について，id は変数を表すデータ型である．変数については連想配列の参照と束縛に使うだけなので，決定性（$\forall x\ y.x = y \vee x \neq y$）さえ言えていればよい．DType は Lambda を拡張した言語の型を表すデータ型である．3 行目の値について，クロージャは Arith の vnum に比べると多少複雑である．id 及び DType は項と一緒であるが，第三引数は拡張した言語の項を表すデータ型を閉じたものになる．クロージャは値であるがの中身は値ではなく項として扱うべきなので再帰部を埋める穴でなく，拡張した言語の項の型を直接参照する．第四引数の (L a) は環境である．より正確に言うならば環境は型クラスとして定義するので，環境としての読み書きが可能なデータ型である．L に渡す型引数は環境が束縛する型である．環境は値を束縛するべきなので L に渡す型変数は再帰部を埋めるための a を渡す．

4.2.2　型付け規則

ラムダ計算言語に対する型付け関数は以下のようになる．

```
1  TLambda :<: DType =>
2    typeofLambda :: MAlgebra Lambda (Gamma -> option DType)
3  typeofLambda rec (var x) g = lookup g x
4  typeofLambda rec (lam x dt e) = case rec e (update g x dt) of
5      Some t' -> Some (inj (TArrow dt t'))
6      _ -> None
7  typeofLambda rec (app e1 e2) = case (rec e1 g, rec e2 g) of
8      (Some t1, Some t2) -> case prj t1 of
9          Some (TArrow t1a t1b) -> if (t2 == t1a) then (Some t1b) else None
10         _ -> None
11     _ -> None
```

前提として拡張した言語の型を表すデータ型を DType と仮定する．2 行目の型シグネチャはこの型付け関数が型の上で型付け Algebra となることを示している．

3 行目が T-Var に対応する場合である．4 から 6 行目が T-Lam に対応する場合である．ラムダ抽象に型が付くためには中の項を，x に T を束縛した型環境で型が付く必要がある．7 から 11 行目が T-App に対応する場合である．オペレータに $T_1 \to T$ の型が付き，オペランドに T_1 の型が付く場合，ラムダ適用には T の型が付く．

4.2.3 評価規則

Pretty-big-step でラベルを用いた評価関数は以下の通り．

```
1  vLambda :<: V, stuck :<: V, (Lambda :+: Lambda_i (Fix V)) :<: F', Env L (Fix
       V)
2    => evalLambda ::
3      MAlgebra (Lambda :+: Lambda_i (Fix V)) (nat -> L -> (Val', L))
4  evalLambda _ _ 0 l = (TIMEOUT, l)
5  evalLambda rec (inl (var x)) n l = case lookup l x of
6      Some v => (Completed v, l)
7      None => (Completed stuck, l)
8  evalLambda rec (inl (lam x dt e)) n l = (Completed (inj (clo x dt e   l)), l)
9  evalLambda rec (inl (app e1 e2)) n l = let (v1, l1) := rec e1 (n-1) l in
10     evalLambda rec (inr (app1 v1 e2)) (n-1) l1
11 evalLambda rec (inr (app1 v1 e2)) n l = let (v2, l2) := rec e2 (n-1) l in
12     evalLambda rec (inr (app2 v1 v2)) (n-1) l2
13 evalLambda rec (inr (app2 v1 v2)) n l = case (v1, v2) of
14     (Completed v1', Completed v2') -> case prj v1' of
15         Some (clo x dt l' e) -> let (v, l'') :=
16             rec e (n-1) (injWith (update l' x v2') l) in
17             (v, injWith l l'')
18         _ -> (Completed stuck, l)
19     _ -> (TIMEOUT, l)
```

V が拡張後の値を表すデータ型，F' が拡張後の項と中間項を表すデータ型，L がラベルであり，stuck が行き詰まり状態を表すデータ型なのは Arith と同様である．

4 行目は制限された高さで評価が完了しない場合である．5 から 7 行目が PBL-Var に対応する場合である．ラベルの環境部分の x に束縛されている値を参照する．値が束縛されていれば評価が完了したことを表す Completed でラップして返す．束縛されていなければ行き詰まり状態なので stuck を Completed でラップして返す（評価が完了しなかったのではなく，評価が完了して行き詰まり状態になったということに注意）．8 行目が PBL-Lam に対応する場合である．ラムダ抽象の場合はラムダ抽象にラベルの環境を渡しクロージャとする．9，10 行目が PBL-App に対応する場合である．ラムダ適用のオペレータを部分項評価関数で評価し，再帰呼び出しで中間項の評価をしている．11, 12 行目が PBL-App1 に対応する場合である．13 から 19 行目が PBL-App2 に対応する場合である．app2 の持つ 2 つの項が評価完了して (14 から 18 行目) かつ，オペレータが

クロージャに評価される場合 (15 から 17 行目)，クロージャの中身を評価する．injWith 関数は 2 つのラベルを取り，最初のラベルの環境部分と 2 番目のラベルの環境以外の部分を組み合わせて全体のラベルを構成する関数である．従ってクロージャの中身を評価する際は，クロージャが持つラベルの x に v2' を束縛したものの環境部分と app2 を評価するときに受け取っているラベルの環境以外の部分を組み合わせたラベルで評価をし，返ってきたラベルの環境以外の部分と，app2 を評価する時に受け取っているラベルの環境部分を組み合わせて返している．オペレータがクロージャに評価されなかった場合 (18 行目) は行き詰まり状態になる．オペレータかオペランドのどちらかが評価完了していなかった場合 (19 行目) は app2 自体の評価が終わらなかったこととする．

Arith と同様に，中間項を取らない形の評価規則も定義する．

```
1 vLambda :<: V, stuck :<: V, Lambda :<: F, Env L (Fix V) =>
2     evalLambda' :: MAlgebra Lambda (nat -> L -> (Val', L))
3 evalLambda' rec e = evalLambda rec (inj e)
```

4.2.4　値と型に対する well-formed 性

値と型の well-formed 性を定義する．前章で述べた通り型安全性を項の構造に関する帰納法で証明するためにはクロージャの型付け規則に帰納法の前提を埋め込む必要がある．帰納法の前提を埋め込んだ Well-Formed なクロージャとは以下のようなものである．

定理 4.2.1 (Lambda に対する値と型の well-formed 性)

$$
\begin{aligned}
&\Gamma \vdash \texttt{vlam}\ x\ DT\ env\ t : DT \to T_2 \quad \Leftrightarrow \\
&(t_{env[x \mapsto v']} \Downarrow_{\sigma'} v \wedge \Gamma \models env \wedge \Gamma \vdash v' : dt \Rightarrow \\
&\Gamma \vdash v : T_2 \wedge \Gamma \models \sigma')
\end{aligned}
$$

これを Coq で実装すると以下のようになる．

```
1  Inductive WFValue_Clos(WFV : WFValue_i -> Prop) : WFValue_i -> Prop :=
2  | WFV_Clos : forall x tx ev e (t1 t2: DType) v T,
3           v = clo x tx ev e ->
4           T = tarrow t1 t2 ->
5           eqv tx t1 = true ->
6           (forall (v v': Value),
7      (exists n, Completed _ v = fst (eval_rec e n (update _ ev x v'))) ->
8      WFV (mk_WFValue_i v' tx) ->
9      WFV (mk_WFValue_i v t2)) ->
10          WFValue_Clos WFV (mk_WFValue_i v T).
```

Arith と同様にこの定義は well-formed 性の定理の ($\Leftarrow$) を公理として定義していることになる．従って ($\Rightarrow$) を証明する必要があるが，この証明は不可能である．なぜならば MTC のフレームワーク

として再帰部の出現は strictly-positive であることが要求されるが，この定義では strictly-positive でない出現がある．このことについては 6 章で改めて議論する．

4.2.5 ラベルと型環境に対する well-formed 性

ラベルと型環境の関係を定義する．型環境の下で環境が健全であるとは，環境と型環境の定義域が一致していて，それぞれについて Well-Formed な関係があることが言えればよい．つまり以下のようにラベルの Well-Formed 性を定義する．

定理 4.2.2 (Lambda に対するラベルと型環境に対する well-formed 性)

$$
\begin{aligned}
&\Gamma \models L \Leftrightarrow \\
&((\forall x\ v. L(x) = Some\ v \Rightarrow (\exists t. \Gamma(x) = Some\ t \wedge \Gamma \vdash v : t)) \wedge \\
&(\forall x. L(x) = None \Rightarrow \Gamma(x) = None))
\end{aligned}
$$

これは以下のように書ける．

```
1 Inductive WFLabel_Lambda (WFL : WFLabel_i -> Prop) :  WFLabel_i -> Prop :=
2 | WFL_lam : forall (l: L)(g: Gamma),
3             (forall x v, lookup l x = Some v ->
4                          exists t, lookup g x = Some t /\
5                                    WFValueC v t) ->
6             (forall x, lookup l x = None ->
7                        lookup g x = None) ->
8                 WFLabel_Lambda WFL (mk_WFLabel_i l g).
```

これもこれまでと同様に，$(\Rightarrow)$ の証明が必要であるがこの証明は簡単である．

4.2.6 型安全性の証明

Arith と同様の型安全性の命題を証明する．証明の概略は前章で述べたが Coq ではうまく定義できない．なぜならばラベルの形式化が不十分であるためである．ラベルに対する条件として計算領域が分割可能であり，それぞれについて相互作用がないもの，ということが必要であり，実際型安全性の証明にはこれらの性質が不可欠である．しかし，この性質を実際にどのように定義するのが再利用性の観点から優れているのかわかっておらず，証明が未完のままとなっている．この条件さえ定めることができれば型安全性の証明は完了する．この未完の部分については改めて 6 章で議論する．

4.3 Arith と Lambda の合成

本節では本章で定義してきた Arith 言語と Lambda 言語の定義を再利用し，Arith と Lambda の両方の言語機能を持つ言語の定義と型安全性の証明を構成する．

4.3.1　構文規則

型の構文規則は以下のように定義する．

```
Definition D := TArith :+: TLambda
Definitoin DType := Fix D
```

この定義により以下の条件は自明．

```
TArith :<: D
TLambda :<: D
WF_Functor TArith D
WF_Functor TLambda D
```

同様にして項，値についても同様に構文規則を定義する．中間項については定義しない．

```
Definition F := Arith :+: Lambda
Definition Exp := Fix F
Definition V := vArith :+: vLambda
Definition Val := Fix V
```

4.3.2　型付け規則

型付け規則は Algebra の合成を利用し，Arith と Lambda の型付け規則を合成することにより定義できる．

```
Definition typeof_alg_F := MAlgebra F (Gamma -> option DType) :=
  alg_plus typeofArith typeofLambda
```

但し alg_plus は MAlgebra F A と MAlgera G A を合成して MAlgebra (F :+: G) A を生成するものであった．

4.3.3　評価規則

評価規則についても型付け規則同様 Algebra の合成を利用して定義する．

```
Definition eval_alg_F := MAlgebra F (nat -> L -> (Val', L)) :=
  alg_plus evalArith' evalLambda'
```

ここで各評価規則を定義する際，型シグネチャに中間項が現れないものも定義していたため，中間項について意識することなく評価規則を合成することが可能になっている．

4.3.4　値と型に対する well-formed 性

Arith 言語と Lambda 言語の well-formed 性を合成する．合成の際は値の型付けが一貫性を持つことを証明する必要がある．つまり自然数がラムダ計算のアロー型に決して型付けされず，同様にクロージャが Nat 型に型付けされないことを証明する必要がある．証明は定義に従って展開して矛盾を導くだけなので省略する．

4.3.5 ラベルと型環境に対する well-formed 性

Arith 言語の評価にはラベルを使わないので well-formed 性を合成する必要はない．Lambda 言語のものをそのまま使う．

4.3.6 型安全性の証明

Arith 言語の型安全性の証明と Lambda 言語の型安全性の証明を合成する．合成するために必要な条件は既に揃っているので新しく命題を証明する必要はなく，既存の補題を適用するだけで証明は完了する．具体的な Coq コードは以下．

```
1  Lemma soundness_fpalg:
2        FPAlgebra Exp
3        (eval_alg_Soundness_P D F (L Value) V Gamma WFV WFL
4                      (f_algebra (FAlgebra := typeof_alg_F _))
5                      (f_algebra (FAlgebra := eval_F)))
6        F _ _
7        in_t.
8  Proof.
9    rewrite <- inject_in_t;
10   apply FPAlgebra_Plus_cont_inject; eauto.
11   eapply (Arith_soundness_fpalg D F V Gamma (Typeof_F := Typeof_F) Label (
         eval_F := eval_F) WFV _ WFL (Sub_Arith_F := Sub_Arith_F)).
12   eapply (Lambda_soundness_fpalg D var F L V Gamma (Typeof_F := Typeof_F) (
         eval_F := eval_F) WFV _ WFL _ (Sub_Lambda_F := Sub_Lambda_F)).
13   Unshelve.
14   auto.
15   auto.
16   auto.
17   auto.
18   auto.
19 Defined.
20
21 Theorem eval_soundness:
22   forall (e: Exp)(T: DType) G (n m: nat)(l l': Label)(v: Value),
23       (typeof D Gamma e G (TypeOf := Typeof_F)) = Some T ->
24       (Completed _ v, l') = beval F Label V n e m l ->
25       (WFLabelC D Label Gamma WFL l G) ->
26       (WFValueC D V WFV v T /\ WFLabelC D Label Gamma WFL l' G).
27 Proof.
28   apply eval_soundness.
29   apply soundness_fpalg.
30 Qed.
```

1 行目から 19 行目の補題 soundness_fpalg の証明は Arith 言語と Lambda 言語を合成した言語の型安全性を表現する FPAlgebra の証明である．証明は定義の展開 (9，10 行目)，Arith 言語に対する証明の再利用 (11 行目)，Lambda 言語に対する証明の再利用 (12 行目)，及び Coq が自

動で処理しきれなかった補題について自動化タクティックを使った解決 (13 行目から 18 行目) により構成されている．

この補題を使って実際の型安全性を証明したのが 21 行目以下である．21 行目から 26 行目で宣言される命題は型安全性の定理 (定理 3.1.1) に等しい．この定理を証明するために必要なのは 28 行目から 29 行目の 2 つの命令である．28 行目はこの型安全性の定理を証明するために先ほど 1 行目から 19 行目で証明したような `FPAlgebra` を利用すればよいという補題 (フレームワークが提供するもの) を適用する命令で 29 行目が 19 行目までに証明した `FPAlgebra` を適用する命令である．

以上により構成要素について証明がされていれば，それらを再利用して合成した言語の証明を構成することができることが示された．

第5章　関連研究

N. Amin らの Definitional interpreter[26][3] は大ステップ意味論の問題点として知られていた不停止なプログラムと型の付かないプログラムの区別ができないということを，評価木の高さを制限し，型安全性の証明では任意の高さについて言及することで解決する手法を提案している．またラムダ計算の評価戦略として従来の書き換えでなく，環境から評価時に参照する手法の利点を述べている．しかしモジュラーに拡張可能にしようとはしていない．本研究では評価木の高さを制限して不停止なプログラムと型の付かないプログラムの区別をつける手法は参考にしており，またラムダ計算を書き換えでなく環境から参照する手法を取っているが，構文や証明をモジュラーに拡張可能にしようとしている．

大ステップ意味論で型安全性の証明を記述しようとする研究として，N. Danielsson の Operational Semantics Using the Partiality Monad[13] がある．これは partiality monad[10] を応用して大ステップ意味論を余帰納的に与えて Agda を使って型安全性の証明を行っている．Coq では余帰納的な定義に制限がかかってしまい定義ができないことがわかっているため，この手法は Coq を使って実装をするのに使うことが難しそうである．D. Ancona らの Reasoning on Divergent Computations with Coaxioms[4] も大ステップ意味論で型安全性を証明する研究である．この研究では余原理と呼ばれる手法で不停止なプログラムと型を識別している．

Pretty-big-step 意味論の関連する研究として，G. Cabon らの Annotated Multisemantics To Prove Non-Interference Analyses[9] がある．これは pretty-big-step 意味論で書かれた構文を一度扱いやすい形式に変換して hyperproperty の証明を Coq を使って与えている．Hyperproperty はセキュリティポリシーと表現でき，セキュリティの情報や権限が正しく機能しているか表す定理である．M. Bodin の A Trusted Mechanised JavaScript Specification[8] は pretty-big-step 意味論で JavaScript の ECMA 標準である JSCert を提案している．証明には Coq を使用している．

第6章　結論

6.1　まとめ

本研究では操作的意味論で拡張可能な型安全性の証明の手法が示された．それにより連続性の証明を回避することができた．その結果 MTC では Arith の証明スクリプト定義から含めて約 720 行から本研究の手法で約 600 行に，およそ 120 行削減することに成功した．

また意味論を拡張する手法も提案されているのでラムダ計算に参照等を拡張するのにラムダ計算の定義が再利用が可能である．

6.2　新たに明らかになった課題

本研究によって明らかになった問題点について述べる．

前章までに単純型付ラムダ計算の定義と型安全性の証明をモジュラーに与えるための手法を示したが，実際のところ Coq による証明は不完全である．その理由としてわかっていることを 2 点述べる．

6.2.1　Strictly positive でない再帰部の出現

値と型の関係が strictly-positive でない出現があるということが挙げられる．

定理 4.2.1 はクロージャの型付けの条件である．これを日本語で書き下すと，「『e が，環境 env の x に v' を新たに束縛した環境で評価すると v になり評価後ラベルは l' であり，Γ の下で環境 env が健全かつ v' に dt の型が付く』ならば Γ の下で v に $t2$ の型が付きかつ l' が健全である」とき，またその時に限り Γ の下で $\lambda x : dt.e\ env$ に $dt \to t2$ 型が付く，というものである．

今問題にしている strictly positive でない出現とは，定義しようとしている型環境と値と型の関係が strictly positive でない位置（二重かぎ括弧の中）に出現しているということである．

この型付けの条件も Functor と同様に穴を開けて後で埋める方法で定義する．従って穴は定義しようとしている関係の再帰部なので型環境と値と型の関係の部分が穴になる．しかし Coq が無限ループを防ぎ整合性を保つため，穴の出現は strictly-positive な出現のみ認められている．実際これが原因で証明の仮定で解決不可能なゴールが Coq により生成されてしまうため，一部で証明を諦めることをしてしまっている．

先行研究 [16, 19] では環境に特化した議論をすることでこの strictly-positive 問題を回避しており，これがラベルという一般のデータ構造に対して適用できるか明らかではない．試行錯誤の結

果そのまま適用はできないため何らかの工夫を施すか，全く別のアプローチを考える必要がありそうであることがわかっており，新たに明らかになった問題である．

6.2.2　ラベルの形式化

ラベルは前述の通り，計算領域が分割可能であり，それぞれについて相互作用がないもの，という条件が必要である．実際にこの条件があれば証明が可能であるが，これを Coq のスクリプトとして具体的にどう形式化すればよいか．拡張する際の都合も考え，どのように実装させるのが便利そうであるかがきちんと議論できていない．これをきちんと形式化し，フレームワークに組み込むこともまた今後の課題の一つと言える．

参考文献

[1] Abbott, M., Altenkirch, T. and Ghani, N.: Containers: Constructing strictly positive types, *Theoretical Computer Science*, Vol. 342, No. 1, pp. 3 – 27 (2005). Applied Semantics: Selected Topics.

[2] Allemand, M., Coupet-Grimal, S., Jakubiec, L. and Paillet, J.-L.: A System for Modelling and Proving Circuits, *Proceedings of the 1996 European Conference on Design and Test*, EDTC '96, Washington, DC, USA, IEEE Computer Society, pp. 605– (1996).

[3] Amin, N. and Rompf, T.: Type Soundness Proofs with Definitional Interpreters, *SIGPLAN Not.*, Vol. 52, No. 1, pp. 666–679 (2017).

[4] Ancona, D., Dagnino, F. and Zucca, E.: Reasoning on Divergent Computations with Coaxioms, *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA, pp. 81:1–81:26 (2017).

[5] Appel, K. and Haken, W.: Every planar map is four colorable. Part I: Discharging, *Illinois J. Math.*, Vol. 21, No. 3, pp. 429–490 (1977).

[6] Bach Poulsen, C. and Mosses, P. D.: Deriving Pretty-Big-Step Semantics from Small-Step Semantics, *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, New York, NY, USA, Springer-Verlag New York, Inc., pp. 270–289 (2014).

[7] Benjamin C. Pierce, Chris Casinghino, M. G. V. S. and Yorgey, B.: Software Foundations(http://softwarefoundations.cis.upenn.edu).

[8] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A. and Smith, G.: A Trusted Mechanised JavaScript Specification, *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, New York, NY, USA, ACM, pp. 87–100 (2014).

[9] Cabon, G. and Schmitt, A.: Annotated Multisemantics To Prove Non-Interference Analyses, *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, New York, NY, USA, ACM, pp. 49–62 (2017).

[10] Capretta, V.: General Recursion via Coinductive Types, *CoRR*, Vol. abs/cs/0505037 (2005).

[11] Castéran, P. and Sozeau, M.: A Gentle Introduction to Type Classes and Relations in Coq (2014).

[12] Charguéraud, A.: Pretty-Big-Step Semantics, *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, Berlin, Heidelberg, Springer-Verlag, pp. 41–60 (2013).

[13] Danielsson, N. A.: Operational Semantics Using the Partiality Monad, *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, New York, NY, USA, ACM, pp. 127–138 (2012).

[14] de Recherche en Informatique et en Automatique, I. N.: The Coq Proof Assistant(https://coq.inria.fr/).

[15] Delaware, B., Cook, W. and Batory, D.: Product Lines of Theorems, *SIGPLAN Not.*, Vol. 46, No. 10, pp. 595–608 (2011).

[16] Delaware, B., d. S. Oliveira, B. C. and Schrijvers, T.: Meta-theory à La Carte, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, ACM, pp. 207–218 (2013).

[17] Delaware, B., Keuchel, S., Schrijvers, T. and Oliveira, B. C.: Modular Monadic Meta-theory, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, New York, NY, USA, ACM, pp. 319–330 (2013).

[18] Igarashi, A., Pierce, B. C. and Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ, *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 3, pp. 396–450 (2001).

[19] Keuchel, S. and Schrijvers, T.: Generic Datatypes à La Carte, *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, New York, NY, USA, ACM, pp. 13–24 (2013).

[20] Mosses, P. D.: Foundations of Modular SOS, *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, MFCS '99, London, UK, UK, Springer-Verlag, pp. 70–80 (1999).

[21] Nakata, K. and Uustalu, T.: Trace-Based Coinductive Operational Semantics for While, *Theorem Proving in Higher Order Logics* (Berghofer, S., Nipkow, T., Urban, C. and Wenzel, M.(eds.)), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 375–390 (2009).

[22] Norell, U. and Coquand, C.: The Agda Wiki(http://wiki.portal.chalmers.se/agda/pmwiki.php).

[23] Oliveira, B. C. d. S. and Cook, W. R.: Extensibility for the Masses, *ECOOP 2012 – Object-Oriented Programming* (Noble, J.(ed.)), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 2–27 (2012).

[24] Paulson, L.: Isabelle(https://isabelle.in.tum.de/).

[25] Pierce, B. C.: *Types and Programming Languages* (2002).

[26] Reynolds, J. C.: Definitional Interpreters for Higher-Order Programming Languages, *Higher Order Symbol. Comput.*, Vol. 11, No. 4, pp. 363–397 (1998).

[27] Schwaab, C. and Siek, J. G.: Modular Type-safety Proofs in Agda, *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, New York, NY, USA, ACM, pp. 3–12 (2013).

[28] Sheard, T.: Generic Unification via Two-level Types and Parameterized Modules, *SIGPLAN Not.*, Vol. 36, No. 10, pp. 86–97 (2001).

[29] Swierstra, W.: Data Types à La Carte, *J. Funct. Program.*, Vol. 18, No. 4, pp. 423–436 (2008).

[30] Torgersen, M.: The Expression Problem Revisited, *ECOOP 2004 – Object-Oriented Programming* (Odersky, M.(ed.)), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 123–146 (2004).

[31] Torrini, P. and Schrijvers, T.: Reasoning about modular datatypes with Mendler induction, *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015.*, pp. 143–157 (2015).

[32] Wadler, P.: The Expression Problem, Mailing list (1998).