モジュールの生成が可能なマルチステージ言語の提案

渡部 恭久　亀山幸義

本研究ではモジュールの生成が可能なマルチステージ言語の提案を行う．マルチステージ・プログラミングはメタプログラミングの実現手法の一つであり, コードの生成を行う段階, 生成したコードを実行する段階といったようにプログラムの実行が複数の段階 (ステージ) に分かれているという特徴がある．マルチステージ・プログラミングをサポートするプログラム言語をマルチステージ言語という．マルチステージ・プログラミングはプログラムの実行効率と保守性・再利用性を両立可能なプログラム手法として有力なものであることが知られているが, 現在のマルチステージ言語は生成の対象をプログラムの項のみに限定しており, 他の言語要素をコード生成の対象として扱うことができない．そこで本研究ではモジュールの生成が可能なマルチステージ言語 $\lambda^{\langle M \rangle}$ を提案し, 提案体系を用いてモジュールを活用したソフトウェアの実行効率が改善可能であることを示す．

1　はじめに

マルチステージ・プログラミングはメタプログラミングの一形態であり, プログラムの実行効率と保守性・再利用性を両立可能なプログラム手法として有用なものである．マルチステージ・プログラミングをサポートする機能をもつ言語 (以下，マルチステージ言語と呼ぶ) の中でも, MetaOCaml などの MetaML 系統のマルチステージ言語は, 静的束縛のもと, 生成されるコードの型安全性を保証する理論に支えられているという特徴がある．本研究は，MetaML 系統のマルチステージ言語を拡張し, モジュールを表すコードを生成することが可能なマルチステージ言語を提案し, モジュールを活用したソフトウェアの実行効率を改善することを目標とする．

現在の MetaOCaml はコード生成の対象を項のみに限定しており, 他の言語要素, 例えば型やモジュールといった要素を生成することができない．モジュールは, ソフトウェアの構造化・抽象化を提供する強力な言語機能として OCaml プログラムでは広く利用されているが, ファンクタ (モジュール上の関数) を利用したプログラムでは, モジュールのメンバ参照に関して実行時オーバヘッドが生じてしまうという問題がある．本研究では, マルチステージ・プログラミングの手法により, モジュール内部メンバへの参照を参照先のプログラムで置き換えたコードを生成することで, このオーバヘッドを削減し, モジュール抽象化と実行効率の両立をはかる．

ML 系統のマルチステージ言語におけるモジュールの扱いに関する従来研究として,Inoue らの研究 [1] がある．彼らは, モジュール生成機能が必要と思われるいくつかのメタプログラムの例について論じた後, それらの例は, 手動によるプログラムの書換えによってモジュール生成が必要のない MetaOCaml プログラムとして実現できることを示した．ただし, この書換えが有効な範囲については論じていない．本研究は MetaOCaml のコード言語にモジュール生成機能を加えた新しいプログラム言語 $\lambda^{\langle M \rangle}$ を提案し, その言語で記述された任意のプログラムが, Inoue らの書換えを一般化した変形により,MetaOCaml プログラムに変換できることを示す．

本稿の構成は以下の通りである．2 章では, 本研究

Multi-Stage Module Generation

Takahisa Watanabe, Yukiyoshi Kameyama, 筑波大学大学院 システム情報工学研究科コンピュータサイエンス専攻, Dept. of Computer Science, University of Tsukuba.

の背景であるマルチステージ・プログラミングについて述べる．3 章では, マルチステージ・プログラミングによるモジュールの生成を概観し, 本研究の立場・目的を明らかにする．4 章では, モジュールの生成が可能なマルチステージ言語の実現方針について述べる．5 章では, 本研究で提案するマルチステージ言語 $\lambda^{\langle M \rangle}$ についての概略を述べる．6 章では, $\lambda^{\langle M \rangle}$ に対する型システムの設計などについて明らかにする．7 章では, $\lambda^{\langle M \rangle}$ から MetaOCaml への変換について述べ, モジュールの生成が可能なマルチステージ言語の実現手法を明らかにする．8 章では本研究の内容とその成果について総括する．また今後の課題についても述べる．

2 マルチステージ・プログラミング

マルチステージ・プログラミングは, プログラムを生成する段階・生成したプログラムを実行する段階といったようにプログラムの実行が複数の段階 (ステージ) を持つという特徴を持つ．保守性・再利用性の高い形でプログラムを生成するプログラムであるプログラム生成器を記述し, 記述したプログラム生成器から対象領域に関する知識や実行環境などのパラメータに特化した実行効率の良いプログラムを生成・利用することで, プログラムの保守性・再利用性と実行効率を両立させることができる．本論文ではプログラム生成器が生成する値としてのプログラム片を, 通常のプログラムと区別してコードと呼ぶことにする．マルチステージ・プログラミングの利点としてプログラム生成器によって生成されるコードが構文エラー・自由変数への参照・型の不整合などを含まないことが型システムによって静的に保証されるという点が挙げられる．型システムによる保証のないコード生成では, 生成されたコードの安全性に関する問題がコードが実行される段階になって表面化するために, エラーの発見が遅れてしまいデバッグが難しくなるという問題がある．生成されるコードの安全性に関する型システムによるサポートが受けられるプログラム言語として MetaOCaml, Scala LMS などがある．

本研究で提案するマルチステージ言語は MetaOCaml をその基盤としているため, 本節では MetaOCaml でのマルチステージ・プログラミングについて説明する．MetaOCaml は OCaml に対するマルチステージ・プログラミングのための拡張言語であり, 擬似引用 (Quasi-quotation) 方式によるマルチステージ言語である．

MetaOCaml は 3 種類のマルチステージ構成子によってマルチステージ・プログラミングを実現している．

- **コードの生成 `.<e>.`**
  式 ブラケット`.<>.`で括ることで, コードが生成される．コード化された式は現在の段階では評価されず, そのままの形で保持される．例えば, 式 `1 + 2`を評価すると `3`という結果が得られるが, 式 `.<1 + 2>.`という式を評価しても, `.<>.`の中身は評価されず, `.<1 + 2>.`という評価結果になる．
- **コードの合成 `.~e`**
  ブラケット中の式 `.~e` は, 式 `e`を評価して得られるコードをブラケットの中に埋め込む．`let x = .<1>. in .<.~x + 2>.`という式の評価結果は `.~x`の箇所に`.<1>.`というコードの中身が展開され, `.<1 + 2>.`という評価結果となる．
- **コードの実行 `!.e`**
  式 `!.e` は式 `e`を評価して得られるコードを実行する．式 `!.(.<1 + 2>.)`の評価結果はコード `.<1 + 2>.`の中身が評価され, `3`という評価結果となる．

3 モジュールの生成

MetaOCaml のモジュールにはモジュールの境界を越えたメンバ参照 (間接参照) に関して実行時オーバーヘッドが存在する．例えば効率の良い SQL 問い合わせを生成するための言語統合クエリ (LINQ) に関する研究 [3] では, 再帰モジュールを用いてモジュールの内部メンバを膨大な回数参照するプログラムが用いられており, 間接参照にかかるコストは非常に小さいながらも, その回数が膨大になるとオーバーヘッドが累積して無視できないものになる．

ソースコード 1 は MetaOCaml のモジュールを用いた集合型を表現するモジュールの素朴な実装例である．MetaOCaml (OCaml) ではモジュールの型を

sig ... **end** で, モジュールの実装を **struct** ... **end** で記述する. モジュール型 `EQ` は等しさが定義できるような型の表現, `MakeSet` は `EQ` 型のモジュール `Eq` を受けとって, `Eq.t` 型の値を要素とするとうな集合型を作るモジュールである. `MakeSet` は引数にモジュールを取るモジュール (モジュール上の関数) になっており, このようなモジュールを特にファンクターと呼ぶ. `IntSet` は `MakeSet` を用いて作成した `int` 型の要素を持つ集合型の実装である.

```
1  module type EQ = sig
2    type t
3    val eq: t -> t -> bool
4  end
5
6  module type SET = sig
7    type elt
8    type set
9    val member: (elt -> set -> bool)
10 end
11
12 module MakeSet (Eq: EQ)
13  : SET with type t = Eq.t =
14  struct
15   type elt = Eq.t
16   type set = Eq.t list
17   let member elt = function
18   | [] -> false
19   | elt' :: set' ->
20     Eq.eq elt elt' || member elt set'
21 end
22
23 module IntSet = MakeSet (struct
24   type t = int
25   let eq = (=)
26 end)
```

ソースコード 1: 間接参照を含むプログラム

このプログラムは 11 行目で `Eq.eq` という間接参照を含んでいる. この間接参照は最大で集合の要素数だけ呼ばれるため, 間接参照のオーバーヘッドが累積する問題のあるプログラム例になっている.

間接参照を除去した結果のプログラムがソースコード 2 である. ソースコード 1 の 11 行目に対応する箇所がソースコード 27 行目であり, 間接参照 `Eq.eq` が展開されて, 参照先のプログラムである `(=)` で置き換えられている. このような展開をマルチステージ・プログラミングによって実現したい.

```
1  module IntSet = struct
2    type elt = int
3    type set = int list
4    let member = elt = function
5    | [] -> false
6    | elt' :: set' ->
7      (=) elt elt' || member elt elt'
8  end
```

ソースコード 2: 効率化されたモジュール

ソースコード 3 は Inoue らの手法を用いた擬似的モジュール生成を用いた間接参照の除去を行うプログラムである. モジュールの境界を越えて参照される `Eq.eq` 関数をコードとして保持しておき, 参照先でそのコードを展開することでソースコード 2 のようなプログラムを生成する. `Eq.eq` という間接参照はコードを生成する時に一度行われるのみとなり, 生成されたコードからは間接参照を完全に除去することができる.

ソースコード 3 はソースコード 1 とのギャップが大きい. 特に `MakeSet` モジュールが受けとる引数の型が `EQ_CODE` という新しいモジュール型に変わってしまっており, `EQ` 型の要素を再利用できなくなってしまっている点が問題である.

```
1  module type EQ_CODE = sig
2    type t
3    val eq: (t -> t -> bool) code
4  end
5
6  module type SET = sig
7    type elt
8    type set
9    val member: (elt -> set -> bool)
10 end
11
12 let makeSet (type a)
13   (module Eq
14     : EQ_CODE with type t = a) =
15   (module struct
16     type elt = Eq.t
17     type set = Eq.t list
18     let member = Runcode.run
19       .<let rec member elt = function
20         | [] -> false
21         | elt' :: set' ->
```

```
22            .~(Eq.eq) elt elt'
23               || member elt set'
24          in member>.
25     end: SET with type elt = a)
26  module IntSet =
27    (val makeSet (module struct
28      type t = int
29      let eq = .<(=)>.
30    end: EQ_CODE with type t = int))
```

ソースコード 3: MetaOCaml を用いた間接参照の除去

本研究ではソースコード 1 とのギャップの少ない, モジュールの生成を自然に記述できる表層構文を提案する. ソースコード 4 は提案するマルチステージ言語を用いて記述したものである. 提案言語を用いることで, 元々の `EQ` モジュール型に **code** という型構成子を付けて再利用できるようになっており, Inoue らの手法をそのまま適用した場合と比べてプログラムの再利用性が向上している. また, ソースコード 1 のプログラムをソースコード 4 のプログラムに変換するためには, マルチステージプログラミングのための構成子を挿入する必要があるが, これは部分計算における束縛時解析の手法で容易に実行できる.

```
1   module type EQ = sig
2     type t
3     val eq: t -> t -> bool
4   end
5
6   module type SET = sig
7     type elt
8     type set
9     val member: (elt -> set -> bool)
10  end
11
12  let makeSet (type a)
13    (module Eq
14       : (EQ with type t = a) code) =
15    .<(module struct
16        type elt = %(⇓Eq).t
17        type set = %(⇓Eq).t list
18        let member elt = function
19        | [] -> false
20        | elt' :: set' ->
21          .~((⇓Eq).eq) elt elt'
22             || member elt set'
23       end : SET with type elt = a) >.
24
25  module IntSet =
26    run_module (val makeSet .<(module
          struct
27        type t = int
28        let eq = (=)
29      end: EQ with type t = int)>.)
```

ソースコード 4: 提案するマルチステージ言語を用いた間接参照の除去

4 本研究のアイデア

Inoue らによる第一級モジュールを用いた擬似的なモジュールコードの生成を観察することにより, 「モジュールのコード」と「コードからなるモジュール」が同型であるため, 前者で記述したプログラムを後者に変換すればよいという観察に至った. これを詳しく説明する.

図 1 は、モジュールのコードの型 (左) とコードをメンバとして持つモジュールの型 (右) を表す. これらの型の自然な解釈のもとでは, 両者は全単射で対応する集合として解釈され, 型として同型であると考えられる。MetaOCaml では直積型に対して $(A * B)$ **code** と $(A$ **code**$) * (B$ **code**$)$ は同型であると考えらえ, 図 1 は直積型をモジュール型に一般化したものである.

(**sig**	**sig**
type $t =$ **int**	**type** $t =$ **int**
val $v_1 : t$	**val** $v_1 : t$ **code**
val $v_2 : t \to t$	**val** $v_2 : (t \to t)$ **code**
end) **code**	**end**

図 1: 左: モジュール・コードの型, 右: コードの要素を含むモジュールの型

現在の MetaOCaml では, 図 1 の左側の型は表現できない一方で, 右側の型は表現可能である. そこで本研究では上記の対応に基づき, 左側の型の要素を右側の型の要素に変換する. これはモジュール・コードの生成が可能なマルチステージ言語をモジュール・コードの生成ができない MetaOCaml へ変換することに相当するというのが本研究の基本的なアイデアである. なお, 上記における「同型性」は, 圏論等におけ

る厳密に定式化された用語ではなく, 非形式的な意味論にもとづくものである.

上記の変換で注意すべき点は, モジュールのメンバとしての型 t の具体的な型である **int** は変換していない (**int code** にしていない) 点である. これは型 t をコード型に変形してしまうと, たとえば図における v_2 の型が記述できなくなるためである.

また本研究では, 型の生成 (型をあらわすコードの生成) は許していないことにも注意されたい. 型 (のコード) の生成を許す言語では, コード生成段階で使える型の集合より, コード実行段階で使える型の集合が多くなる. コード生成前に, 生成されるコードの型安全性を保証する MetaML 系統のマルチステージ言語では, 型の生成を許すことは技術的に相当大きな困難があると予想される. 一方で, 型の生成が絶対不可欠な (型の生成をおこなわない同等のプログラムが存在しない) プログラム効率化の具体例は見つけられていない. そこで本研究では, 型の生成は扱わないこととした. これによりモジュール・コードの生成を簡潔に扱うことができるようになるという利点があった.

5 提案するマルチステージ言語 $\lambda^{\langle M \rangle}$

モジュールの生成が可能なマルチステージ言語には設計上の選択肢が幾つか存在するが, 本研究で提案するマルチステージ言語 $\lambda^{\langle M \rangle}$ では一級モジュールとマルチステージ構成子を組み合わせてモジュール・コードを記述するという設計を採用した. この方式の利点は MetaOCaml にはこれらの機能が既に存在しており, MetaOCaml とのギャップが少ないこと (現在の MetaOCaml ではブラケットの中で一級モジュールを記述するとコンパイル・エラーになる), 一級モジュールを用いることでモジュール・コードやモジュール・コードから生成されたモジュールを自然に扱うことができるということなどが挙げられる.

本研究では 提案するマルチステージ言語に 新しいマルチステージ構成子 $\Downarrow$ を導入している. このマルチステージ構成子はモジュール・コードをコードのモジュールへと変換する働きをする, 本研究の提案する同型性に基づく変換そのものを表している. MetaOCaml ではコードの一部を取り出す操作は許されていないために, モジュール・コードの内部メンバーを参照することができないという問題がある. 間接参照の除去では, モジュール・コードの内部メンバーをコードという形で参照したい. そこで, 本研究で行う変換を $\Downarrow$ という形で明示することで, コードのモジュールを経由したモジュール・コードの内部メンバーの参照を可能にした.

本研究で提案するマルチステージ言語は性質の証明を簡単にするため, 2 段階のマルチステージ言語に制限している. またサブタイピングなどの複雑な要素をなるべく排し, 必要最低限の構成となるように努めた.

本研究で提案するマルチステージ言語はネストしたモジュールをサポートしない. この制限のために, 式をモジュールの内部で利用できる式とモジュールの外部で利用できる式の 2 階層に分割した. また, 変換の都合から `((module M)-> (module N))code` といったような一級モジュール上の関数 (ファンクター) のコードをサポートできないため, この制限のために, 型も Small Type と Large Type の 2 階層に分割している. 直感的には Small Type は 現在の MetaOCaml で記述できる型, Large Type は現在の MetaOCaml では記述できない可能性のある型である.

$\Downarrow$ によるモジュール・コードの内部メンバーを参照を考える上で, モジュール内部メンバーの依存性を考慮しなければならない点に注意されたい. 例えば モジュール・コード `.<(module struct val x : int = 0; val y : int = 1 + x end: sig .. end)>.` における `y` は, それより先に定義されたメンバーである `x` への参照を含んでおり, このようなコードから `y` を素朴に取り出すと `1 + x`という自由変数を含んだコードになるという問題がある. 本研究の現段階ではこの依存性を考慮に入れていないシステムになっているが, この問題は参照する可能性のある定義を全て `let` 束縛の形でコード内に挿入してしまうという手法で解決できると考えている. この手法を用いると先程の問題のあるコードは`.<let x = 0 in 1 + x>.` という形になり, 自由変数への参照を含まない. この手法は生成されるコードを肥大化させてしまうが, 実際のプログラミングにおいては許容できる範囲だろうと予想している. コードの肥大化を抑える工夫については将来

課題である.

6 型システム

本研究では Leroy によるモジュールに対する型システム [2] を基盤とし, 一級モジュールやマルチステージ構成子などに対する型付け規則を追加した型システムの定義を行った. 型システムの完全な定義は付録 B に掲載した.

後述する MetaOCaml の変換について, 提案する型システムの上で以下のような理論的性質が成り立つことが期待されるが, 現在これらの定理について具体的な証明は与えられていない.

定理 6.1. 変換による型付けの保存 (レベル 0)

$$\mathit{If} \quad E \vdash^0 e : \tau, \quad \mathit{then} \quad [\![E]\!]^0 \vdash^0 [\![e]\!]^0 : [\![\tau]\!]^0$$

定理 6.2. 変換による型付けの保存 (レベル 1)

$$\mathit{If} \quad E \vdash^1 e : \tau, \quad \mathit{then} \quad [\![E]\!]^1 \vdash^1 [\![e]\!]^1 : [\![\tau]\!]^1$$

7 MetaOCaml への変換

提案体系 $\lambda^{\langle M \rangle}$ から MetaOCaml への変換を $[\![\circ]\!]^0$, $[\![\circ]\!]^1$ として定義する. それぞれ $\lambda^{\langle M \rangle}$ のレベル 0 の項, レベル 1 の項に対応している. $\circ$ には $\lambda^{\langle M \rangle}$ の構文要素が入る. 変換の完全な定義は付録 C に掲載している.

変換の設計はモジュール・コードとコードの要素を含んだモジュールの間の同型性に着目した. 変換の性質として, それぞれのレベルに対する変換が型付けを保存することを念頭において設計を行っている. 変換後の $\lambda^{\langle M \rangle}$ のプログラムは現在の MetaOCaml でコンパイル可能である.

ソースコード 4 は $\lambda^{\langle M \rangle}$ によるプログラム例である. このプログラムに対して変換を適用すると, Inoue らの手法による MetaOCaml プログラム (ソースコード 3) が得られる. ソースコード 3 を実行することで, 実行効率の改善されたプログラムであるソースコード 2 が得られる.

本研究で提案する変換をソースコード変換器として実装することで, モジュールの生成が可能なマルチステージ言語を容易に実現できるというのが本研究の利点である.

8 まとめと今後の課題

本研究ではマルチステージ・プログラミングによるモジュールの生成について検討し, モジュールの生成が可能なマルチステージ言語 $\lambda^{\langle M \rangle}$ 及びそのシステムの設計を行った. $\lambda^{\langle M \rangle}$ から 既存のマルチステージ言語 MetaOCaml への変換を与え, 提案言語の実現手法を示した. Inoue らによる研究論文で, MetaOCaml の既存の言語機能を用いて擬似的にモジュールの生成を表現するプログラム手法が提案されているが, モジュールの生成を扱うようなプログラムを MetaOCaml 上で表現するための具体的なエンコーディングは与えられていない. 本研究は提案したモジュールの生成を扱えるようなマルチステージ言語から MetaOCaml 上の表現への具体的な変換アルゴリズムを定義した点に新規性がある.

今後の課題としてまず挙げられるのは, 提案体系の理論的性質, 特に進行性 (Progress) や型の保存性 (Presevation) の証明である. 他には提案言語 $\lambda^{\langle M \rangle}$ から MetaOCaml への変換器を実装し, モジュール生成の有効性を様々な例について確かめたいと考えている. 現在の $\lambda^{\langle M \rangle}$ は translucent signature, sharing, subtyping など現在のモジュール・システムでは広く用いられている言語機能のいくつかをサポートしていないので, それらについて拡張することも将来課題の一つである.

参 考 文 献

[1] Inoue, J., Kiselyov, O., and Kameyama, Y.: Staging Beyond Terms: Prospects and Challenges, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, New York, NY, USA, ACM, 2016, pp. 103–108.

[2] Leroy, X.: A Modular Module System, *J. Funct. Program.*, Vol. 10, No. 3(2000), pp. 269–303.

[3] Suzuki, K., Kiselyov, O., and Kameyama, Y.: Finally, Safely-extensible and Efficient Language-integrated Query, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and*

Program Manipulation, PEPM '16, New York, NY, USA, ACM, 2016, pp. 37–48.

付録 A　$\lambda^{\langle M \rangle}$ の抽象構文

抽象構文中の v, t はそれぞれ式, 型の変数を表すメタ変数である.
以降の議論ではマルチステージ構成子の. を省略して $\langle e \rangle$ や ~ e と書く.

(Module types)	M	::=	$\mathbf{sig}\ S\ \mathbf{end}$
(Signatures)	S	::=	$\epsilon \mid C;\ S$
(Signature components)	C	::=	$\mathbf{val}\ v : \sigma \mid \mathbf{type}\ t :: \kappa = \sigma$
(Typing environment)	E	::=	ϵ
			$\mid E;\ (\mathbf{val}\ v : \tau)^i \quad \text{where } i = 0, 1$
			$\mid E;\ (\mathbf{type}\ t :: \kappa = \tau)$
(Kinds)	κ	::=	$*$
(Small Types)	σ	::=	$t \mid v.t \mid (\Downarrow v).t \mid \sigma \to \sigma \mid \sigma\ \mathbf{code} \mid \%\sigma$
(Large Types)	τ	::=	$\sigma \mid \tau \to \tau \mid (\mathbf{module}\ M) \mid (\mathbf{module}\ M)\ \mathbf{code}$

提案するマルチステージ言語 $\lambda^{\langle M \rangle}$　型の抽象構文

(Module expressions)	m	::=	$\mathbf{struct}\ s\ \mathbf{end}$
(Structures)	s	::=	$\epsilon \mid c;\ s$
(Structure components)	c	::=	$\mathbf{val}\ v : \sigma = e \mid \mathbf{type}\ t :: \kappa = \sigma$
(Simple Expressions)	e	::=	$v \mid v.v \mid (\Downarrow v).v \mid \mathbf{fun}\ (v : \sigma) \to e \mid \mathbf{let}\ v = e\ \mathbf{in}\ e$
			$\mid e\ @\ e \mid \langle e \rangle \mid !\, e \mid \sim e$
(General Expressions)	g	::=	$e \mid \mathbf{fun}\ (v : \tau) \to g \mid \mathbf{let}\ v = g\ \mathbf{in}\ g \mid g\ @\ g \mid (\mathbf{module}\ m : M)$
			$\mid \langle g \rangle \mid \mathbf{module\_run}\ g$

提案するマルチステージ言語 $\lambda^{\langle M \rangle}$　式の抽象構文

付録 B　$\lambda^{\langle M \rangle}$ の型システム

付録 B. 1　型に関する規則

$\boxed{E \vdash M\ \mathbf{wf}}$

$$\frac{E\ \mathbf{wf} \qquad E \vdash S\ \mathbf{wf}}{E \vdash (\mathbf{sig}\ S\ \mathbf{end})\ \mathbf{wf}} \quad (1)$$

$\boxed{E \vdash^i S\ \mathbf{wf}}$

$$\frac{E\ \mathbf{wf}}{E \vdash \epsilon\ \mathbf{wf}} \quad (2)$$

$$\frac{E \vdash \tau :: \kappa \qquad E \vdash S\ \mathbf{wf} \qquad v \notin \mathrm{Dom}(E)}{E \vdash (\mathbf{val}\ v : \sigma;\ S)\ \mathbf{wf}} \quad (3)$$

$$\frac{E \vdash \tau :: \kappa \qquad E;\ (\mathbf{type}\ t :: \kappa = \sigma) \vdash S\ \mathbf{wf} \qquad t \notin \mathrm{Dom}(E)}{E \vdash (\mathbf{type}\ t :: \kappa = \sigma;\ S)\ \mathbf{wf}} \quad (4)$$

$\boxed{E \vdash \tau :: \kappa \quad \textbf{and} \quad E \vdash \sigma :: \kappa}$

$$\frac{E = E_1;\ (\mathbf{type}\ t :: \kappa = \tau);\ E_2}{E \vdash t :: \kappa} \quad (5)$$

$$\frac{E = E_1;\ (\mathbf{val}\ v : (\mathbf{module}\ (\mathbf{sig}\ S\ \mathbf{end})));\ E_2 \qquad S = S_1;\ (\mathbf{type}\ t :: \kappa = \sigma);\ S_2}{E \vdash v.t :: \kappa} \quad (6)$$

$$\frac{E = E_1;\ (\mathbf{val}\ v : (\mathbf{module}\ M)\ \mathbf{code})^0;\ E_2 \qquad (\Downarrow (\mathbf{module}\ M)\ \mathbf{code}) = \mathbf{sig}\ S_1;\ (\mathbf{type}\ t :: \kappa = \sigma\ \mathbf{code});\ S_2\ \mathbf{end}}{E \vdash (\Downarrow v).t :: \kappa} \quad (7)$$

$$\frac{E \vdash \sigma_1 :: * \qquad E \vdash \sigma_2 :: *}{E \vdash (\sigma_1 \to \sigma_2) :: *} \quad (8)$$

$$\frac{E \vdash \sigma :: *}{E \vdash \sigma\ \mathbf{code}\ :: *} \quad (9)$$

$$\frac{E \vdash M\ \mathbf{wf}}{E \vdash (\mathbf{module}\ M) :: *} \quad (10)$$

$$\frac{E \vdash M\ \mathbf{wf}}{E \vdash (\mathbf{module}\ M)\ \mathbf{code}\ :: *} \quad (11)$$

$$\frac{E \vdash \tau_1 :: * \qquad E \vdash \tau_2 :: *}{E \vdash (\tau_1 \to \tau_2) :: *} \quad (12)$$

$\boxed{\Downarrow M \quad \textbf{and} \quad \Downarrow C}$

$$\begin{aligned} \Downarrow (\mathbf{sig}\ C_1;\ ...;\ C_n\ \mathbf{end}) &= (\mathbf{sig}\ \Downarrow C_1;\ ...;\ \Downarrow C_n\ \mathbf{end}) \\ \Downarrow (\mathbf{val}\ v : \sigma) &= \mathbf{val}\ v : \sigma\ \mathbf{code} \\ \Downarrow (\mathbf{type}\ t :: \kappa = \tau) &= \mathbf{type}\ t :: \kappa = \tau \end{aligned}$$

付録 B. 2　式に関する規則

$\boxed{E \vdash^i m : M}$

$$\frac{E \vdash^i s : S}{E \vdash^i (\mathbf{struct}\ s\ \mathbf{end}) : (\mathbf{sig}\ S\ \mathbf{end})} \quad (13)$$

$\boxed{E \vdash^i s : S}$

$$\frac{}{E \vdash^i \epsilon : \epsilon} \quad (14)$$

$$\frac{E \vdash^i e : \tau \qquad E \vdash \sigma :: \kappa \qquad E;\ (\mathbf{val}\ v : \sigma)^i \vdash^i s : S \qquad v \notin \mathrm{Dom}(E)}{E \vdash^i (\mathbf{val}\ v : \sigma = e;\ s) : (\mathbf{val}\ v : \sigma;\ S)} \quad (15)$$

$$\frac{E;\ (\mathbf{type}\ t :: \kappa = \sigma) \vdash^i s : S \qquad t \notin \mathrm{Dom}(E)}{E \vdash^i (\mathbf{type}\ t :: \kappa = \sigma;\ s) : (\mathbf{type}\ t :: \kappa = \sigma;\ S)} \quad (16)$$

$\boxed{E \vdash^i e : \sigma}$

$$\frac{E = E_1;\ (\mathbf{val}\ v : \sigma)^i;\ E_2}{E \vdash^i v : \sigma} \quad (17)$$

$$\frac{E \vdash^0 v_1 : (\mathbf{module\ sig}\ S\ \mathbf{end}) \qquad S = S_1;\ (\mathbf{val}\ v_2 : \sigma);\ S_2}{E \vdash^0 v_1.v_2 : \sigma\ \{t \leftarrow v_1.t\ \mid\ t \in \mathrm{Dom}(S_1)\}} \quad (18)$$

$$\frac{E = E_1;\ (\mathbf{val}\ v_1 : (\mathbf{module}\ M)\ \mathbf{code})^0;\ E_2 \qquad (\Downarrow (\mathbf{module}\ M)\ \mathbf{code}) = \mathbf{sig}\ S_1;\ (\mathbf{val}\ v_2 : \sigma\ \mathbf{code});\ S_2\ \mathbf{end}}{E \vdash^0 (\Downarrow v_1).v_2 : \sigma\ \mathbf{code}} \quad (19)$$

$$\frac{E \vdash \sigma_1 :: \kappa \qquad E;\ (\mathbf{val}\ v : \sigma_1)^i \vdash^i e : \sigma_2}{E \vdash^i (\mathbf{fun}\ (v : \sigma_1) \to e) : (\sigma_1 \to \sigma_2)} \quad (20)$$

$$\frac{E \vdash^i e_1 : \sigma_1 \to \sigma_2 \qquad E \vdash^i e_2 : \sigma_1}{E \vdash^i e_1\ @\ e_2 : \sigma_2} \quad (21)$$

$$\frac{E \vdash^i e_1 : \sigma_1 \qquad E; (\mathbf{val}\ v : \sigma_1)^i \vdash^i e_2 : \sigma_2}{E \vdash^i \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \sigma_2} \quad (22)$$

$$\frac{E \vdash^1 e : \sigma}{E \vdash^0 \langle e \rangle : \sigma\ \mathbf{code}} \quad (23)$$

$$\frac{E \vdash^0 e : \sigma\ \mathbf{code}}{E \vdash^1 .^{\sim} e : \sigma} \quad (24)$$

$$\frac{E \vdash^0 e : \sigma\ \mathbf{code}}{E \vdash^0 !\, e : \sigma} \quad (25)$$

$\boxed{E \vdash^i g : \tau}$

$$\frac{E = E_1;\ (\mathbf{val}\ v : \tau)^i;\ E_2}{E \vdash^i v : \tau} \quad (26)$$

$$\frac{E \vdash \tau_1 :: \kappa \qquad E;\ (\mathbf{val}\ v : \tau_1)^i \vdash^i g : \tau_2}{E \vdash^i (\mathbf{fun}\ (v : \tau_1) \to g) : (\tau_1 \to \tau_2)} \quad (27)$$

$$\frac{E \vdash^i g_1 : \tau_1 \to \tau_2 \qquad E \vdash^i g_2 : \tau_1}{E \vdash^i g_1\ @\ g_2 : \tau_2} \quad (28)$$

$$\frac{E \vdash^i g_1 : \tau_1 \qquad E; (\mathbf{val}\ v : \tau_1)^i \vdash^i e_2 : \tau_2}{E \vdash^i \mathbf{let}\ v = g_1\ \mathbf{in}\ g_2 : \tau_2} \quad (29)$$

$$\frac{E \vdash^1 g : (\mathbf{module}\ M)}{E \vdash^0 \langle g \rangle : (\mathbf{module}\ M)\ \mathbf{code}} \quad (30)$$

$$\frac{E \vdash^0 g : (\mathbf{module}\ M)\ \mathbf{code}}{E \vdash^0 \mathbf{module\_run}\ g : (\mathbf{module}\ M)} \quad (31)$$

付録 C　$\lambda^{\langle M \rangle}$ から MetaOCaml への変換

$\boxed{[\![M]\!]^0}$

$$[\![\mathbf{sig}\ C_1;\ \ldots;\ C_n\ \mathbf{end}]\!]^0 \quad = \quad \mathbf{sig}\ [\![C_1]\!]^0;\ \ldots;\ [\![C_n]\!]^0\ \mathbf{end}$$

$\boxed{[\![C]\!]^0}$

$$\begin{aligned} [\![\mathbf{val}\ v : \sigma]\!]^0 &= \mathbf{val}\ v : [\![\sigma]\!]^0 \\ [\![\mathbf{type}\ t :: * = \sigma]\!]^0 &= \mathbf{type}\ t :: * = [\![\sigma]\!]^0 \end{aligned}$$

$\boxed{[\![\sigma]\!]^0}$

$$\begin{aligned} [\![t]\!]^0 &= t \\ [\![v.t]\!]^0 &= v.t \\ [\![(\Downarrow v).t]\!]^0 &= v.t \\ [\![\sigma_1 \to \sigma_2]\!]^0 &= [\![\sigma_1]\!]^0 \to [\![\sigma_2]\!]^0 \\ [\![\sigma\ \mathbf{code}]\!]^0 &= [\![\sigma]\!]^0\ \mathbf{code} \end{aligned}$$

$\boxed{[\![\tau]\!]^0}$

$$\begin{aligned} [\![\tau_1 \to \tau_2]\!]^0 &= [\![\tau_1]\!]^0 \to [\![\tau_2]\!]^0 \\ [\![(\mathbf{module}\ M)]\!]^0 &= (\mathbf{module}\ [\![M]\!]^0) \\ [\![(\mathbf{module}\ M)\ \mathbf{code}\ ]\!]^0 &= (\mathbf{module}\ [\![M]\!]^1) \end{aligned}$$

$\boxed{[\![m]\!]^0}$

$$[\![\mathbf{struct}\ c_1;\ \ldots;\ c_n\ \mathbf{end}]\!]^0 \quad = \quad \mathbf{struct}\ [\![c_1]\!]^0;\ \ldots;\ [\![c_n]\!]^0\ \mathbf{end}$$

$\boxed{[\![c]\!]^0}$

$$\begin{aligned} [\![\mathbf{val}\ v : \sigma = e]\!]^0 &= \mathbf{val}\ v : [\![\sigma]\!]^0 = [\![e]\!]^0 \\ [\![\mathbf{type}\ t :: \kappa = \sigma]\!]^0 &= \mathbf{type}\ t :: \kappa = [\![\sigma]\!]^0 \end{aligned}$$

$\boxed{[\![e]\!]^0}$

$$
\begin{array}{rcl}
[\![v]\!]^0 & = & v \\
[\![v_1.v_2]\!]^0 & = & v_1.v_2 \\
[\![(\Downarrow v_1).v_2]\!]^0 & = & v_1.v_2 \\
[\![\mathbf{fun}\ (v:\sigma) \to e]\!]^0 & = & \mathbf{fun}\ (v : [\![\sigma]\!]^0) \to [\![e]\!]^0 \\
[\![e_1\ @\ e_2]\!]^0 & = & [\![e_1]\!]^0\ @\ [\![e_2]\!]^0 \\
[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!]^0 & = & \mathbf{let}\ v = [\![e_1]\!]^0\ \mathbf{in}\ [\![e_2]\!]^0 \\
[\![\langle e \rangle]\!]^0 & = & \langle [\![e]\!]^0 \rangle \\
[\![\ !\ e]\!]^0 & = & !\ [\![e]\!]^0
\end{array}
$$

$[\![g]\!]^0$

$$
\begin{array}{rcl}
[\![\mathbf{fun}\ (v:\tau) \to g]\!]^0 & = & \mathbf{fun}\ (v : [\![\tau]\!]^0) \to [\![g]\!]^0 \\
[\![g_1\ @\ g_2]\!]^0 & = & [\![g_1]\!]^0\ @\ [\![g_2]\!]^0 \\
[\![\mathbf{let}\ v = g_1\ \mathbf{in}\ g_2]\!]^0 & = & \mathbf{let}\ v = [\![g_1]\!]^0\ \mathbf{in}\ [\![g_2]\!]^0 \\
[\![(\mathbf{module}\ m : M)]\!]^0 & = & (\mathbf{module}\ [\![m]\!]^0 : [\![M]\!]^0) \\
[\![\langle(\mathbf{module}\ m : M)\rangle]\!]^0 & = & (\mathbf{module}\ [\![m]\!]^1 : [\![M]\!]^1) \\
[\![(\mathbf{module\_run}\ g : M)]\!]^0 & = & (\text{付録 C.1 を参照})
\end{array}
$$

$[\![M]\!]^1$

$$
[\![\mathbf{sig}\ C_1;\ \ldots;\ C_n\ \mathbf{end}]\!]^1 \quad = \quad \mathbf{sig}\ [\![C_1]\!]^1;\ \ldots;\ [\![C_n]\!]^1\ \mathbf{end}
$$

$[\![C]\!]^1$

$$
\begin{array}{rcl}
[\![\mathbf{val}\ v : \tau]\!]^1 & = & \mathbf{val}\ v : [\![\tau]\!]^1\ \mathbf{code} \\
[\![\mathbf{type}\ t :: * = \tau]\!]^1 & = & \mathbf{type}\ t :: * = [\![\tau]\!]^1
\end{array}
$$

$[\![\sigma]\!]^1$

$$
\begin{array}{rcl}
[\![t]\!]^1 & = & t \\
[\![v.t]\!]^1 & = & v.t \\
[\![(\Downarrow v).t]\!]^1 & = & v.t \\
[\![\sigma_1 \to \sigma_2]\!]^1 & = & [\![\sigma_1]\!]^1 \to [\![\sigma_2]\!]^1 \\
[\![\%\sigma]\!]^1 & = & [\![\sigma]\!]^1
\end{array}
$$

$[\![\tau]\!]^1$

$$
\begin{array}{rcl}
[\![\tau_1 \to \tau_2]\!]^1 & = & [\![\tau_1]\!]^1 \to [\![\tau_2]\!]^1 \\
[\![(\mathbf{module}\ M)]\!]^1 & = & (\mathbf{module}\ [\![M]\!]^1)
\end{array}
$$

$[\![m]\!]^1$

$$[\![\textbf{struct } c_1;\ \ldots\ ;\ c_n \textbf{ end}]\!]^1 \quad = \quad \textbf{struct } [\![c_1]\!]^1;\ \ldots\ ;\ [\![c_n]\!]^1 \textbf{ end}$$

$[\![c]\!]^1$

$$\begin{aligned}
[\![\textbf{val } v : \sigma = e]\!]^1 &= \textbf{val } v : [\![\sigma]\!]^1 = \langle [\![e]\!]^1 \rangle \\
[\![\textbf{type } t :: \kappa = \sigma]\!]^1 &= \textbf{type } t :: \kappa = [\![\sigma]\!]^1
\end{aligned}$$

$[\![e]\!]^1$

$$\begin{aligned}
[\![v]\!]^1 &= v \\
[\![v_1.v_2]\!]^1 &= v_1.v_2 \\
[\![(\Downarrow v_1).v_2]\!]^1 &= v_1.v_2 \\
[\![\textbf{fun } (v : \sigma) \to e]\!]^1 &= \textbf{fun } (v : [\![\sigma]\!]^1) \to [\![e]\!]^1 \\
[\![e_1 \ @ \ e_2]\!]^1 &= [\![e_1]\!]^1 \ @ \ [\![e_2]\!]^1 \\
[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]^1 &= \textbf{let } v = [\![e_1]\!]^1 \textbf{ in } [\![e_2]\!]^1 \\
[\![\sim e]\!]^1 &= \sim [\![e]\!]^0
\end{aligned}$$

$[\![g]\!]^1$

$$\begin{aligned}
[\![\textbf{fun } (v : \tau) \to g]\!]^1 &= \textbf{fun } (v : [\![\tau]\!]^1) \to [\![g]\!]^1 \\
[\![g_1 \ @ \ g_2]\!]^1 &= [\![g_1]\!]^1 \ @ \ [\![g_2]\!]^1 \\
[\![\textbf{let } v = g_1 \textbf{ in } g_2]\!]^1 &= \textbf{let } v = [\![g_1]\!]^1 \textbf{ in } [\![g_2]\!]^1 \\
[\![(\textbf{module } m : M)]\!]^1 &= (\textbf{module } [\![m]\!]^1 : [\![M]\!]^1)
\end{aligned}$$

付録 C. 1　module_run の変換

module_run はモジュール・コードの実行を行うためのプリミティブであり, コードのモジュール内の **val** $v : \sigma = e$ の形をした要素たちの右辺にコードの実行を表すマルチステージ構成子 **!** を挿入する. このような **module_run** は, 単純な変換では実現することができない. 例えば, `fun (v : (module M)code)-> module_run v`という式中の `m` では **module_run** の対象である `v` は変数なので, 関数に具体的なモジュール・コードが適用されるまで, **!** を挿入することができない.

そこで **module_run** g という形の式に限り, **module_run** の対象となる式 `g` の型を基に変換を行うことにする. g の型が `module (sig type t; val v1 : σ1; val v2 : σ2 end)`のとき, 以下のようなプログラムに変換する.

```
[[ module_run g ]]^0
= module struct
      val g' = ⇓ g
      val v1 = run g'.v1
      val v2 = run g'.v2
  end
```