継続の適用をジョインポイントとするアスペクト指向プログラミングモデル

A Model for Aspect-Oriented Programming that Regards Applications to Continuations as Join Points

遠藤 侑介 † 増原 英彦 ‡ 米澤 明憲 †

Yusuke Endoh Hidehiko Masuhara Akinori Yonezawa

† 東京大学 大学院情報理工学系研究科
Graduate School of Information Science and Technology, the University of Tokyo

‡ 東京大学 大学院総合文化研究科
Graduate School of Arts and Sciences, the University of Tokyo

{mame,yonezawa}@yl.is.s.u-tokyo.ac.jp, masuhara@acm.org

本論文では継続の適用をジョインポイントとし、ジョインポイントを一連の動作の実行期間ではなく時間上の一点と定義するアスペクト指向言語の実行モデルを提案する。従来のアスペクト指向プログラミングでは、アドバイスの実行のタイミングがアドバイスの修飾子で指示されていた。提案するモデルでは関数に値を渡す時点と関数の返り値を受け取る時点をそれぞれジョインポイントとするため、タイミングの指示をポイントカットとして記述できる。これによりプログラミング形式の違いをポイントカットで吸収でき、アスペクトが作用する対象の変更により柔軟に対応できる。さらにこのモデルには、継続渡し形式を用いた簡潔な表示的意味が与えられる。この定式化は他の言語機能との相性が良い。

1 はじめに

アスペクト指向プログラミング (AOP) は、ロギングやセキュリティ検査などのようにモジュール間を横断する関心事 (crosscutting concerns) をモジュール化するプログラミングパラダイムである [6]。AOP 言語の多くはアスペクトというモジュールを持ち、横断的関心事を 1 つのアスペクトとしてまとめて記述できる。

現在普及している多くの AOP 言語 [1, 2, 4, 5, 7–9] はポイントカット・アドバイス機構を持つ。この機構では例えば「全てのモジュールにおいて、ユーザから入力があった場合にログをとる」という横断的関心事を、アスペクトというポイントカットとアドバイスの組の集合によってモジュール化できる。ポイントカットとアドバイスの働きは**ジョインポイント**というプログラム実行中の特定の種類の「動作」によって説明される。例えば多くの AOP 言語は「関数呼び出し」をジョインポイントとしている。ポイントカットではジョインポイントが満たすべき条件 (例えば「入力を受け取る関数の呼び出し」) を記述し、アドバイスではそのジョインポイントで実行すべき挙動 (例えば「ログをとる」) を記述する。

この機構の特徴は、横断的関心事の仕様やアスペクト以外のモジュールの定義が変更され、アスペクトが作用すべき対象が変化したときに、ポイントカットの修正だけでアスペクトを対応できることである。しかし従来のモデルでは、この変化が些細なものであってもポイントカットの修正だけでは対応できずアドバイス定義を変更・複製しなければならない場合がある。この事例は 2 節で詳説する。

この問題の原因は、従来のジョインポイントが時間上の一点ではなく一連の動作の実行期間として定義され (例えば call ジョインポイントは「関数を呼び出し、関数本体を実行し、関数から帰ってくるまで」)、その実行期間の始点・終点を表す `before` や `after` といった修飾をアドバイスに施すことで実行のタイミングを指示するという設計にある。以下ではこのような従来のモデルを**区間モデル**と呼ぶ。

本研究では、区間モデルのジョインポイントの始点と終点をそれぞれ異なる 2 つのジョインポイントとすることで、ジョインポイントを時間上の一点とするアスペクト指向言語の実行モデルを提案する。このモデルを**点モデル**と呼ぶ。例えば call ジョインポイントは「関数を呼び出す」だけの時点とする・また、「関数から帰ってくる」時点に対し return ジョインポイントという新たなジョインポイントを割り当てる。これにより `before` 、`after` で表現していたタイミングの指示もポイントカットとして記述する

ことができ、アスペクト以外のモジュールの変更やアスペクトへの要請の追加・変更に区間モデルより柔軟に対応できる。

さらに本研究は、点モデルに継続渡し形式 (Continuation Passing Style) を用いた表示的意味を与える。return ジョインポイントは関数に渡された継続を適用する時点に相当する。このため call ジョインポイントと return ジョインポイントは継続渡し形式の上で λ 項に値を渡す時点として統一して扱うことができる。さらにこの定式化は、他の言語機能やジョインポイントも組み込みやすく、複数の言語機能がからみあう場合も意味論で明確に表現できる。このため AOP 研究者が新たな言語機能を考える土台となることが期待できる。

本論文の以降の構成は以下の通りである。次節では区間モデルでは簡潔に記述できない事例を説明する。3 節で本研究が提案する点モデルの直感的な説明を行う。続いて 4 節で点モデルの定式化を概説し、5 節で他の言語機能との連携を議論する。7 節で本論文をまとめる。

2 区間モデルの問題点

例として「ユーザの入力のログをとる」という横断的関心事を考え、プログラムが (1) コンソールから入力を受け取る場合と、(2) コンソールと GUI 部品の両方から受け取る場合について、横断的関心事を実現するアスペクトの定義を見る。本来アスペクトが作用する対象が変わった場合はポイントカットの修正だけで済むはずであるが、(1) から (2) への変更ではそれ以外の変更も必要となることを見る。

2.1 コンソールから入力を受け取る場合

図 1 は Java の AOP 拡張である AspectJ [5] で記述された、ユーザの入力のログをとるアスペクトである。プログラム実行中 readLine メソッドの呼び出しがあった直後に、「その返り値である文字列を記録する」という挙動を実行する。

図 1 の 2・3 行目がポイントカット宣言である。この userInput というポイントカットは、「readLine メソッドを呼び出す」という条件を満たす全てのジョインポイントにマッチする。4 – 7 行目がアドバイスである。4 行目の after という修飾子が「マッチしたジョインポイントの後で実行する」ことを指示し、returning(String str) がメソッド呼び出しの返り値を str に束縛することを指示している。アドバイス本体の 6 行目でその返り値を記録する。

```
1 aspect UserInputLogging {
2   pointcut userInput():
3     call(String *.readLine(void));
4   after() returning(String str)
5     : userInput() {
6     Log.add(str);
7   }
8 }
```

図 1: コンソールから入力のログを受け取るアスペクト

ここでもし、アスペクト以外のモジュールが環境変数を読むようになり、これもユーザの入力としてログをとる必要がある場合は、次のように環境変数を読む時点をポイントカットに追加する。

```
2   pointcut userInput():
3     call(String *.readLine(void)) ||
4     call(String System.getenv(String));
```

この例では、アスペクト以外のモジュールの変更が些細であっても、アスペクトのポイントカットの修正だけで対応できている。

2.2 コンソールと GUI 部品の両方から入力を受け取る場合

ここで、プログラムがコンソールだけでなく GUI からの入力もサポートする場合を考える。この GUI フレームワークは、ユーザがボタンを押した時、ユーザの入力テキストを実引数として onSubmit(String) メソッドを呼び出すものとする。

この GUI からの入力も同じアスペクトでログをとるためには、アスペクトは図 2 のように変更しなければならない。図 1 は「ユーザの入力は何らかのメソッド呼び出しが 終了した ときにその 返り値 として得られる」と仮定しているのに対し、ここでは、ユーザの入力を受け取るのが onSubmit メソッド呼び出しを 開始した ときにその 実引数 として得られる。従って図 1 のポイントカットの変更だけでは対応できない。

4 – 6 行では、onSubmit メソッド呼び出しのジョインポイントを指示し、さらにその呼び出しの実引数を変数 str に束縛するポイントカット userInput2

```
 1 aspect UserInputLogging {
 2   pointcut userInput():
 3     call(String *.readLine(void));
 4   pointcut userInput2(String str):
 5     call(String *.onSubmit(String))
 6     && args(str);
 7   after() returning(String str)
 8     : userInput() {
 9     Log.add(str);
10   }
11   before(String str)
12     : userInput2(str) {
13     Log.add(str);
14   }
15 }
```

図 2: コンソールと GUI 部品の両方から入力を受け取るアスペクト

が宣言されている。11 – 14 行ではアドバイスの定義が追加されている。このアドバイスは onSubmit メソッドの呼び出しの直前で、その呼び出しの実引数である文字列を記録する。

2.3 問題点のまとめ

ポイントカット・アドバイス機構の特徴は、アスペクトが作用する対象の変化に対してポイントカットの修正だけで対応できることであった。それにもかかわらず、アスペクト以外のモジュールの変更に対し、アスペクトにアドバイスの定義を複製するというポイントカット以外の修正を迫られる事例がある。この問題の根本的な原因は、区間モデルでは 1 つのジョインポイントが一連の動作の実行期間として定義され、その始点である「返り値を受け取る点」と、終点である「実引数を与える点」とをポイントカット以外の機構、つまりアドバイスで区別していることである。

今回説明した事例では、アドバイスの本体部分が Log.add(str); だけなのでこれだけの修正で済んでいるが、実際の事例ではもっと長くなるためメソッドにくくり出す必要があるだろう。さらに userInput をポイントカットに持つアドバイスが複数あった場合、そのアドバイスの数だけ修正を行うことになる。

また、アスペクトをライブラリとして定義する場合の手間も増大させる。AspectJ のような言語では「ログを取る」ような汎用的な横断的関心事は、抽象ポイントカットとアスペクトの継承機構によってライブラリ化できることが知られている。しかし、ライブラリ定義時にジョインポイントの前後どちらでログを取るかがわからないと、同じ動作をするアドバイスを複数定義し、それぞれに抽象ポイントカットを定義しなければならなくなる。

一般に、ある処理に値を渡すには、メソッド呼び出しの引数として渡すやり方と、メソッドの返り値として渡されるやり方がある。そのやり方を変更した場合には常に今回の問題が起こり得る。例えば、イベントドリブンとポーリング、ブロッキング I/O とノンブロッキング I/O 、何らかの失敗を通知するのにエラーを表す定数を返す場合と例外を発生させる場合などのプログラミング形式を変更する場合や、スタンドアローンアプリケーションをアプリケーションフレームワーク向けに修正する場合が挙げられる。関数型言語においては、返り値を受け取っていたところを返り値を受け取るクロージャを渡す形式 (継続渡し形式) でプログラムを書きたくなることがしばしばある。

3 継続ジョインポイント

3.1 概要

本研究が提案するジョインポイントモデル (点モデル) は区間モデルに対して以下の変更と追加を行ったものである。

- ジョインポイントは一連の動作の実行期間ではなく、その始まりの時点のみとして定義する。
- before や after など、アドバイスの修飾子は排除する。
- メソッド呼び出しなどの終わりの時点を指す、新しい種類のジョインポイントを追加する。
- 新しく導入したジョインポイントにマッチするポイントカットを導入する。

以下、区間モデルの call ジョインポイントを例にとって説明する。call ジョインポイントは 区間モデルでは「関数を呼び出し、関数本体を実行し、関数から帰ってくるまで」という一連の動作の実行期間を指していた (図 3) 。点モデル (図 4) では、call ジョインポイントは「関数を呼び出す」時点のみとなる。

さらに「関数から帰ってくる」時点に対応するジョインポイント return を導入し、加えてこのジョインポイントを指示するポイントカット return(f) も導入する。以上より

```
before(): call(*.onSubmit(String))
after() returning: call(*.readLine(void))
```

と記述していたところは、それぞれ

```
advice(): call(*.onSubmit(String))
advice(): return(*.readLine(void))
```

と書くことになる。

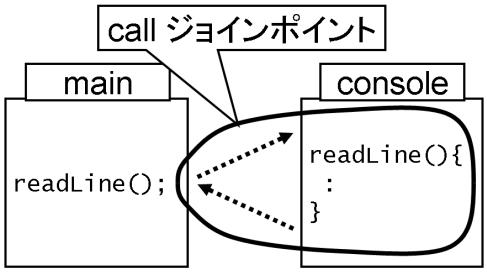

図 3: 従来モデルの call ジョインポイント

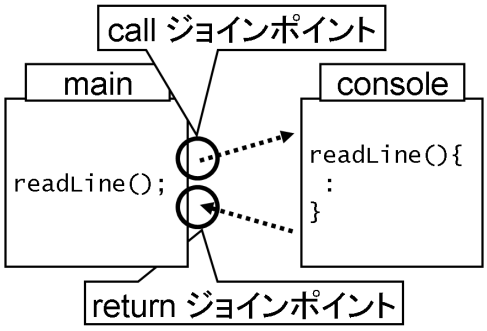

図 4: 提案モデルの call ジョインポイント

従来 call ジョインポイントに対する args(x) ポイントカットは、実引数を変数 x に束縛するものであった。これは提案モデルの call ジョインポイントに対しても同じである。また return ジョインポイントに対しては、返り値を x に束縛するものとする。従って

```
before(String str):
  call(*.onSubmit(String)) && args(str)
after() returning(String str):
  call(*.readLine(void))
```

と記述していたところは、それぞれ

```
advice(String str):
  call(*.onSubmit(String))
advice(String str):
  return(*.readLine(void))
```

と書くことになる。

3.2 点モデルによる「ログをとる」アスペクト

提案したモデルで前節のアスペクト 2 つを記述した例がそれぞれ図 5 と図 6 である。

```
1 aspect UserInputLogging {
2   pointcut userInput(String str):
3     return(String *.readLine(void)) &&
4     args(str);
5   advice(String str): userInput(str) {
6     Log.add(str);
7   }
8 }
```

図 5: 提案モデルで記述されたコンソールから入力のログを受け取るアスペクト

```
2   pointcut userInput(String str):
3     (return(String *.readLine(void)) ||
4      call(String *.onSubmit(void))) &&
5     args(str);
```

図 6: 提案モデルで記述されたコンソールと GUI 部品の両方から入力を受け取るアスペクトの変更点

コンソールから入力のログを受け取るアスペクトは、細かな文法的差異を除けば図 1 と変わらない。図 5 の 2 – 4 行目で readLine メソッド呼び出しの返り値を束縛する userInput ポイントカットを宣言している。5 – 7 行目ではポイントカット userInput を満たす点に対しログをとるアドバイスを定義している。

これに対する修正は 4 行目にポイントカット宣言を 1 行分追加するだけである。これによって userInput

ポイントカットは「readLine メソッドからの返り値を str に束縛する」または「onSubmit メソッド呼び出しの実引数を str 束縛する」ポイントカットとなる。アドバイスの定義は一切修正されておらず、ポイントカットの修正のみで対応できている。

このように、メソッド呼び出しの引数として渡すやり方と、メソッドの返り値として渡されるやり方の違いはポイントカットだけで吸収できる。またアドバイスの定義時にジョインポイントの前後どちらを指定すべきか分からなくても、そのアスペクトを使うユーザにポイントカットで指定して貰うことができる。

4 定式化

この章では、他の言語機能を点モデルに組み込むことを議論するために定式化を行う。他の機能の例としては、例外機構や一級継続、内部イテレータなどの AOP に限らない言語機能や、call 以外のジョインポイント、around と proceed、cflow など従来の AOP 言語に備わっていた機能などが挙げられる。

意味論は例外機構を持つ型無し ML のサブセットに、点モデルのアスペクト機構を追加した言語に対して定義した。本文中ではその主要部分のみを示し、全体は Appendix に掲載する。

4.1 言語の文法と直感的な意味

言語の文法は図 7 に定義される。本文中では形無し λ 計算にアドバイス機構を追加した言語としている。

(Expression)

e	::=	x	(IDENTIFIER)
	|	$\texttt{fun}\, x \to e$	(FUNCTION)
	|	$e\, e$	(APPLICATION)

(Pointcut)

p	::=	$\texttt{call} \mid \texttt{return}$

(Advice)

a	::=	$\texttt{advice}(x) : p \to e$

図 7: 言語の文法

さらに説明を簡単化するため、以下の制限を行っている。

- アドバイスは常に 1 つだけ定義されているものとしている。Appendix に掲載されている定式化では複数個のアドバイスが定義できる。
- ポイントカットは call と return のみで、どちらも引数はとらないものとする。このポイントカットはそれぞれすべての call ジョインポイント、すべて return ジョインポイントにマッチする。Appendix に掲載されている定式化では、これらのポイントカットは関数名を引数をとり、その関数の各ジョインポイントにマッチする。

アドバイスは常に引数を 1 つとる関数として定義される。アドバイスは各ジョインポイントに対して次にように実行される。

call **ジョインポイントの場合** ジョインポイントになる関数適用の実引数にアドバイスを適用し、その結果の値を新たな実引数として関数を呼び出す。

return **ジョインポイントの場合** ジョインポイントになる関数適用の返り値にアドバイスを適用し、その結果の値を新たな返り値として返す。

前節での説明と異なる点が 2 つある。1 つは args ポイントカットが明示的には存在しない点であり、もう 1 つはアドバイスの返り値が新たな実引数や返り値として利用される点である。

アドバイスの例を挙げる。`advice(x):call → "Hello, " + x` というアドバイスは、すべての関数適用で引数の文字列の先頭に `"Hello, "` という文字列を連結した値に差し替える。例えば `onSubmit "world"` という関数適用があったときは、関数 `onSubmit` には `"Hello, world"` という引数が与えられる。同様に `advice(x):return → "Hello, " + x` というアドバイスは関数から帰ってくる時点で返り値の先頭に `"Hello, "` という文字列を連結した値に差し替える。`readLine ()` という式を評価し、この関数が `"world"` という値を返したとすると、これの先頭に `"Hello, "` を連結した値、`"Hello, world"` が得られる。

4.2 言語の意味論

言語の意味定義は、まず、アスペクトのない言語の表示的意味を継続渡し形式によって示し、それにアドバイスを適用する意味を追加する形で行う。

意味代数は図 8 に定義される。ただし *Ide* は識別子、*Int* は整数、*Bool* は真偽値、*Fun* は関数、*Ans* は結果のドメインである。

$$
\begin{aligned}
v &\in Val = Int + Bool + Fun \\
\rho &\in Env = Ide \to Val \\
\kappa &\in Ctn = Val \to Ans \\
\theta &\in Jp = \mathsf{call} + \mathsf{return}
\end{aligned}
$$

図 8: 意味論で使用する意味代数

アスペクトがない場合の APPLICATION の表示的意味論は、式の意味関数 $\mathcal{E}$: Expression $\to$ $Env \to Ctn \to Ans$ を以下のように定義できる:

$$
\begin{aligned}
&\mathcal{E}[\![e_0\ e_1]\!]\,\rho\,\kappa\ = \\
&\quad \mathcal{E}[\![e_0]\!]\,\rho\,(\lambda f.\,\mathcal{E}[\![e_1]\!]\,\rho\,(\lambda v.\,f\,(\lambda r.\,\kappa\,r)\,v))
\end{aligned}
$$

$(f\,(\lambda r.\,\kappa\,r))\,v$ の関数適用が call ジョインポイントとなる。ここでアドバイス $\texttt{advice(}x_a\texttt{):call} \to e_a$ が存在すれば、$\mathcal{E}[\![e_a]\!]\,[v/x_a]\,(\lambda v'.\,f\,(\lambda r.\,\kappa\,r)\,v')$ という意味になる。

return ジョインポイントは、関数呼び出しで渡された継続を返り値に適用する時点に相当する。よって、$\kappa\,r$ の適用が return ジョインポイントとなる。アドバイス $\texttt{advice(}x_a\texttt{):return} \to e_a$ の下では、$\mathcal{E}[\![e_a]\!][r/x_a](\lambda r'.\,\kappa\,r')$ という意味になる。

このように call ジョインポイントも return ジョインポイントも、適用を η-展開[1]しそれを継続としてアドバイスを評価している。従って、アドバイスの意味を意味関数 $\mathcal{A}$ としてくくりだし、不要な η-展開を除去することで以下のように整理できる。

$$
\begin{aligned}
&\mathcal{E}[\![e_0\ e_1]\!]\ \rho\,\kappa = \\
&\ \mathcal{E}[\![e_0]\!]\,\rho\,(\lambda f.\,\mathcal{E}[\![e_1]\!]\,\rho \\
&\quad (\mathcal{A}[\![a]\!]\ \mathsf{call}\ (\lambda v'.\,f\,(\mathcal{A}[\![a]\!]\ \mathsf{return}\ \kappa)\,v')))
\end{aligned}
$$

ただし a は大域的に定義されたアスペクト[2]であり、アスペクトの意味関数 $\mathcal{A}$: Advice $\to$ $Jp \to Ctn \to Ctn$ は

$$
\begin{aligned}
&\mathcal{A}[\![\texttt{advice(}x_a\texttt{):}p \to e_a]\!]\,\theta\,\kappa = \\
&\quad \textbf{if}\ \mathcal{P}[\![p]\!]\,\theta\ \textbf{then}\ (\lambda v.\,\mathcal{E}[\![e_a]\!]\,[v/x_a]\,\kappa)\ \textbf{else}\ \kappa
\end{aligned}
$$

である。ただし $\mathcal{P}[\![p]\!]\,\theta$ はポイントカット p に対しジョインポイント θ がマッチするかどうかを真偽値で返す。

[1]継続を η-展開するため、この定式化を naïve に実装すると末尾呼び出し除去が不可能になる。

[2]実際には式の意味関数 $\mathcal{E}$ に引数として与える。

5 言語の拡張

この章では、例外機構や一級継続など高度な言語機能や区間モデルの AOP 言語にあった他の言語機能が、点モデルでどういう形で実現可能か、また、他の機能にどういう影響を与えるかを議論する。

具体的には、一般の言語にもある機能として例外機構を、区間モデルの AOP 言語にある機能として get ジョインポイント、around と proceed を組み込めることを示す。

5.1 例外機構

例外機構のための構文 $\texttt{try}\ e_0\ \texttt{with}\ x \to e_1$ と $\texttt{raise}\ e$ を導入する (図 9) 。本論文では簡単のため、例外オブジェクトを表す特別な値は導入せず、任意の値を例外として raise することができるものとする。

(Expression)

$$
\begin{aligned}
e \quad ::= \quad & \ldots \\
& \mid \texttt{try}\ e\ \texttt{with}\ x \to e \quad (\textsc{Try}) \\
& \mid \texttt{raise}\ e \quad (\textsc{Raise})
\end{aligned}
$$

図 9: 例外機構の構文を追加した言語

これらの形式的な意味は、意味関数 $\mathcal{E}$ や $\mathcal{A}$ が通常の継続に加え現在のハンドラを表す継続を受け取るようにすることで表現できる。TRY と RAISE の意味関数 $\mathcal{E}$: Expression $\to$ $Env \to Ctn \to Ctn \to Ans$ はそれぞれ以下のようになる:

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{try}\ e_0\ \texttt{with}\ x \to e_1]\!]\,\rho\,\kappa\,\kappa_h\ = \\
&\quad \mathcal{E}[\![e_0]\!]\,\rho\,\kappa\,(\lambda v.\,\mathcal{E}[\![e_1]\!]\,([v/x]\rho)\,\kappa\,\kappa_h) \\
&\mathcal{E}[\![\texttt{raise}\ e]\!]\,\rho\,\kappa\,\kappa_h\ =\ \mathcal{E}[\![e]\!]\,\rho\,\kappa_h\,\kappa_h
\end{aligned}
$$

κ_h が追加された継続である。

さらにアドバイスの意味関数の引数にも現在のハンドラを表す継続を追加する ($\mathcal{A}$: Advice $\to$ $Jp \to Ctn \to Ctn \to Ctn$) 。

$$
\begin{aligned}
&\mathcal{A}[\![\texttt{advice(}x_a\texttt{):}p \to e_a]\!]\,\theta\,\kappa\,\underline{\kappa_h} = \\
&\quad \textbf{if}\ \mathcal{P}[\![p]\!]\,\theta\ \textbf{then}\ (\lambda v.\,\mathcal{E}[\![e_a]\!]\,[v/x_a]\,\kappa\,\underline{\kappa_h})\ \textbf{else}\ \kappa
\end{aligned}
$$

追加された継続 (下線) は、アドバイス実行中に発生した例外を受け取るハンドラを意味する。

アスペクトがない場合の APPLICATION の意味は

次のように変化する:

$$\begin{aligned}&\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa\ \kappa_h\ =\\ &\quad \mathcal{E}[\![e_0]\!]\rho(\lambda f.\ \mathcal{E}[\![e_1]\!]\rho(\lambda v.\\ &\qquad f\ (\lambda r.\ \kappa\ r)\ (\lambda x.\ \kappa_h\ x)\ v)\kappa_h)\kappa_h\end{aligned}$$

この $f\ (\lambda r.\ \kappa\ r)\ (\lambda x.\ \kappa_h\ x)\ v$ の適用が call ジョインポイントとなる。

関数に渡される継続 $(\lambda x.\ \kappa_h\ x)$ が追加されている。これは「関数から例外によって脱出する時点」に相当する。言語設計者が「この時点も関数からの脱出、すなわち return ジョインポイントの 1 つである」と考える場合は、Application の意味を以下のように定義すればよい:

$$\begin{aligned}&\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa\ \kappa_h\ =\\ &\ \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda f.\ \mathcal{E}[\![e_1]\!]\ \rho\\ &\ (\mathcal{A}[\![a]\!]\ \mathsf{call}\ (\lambda v'.\\ &\quad f\ (\mathcal{A}[\![a]\!]\ \mathsf{return}\ \kappa\ \kappa_h)\ (\mathcal{A}[\![a]\!]\ \mathsf{return}\ \kappa_h\ \kappa_h)\ v'\\ &\ )\ \underline{\kappa_h})\ \kappa_h)\ \kappa_h\end{aligned}$$

もし言語設計者が、「例外によって関数から脱出する時点は return ジョインポイントではない」、もしくは「return とは異なる新たなジョインポイントである」と考える場合[3]でも、自明に対応できるだろう。

ここで例外機構とアスペクト機構がからみあう例を考える。例えば、call ジョインポイントで実行中のアドバイス内で例外が発生した場合、その例外はどのハンドラが受け取るだろうか。これは $\mathcal{A}[\![a]\!]$ call の 2 つ目に与えられた継続 (下線) 、すなわち $e_0\ e_1$ の適用があった時点でのハンドラが受け取ることがわかる。このように異なる機能が互いに影響しあう場合でも、どの継続が渡されているかだけに注意すればよく、明確かつシンプルに表現できている。

5.2 get ジョインポイント

AspectJ における get ジョインポイントはフィールドから値を読み出す時点である。これは (区間モデルの) call ジョインポイントなどと異なり、一連の動作の実行期間ではなく実行上の一点である。なぜなら、フィールドから値を読む前後では、AspectJ の言語レベルで観測できる状態の変化が一切おこっていないからである[4]。従って、今回は get ジョインポイントを点モデルに組み込む場合、終点に対応するような新たなジョインポイントを導入しない。

$$\begin{array}{ll}&\text{(Pointcut)}\\ p & ::= \quad \ldots \mid \mathtt{get}\end{array}$$

図 10: get を追加した言語

提案するモデルにはオブジェクト指向 (フィールド) の機能がないため、本論文は変数参照を行う時点として考える。

例えば `advice(x):get` $\rightarrow$ `x` $+ 1$ というアスペクトは、プログラム中すべての変数参照の時点で結果の値に 1 を足す。$(\mathtt{fun\ x} \rightarrow \mathtt{x})\ 3$ という式は、`x` という変数参照を評価するため 4 となる。

Identifier の意味論に注目する。アスペクトがない場合の Identifier の意味は以下のようになる:

$$\mathcal{E}[\![x]\!]\ \rho\ \kappa = \kappa\ (\rho(x))$$

$\kappa\ (\rho(x))$ の適用が get ジョインポイントになるので、アドバイス `advice(`x_a`):get` $\rightarrow e_a$ があると $\mathcal{E}[\![e_a]\!]\ [(\rho(x))/x_a]\ (\lambda r'.\ \kappa\ r')$ という意味になる。従ってアスペクトのある Identifier の意味関数は

$$\mathcal{E}[\![x]\!]\ \rho\ \kappa = \mathcal{A}[\![a]\!]\ \mathsf{get}\ \kappa\ (\rho(x))$$

となる。ただし意味代数 Jp に get を追加している。

$$Jp = \mathsf{call} + \mathsf{return} + \mathsf{get}$$

5.3 around と proceed

区間モデルには before 、after の他に around アドバイスがある。before アドバイスや after アドバイスは実引数や返り値を変更することはできないが、around アドバイスは、実引数や返り値を差し替えることができる。例えば、変数名を自動的に大文字にして、取得した文字例の空白を取り除くアドバイスは

```
String around(String name) :
  call(String System.getenv(String))
  && args(name) {
  name = name.toUpperCase();
  return proceed(name).trim();
}
```

[3] AspectJ において、アドバイスを after returning と after throwing という修飾子で区別する場合に相当する。

[4] 正確には NullPointerException が発生する場合があるが、この定式化では参照型を扱わないため NullPointerException は起こりえない。

のように定義できる。これと同様のことは、点モデルのアドバイスですでに可能である。

さらに around アドバイスでは、proceed を行わない、もしくは複数回行うことができる。proceed を行わない場合は、ジョインポイントに新たな挙動を追加するのではなく新たな挙動で差し替えることができる。例えば、

```
String around() :
  call(String *.readLine()) {
  return "dummy";
}
```

という AspectJ のアドバイスは、readLine メソッドの呼び出しを行わず、即座に "dummy" という文字列を返す。

本研究では現在、同一アドバイス内で 0 回または 1 回だけ proceed を実行するのに相当する機能を定式化に組み込んでいる。同一アドバイス内で複数回 proceed を行う機能については 8 節の今後の展望で触れる。

5.3.1 skip 構文

まず、すべてのジョインポイントに「スキップ先」という概念を持たせる。ここでは簡単に、call ジョインポイントのスキップ先も return ジョインポイントのスキップ先も「その関数呼び出しの終わりの時点の直後」とする。これはつまり return ジョインポイントの直後であり、関数呼び出しを始めた時点での継続の始点に相当する。

アドバイス実行中に「ジョインポイントのスキップ先へジャンプする」構文 skip を導入する。

(Expression)

$$
\begin{array}{lll}
e & ::= & \ldots \\
 & | & \texttt{skip}\, e \quad (\textsc{Skip})
\end{array}
$$

図 11: skip 構文を追加した言語

例えば advice(x):call → skip x というアドバイスは、すべての関数呼び出しを常にキャンセルし、「実引数をそのまま返す」という恒等関数のような振る舞いに変更する。

skip 構文の形式的な意味は、例外機構と同じように意味関数にスキップ先の継続を受け取らせることで表現できる。以下に Skip 、Application の意味関数と、アドバイスの意味関数 $\mathcal{A}$ を示す。

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{skip}\, e]\!]\, \rho\, \kappa\, \kappa_s = \mathcal{E}[\![e]\!]\, \rho\, \kappa_s\, \kappa_s \\
&\mathcal{E}[\![e_0\, e_1]\!]\, \rho\, \kappa\, \kappa_s = \\
&\quad \mathcal{E}[\![e_0]\!]\, \rho\, (\lambda f.\, \mathcal{E}[\![e_1]\!]\, \rho \\
&\qquad (\mathcal{A}[\![a]\!]\, \mathsf{call}\, (\lambda v'.\, f\, (\mathcal{A}[\![a]\!]\, \mathsf{return}\, \kappa\, \underline{\kappa})\, v')\, \underline{\kappa})\, \kappa_s)\, \kappa_s
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{A}[\![\texttt{advice}(x_a) : p \rightarrow e_a]\!]\, \theta\, \kappa\, \kappa_s = \\
&\quad \textbf{if}\ \mathcal{P}[\![p]\!]\, \theta\ \textbf{then}\ (\lambda v.\, \mathcal{E}[\![e_a]\!]\, [v/x_a]\, \kappa\, \kappa_s)\ \textbf{else}\ \kappa
\end{aligned}
$$

下線のある 2 つの継続は、いずれもスキップ先の継続として意味関数 $\mathcal{A}$ に渡されている。call ジョインポイントのアドバイス中で skip が評価されると、関数呼び出しと return ジョインポイントはスキップされその関数適用の結果を返し終える時点から実行が再開することになる。return ジョインポイントのスキップ先は return ジョインポイントの直後であるため、skip してもしなくても同じように再開される。これは $\mathcal{A}[\![a]\!]$ return に渡す 2 つ継続が共に κ になっていることからわかる。

このようにアドバイス機構と skip がからみあう場合も、どの継続が渡されているかだけで明確に表現できている。

6 関連研究

AspectJ [5]、AspectWerkz [9]、JBoss AOP [2]、AspectC++ [8]、AspectC# [7]、AspectR [1]、AspectS [4] など、ポイントカット・アドバイス機構を持つ AOP 言語処理系のほとんどは区間モデルに基づいている。このため、2 節で説明したような事例に対応できない。

また、AOP 言語を定式化する研究もいくつか行われている。Wand ら [11] は cflow のようにジョインポイントの実行時の情報を必要とするポイントカットを扱える AOP 言語に表示的意味論を与えた。この定式化はアドバイスの種類ごとに異なる意味を用意しているため、定式化は巨大かつ複雑になっている。Walker ら [10] は表面言語の変換先として、ラベルとアドバイスを持つ関数型言語を定義し操作的意味論を与えた。この言語は他の定式化に比べ点モデルに近く、我々の研究と相似する点が多い。しかし、実際にユーザが記述する表面言語では区間モデルを採用している点が我々の研究と異なる。

7 結論

本研究では、メソッド呼び出しなどの前後の時点をそれぞれ異なる種類のジョインポイントとすることで、ジョインポイントを時間上の一点とするアスペクト指向言語の実行モデルを提案した。このモデルでは、`before`、`after` で表現していたタイミングの指示もポイントカットとして記述することができ、アスペクト以外の変更に従来より柔軟に対応できる。

さらに本研究は、提案した実行モデルに継続渡し形式を用いた表示的意味論による定式化を与えた。この定式化では、「関数から帰ってくる」時点を継続の適用と見ることで、「関数を呼び出す」時点と「関数から帰ってくる」時点を λ-項の適用として統一的に扱う。この定式化では他の言語機能を組み込みやすく、複数の言語機能がからみあう場合もシンプルかつ明確に表現できることを示した。このため AOP 研究者が新たな言語機能を考える土台となることが期待できる。

8 今後の展望

今後の展望には大きく 2 つの方向がある。1 つは他の機能を定式化に組み込むこと、もう 1 つは点モデルを採用する AOP 言語を実装することである。

AspectJ には `cflow` や `if` など、マッチするかどうかを判断するのに実行時の情報を必要とするポイントカットや、メソッドの呼び出しではなく実行に相当する execution ジョインポイントが存在する。これらが点モデルの中でどのように実現されるかを考える。複数回の `proceed` を定式化に組み込むことを考える。これを実現するには式の意味関数 $\mathcal{E}$ が継続ではなく関数を受け取る必要があるため、継続渡し形式を用いた表示的意味論では難しい。これは部分継続 [3] の概念を用いることで解決するだろう。

点モデルを採用する AOP 言語を実装するためには、いくつかの最適化が必要になるだろう。例えば、定式化では継続を η-展開しているため naïve な実装では末尾呼び出し除去ができない。静的にアドバイスが一切実行されないことがわかるジョインポイントでは末尾呼び出し除去を行うような最適化が考えられる。また、効率的なコンパイルのためにプログラムを全体ではなく部分的に継続渡し形式に変換するような最適化も考えられる。

謝辞 本研究について数々の助言をくださった浅井健一氏 (お茶の水女子大) と Principles of Programming Languages グループ (東大) のメンバーに感謝する。

参考文献

[1] A. Bryant, and R. Feldt. AspectR, http://aspectr.sourceforge.net/

[2] B. Burke, A. Chau, M. Fleury, A. Brock, A. Godwin, and H. Gliebe, JBoss Aspect Oriented Programming, The JBoss Group, http://www.jboss.org/developers/projects/jboss/aop, 2003

[3] O. Danvy and A. Filinski. Abstracting control. In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pages 151–160, Nice, France, June 1990.

[4] R. Hirschfeld, AspectS – Aspect-Oriented Programming with Squeak. in Proceedings of NODe 2002, LNCS 2591, pages 216-232, Springer-Verlag, 2003

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18–22 June 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220–242, 1997.

[7] H. Kim. AspectC#: An AOSD implementation for C#. MSc. Thesis, Comp.Sci, Trinity College, Dublin, Dublin, 2002.

[8] O. Spinczyk, D. Lohmann, and M. Urban, AspectC++: an AOP Extension for C++. in Software Developer's Journal, pages 68-76, 05/2005.

[9] A. Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address it? In AOSD 2004, Dynamic AOP WorkShop, March 2004.

[10] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In International Conference on Functional Programming, 2003.

[11] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. ACM TOPLAS. 2003.

A 言語の完全な文法と意味論

ここで説明する言語は、例外機構のある ML のサブセットに点モデルのアスペクト機構を追加したものである (図 12) 。

サポートするジョインポイントは以下の通りである。

call 関数を呼び出す時点。

return 関数から帰ってくる時点。

throw 関数を例外の発生により脱出する時点。

handler 例外ハンドラが例外の発生を受ける時点。

handler_end 例外ハンドラをすべて実行し終えて脱出する時点。

handler_throw 例外ハンドラを例外の発生により脱出する時点。

本文中での説明と異なる点を列挙する。

- call(x) ポイントカットや return(x) ポイントカットは引数に関数名をとる。call(x) ポイントカットは関数 x を呼び出す call ジョインポイントを指示し、return(x) ポイントカットは return ジョインポイントを指示する。
- 本文中では例外の発生により関数を脱出する時点も return ジョインポイントとしていたが、ここでは throw ジョインポイントという異なるジョインポイントを割り当てている。もし本文中のように関数から帰ってくる時点と例外の発生によって脱出する時点を同一に扱いたい場合は return(x) $\|$ throw(x) というポイントカットを記述すればよい。
- try 文に、例外ハンドラに識別子 x が追加されている。これは、handler(x) ポイントカットの引数としてどの例外ハンドラを指示しているかを指示できるようにするためである。AspectJ では、識別子ではなく例外オブジェクトのクラスにより例外ハンドラを指示するが、本論文の定式化ではオブジェクト指向の機能は扱っていないため、このように識別子を利用している。
- アドバイスは本文中では advice(x):$p \to e$ であったが、ここでは $x \to e$ を関数としてくくりだして advice:$p \to f$ としている。
- 本文中ではアドバイスは常に 1 つ定義されていたが、ここでは複数定義できる。

図 13 は意味論で使用する意味代数である。Ide は識別子、Int は整数、$Bool$ は真偽値、Fun は関数、Ans は結果のドメインである。

この意味論では継続が 3 つ (通常の継続、現在の例外ハンドラの継続、現在のスキップ先の継続) 存在するため、関数 Fun は引数として値の他に継続を 3 つ受け取る。値を受け取り結果を返すのは継続であるため、関数は継続を 3 つ受け取り継続を返すものと考えられる。

Jp のバリアント call や return の引数は $(Ide + *)$ となっている。これは、ジョインポイントに関数名が存在しない、または構文的は判断できない場合のためである。例えば (fun x -> x) 3 のような匿名関数の適用では、call ジョインポイントに関数名が存在しない。このようなジョインポイントは call($*$) として表現する。

図 14 は意味関数の型、図 15 は意味関数の定義である。ただし **extend** : $(Ide \times Val) \times Env \to Env$ は識別子のリストと値のリストを受け取り環境を拡張する関数である。

関数の意味関数 $\mathcal{F}$ は文法上の λ-抽象を意味論に写す。ポイントカットの意味関数 $\mathcal{P}$ はポイントカットとジョインポイントとをマッチした結果を真偽値として返す。アドバイスの意味関数 $\mathcal{W}$ はポイントカットがマッチした場合はアドバイスの関数を適用する継続を返し、マッチしなかった場合は現在の継続をそのまま返す。

$$
\begin{array}{llll}
\text{(Expression)} & e & = & n \qquad \text{(Constant)} \\
 & & \mid & x \qquad \text{(Identifier)} \\
 & & \mid & f \qquad \text{(Function)} \\
 & & \mid & e\ e \qquad \text{(Application)} \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \qquad \text{(If)} \\
 & & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \qquad \text{(Let)} \\
 & & \mid & \texttt{fix}\ (x = f)\ *\ \texttt{in}\ e \qquad \text{(Fix)} \\
 & & \mid & \texttt{try}\ e\ \texttt{with}\ x = f \qquad \text{(Try)} \\
 & & \mid & \texttt{raise}\ e \qquad \text{(Raise)} \\
 & & \mid & \texttt{skip}\ e \qquad \text{(Skip)} \\
\text{(Function)} & f & = & \texttt{fun}\ x \to e \\
\text{(Pointcut)} & p & = & \texttt{call}(x) \mid \texttt{return}(x) \mid \texttt{throw}(x) \mid \texttt{get}(x) \\
 & & \mid & \texttt{handler}(x) \mid \texttt{handler\_end}(x) \mid \texttt{handler\_throw}(x) \\
 & & \mid & p_0 \parallel p_1 \\
\text{(Advice)} & a & = & \texttt{advice:}\ p = f \\
\text{(Advices)} & A & = & \cdot \mid a;\ A \\
\text{(Program)} & r & = & e;\ A
\end{array}
$$

図 12: 言語の文法

$$
\begin{aligned}
& v \in \mathit{Val} = \mathit{Int} + \mathit{Bool} + \mathit{Fun} \\
& \rho \in \mathit{Env} = \mathit{Ide} \to \mathit{Val} \\
& \kappa \in \mathit{Ctn} = \mathit{Val} \to \mathit{Ans} \\
& f \in \mathit{Fun} = \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Val} \to \mathit{Ans} = \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \\
& \theta \in \mathit{Jp} = \mathsf{call}(\mathit{Ide} + *) + \mathsf{return}(\mathit{Ide} + *) + \mathsf{throw}(\mathit{Ide}) + \mathsf{get}(\mathit{Ide}) \\
& \qquad\quad + \mathsf{handler}(\mathit{Ide}) + \mathsf{handler\_end}(\mathit{Ide}) + \mathsf{handler\_throw}(\mathit{Ide})
\end{aligned}
$$

図 13: 意味論で使用する意味代数

$$
\begin{aligned}
& \mathcal{E} : \text{Advices} \to \text{Expression} \to \mathit{Env} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Val} \to \mathit{Ans} \\
& \mathcal{F} : \text{Advices} \to \text{Function} \to \mathit{Env} \to \mathit{Val} \\
& \mathcal{W} : \text{Advices} \to \text{Advices} \to \mathit{Jp} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn} \\
& \mathcal{P} : \text{Pointcut} \to \mathit{Jp} \to \mathit{Bool} \\
& \mathcal{R} : \text{Program} \to \mathit{Ans}
\end{aligned}
$$

図 14: 意味関数の型

(Expression)

$$
\begin{array}{rcl}
\mathcal{E}_A[\![c]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \kappa\,c \\
\mathcal{E}_A[\![x]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{W}[\![A]\!]\,\mathsf{get}(x)\,\kappa\,\kappa_s\,\kappa_h\,(\rho(x)) \\
\mathcal{E}_A[\![f]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \kappa\,(\mathcal{F}_A[\![f]\!]\,\rho) \\
\mathcal{E}_A[\![e_0\ e_1]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e_0]\!]\,\rho\,(\lambda v_0.\,\mathcal{E}_A[\![e_1]\!]\,\rho\,(\lambda v_1. \\
 & & \quad \mathbf{let}\ l = (\mathbf{match}\ e_0\ \mathbf{with}\ x \to x\ [\!]\ \_ \to *)\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \kappa' = \mathcal{W}[\![A]\!]\,\mathsf{return}(l)\,\kappa\,\kappa\,\kappa_h\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \kappa'_h = \mathcal{W}[\![A]\!]\,\mathsf{throw}(l)\,\kappa_h\,\kappa_h\,\kappa_h\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \mathit{Fun}(f) = v_0\ \mathbf{in} \\
 & & \quad \mathcal{W}[\![A]\!]\,\mathsf{call}(l)\,(\lambda v.\ f\,\kappa'\,\kappa'\,\kappa'_h\,v)\,\kappa'\,\kappa'_h\,v_1)) \\
\mathcal{E}_A[\![\mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e_0]\!]\,\rho\,(\lambda v.\ \mathbf{let}\ \mathit{Bool}(b) = v\ \mathbf{in}\ b \to \mathcal{E}_A[\![e_1]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h \\
 & & \qquad\qquad [\!]\ \mathcal{E}_A[\![e_2]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h) \\
\mathcal{E}_A[\![\mathtt{let}\ x = e_0\ \mathtt{in}\ e_1]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e_0]\!]\,\rho\,(\lambda v.\,\mathcal{E}_A[\![e_1]\!]\,([x \to v]\rho)\,\kappa\,\kappa_s\,\kappa_h)\,\kappa_s\,\kappa_h \\
\mathcal{E}_A[\![\mathtt{fix}\ (x = f)\ *\ \mathtt{in}\ e]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e]\!]\,(\mathbf{fix}\,(\lambda\rho'. \\
 & & \quad \mathbf{extend}(\mathbf{map}(\lambda[\![(x = f)]\!].\,(x, \mathcal{F}_A[\![f]\!]\,\rho))[\![(x = f)*]\!], \rho) \\
 & & \quad))\,\kappa\,\kappa_s\,\kappa_h \\
\mathcal{E}_A[\![\mathtt{try}\ e\ \mathtt{with}\ x = f]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e]\!]\,\rho\,\kappa\,\kappa_s(\lambda v. \\
 & & \quad \mathbf{let}\ \kappa' = \mathcal{W}[\![A]\!]\,\mathsf{handler\_end}(x)\,\kappa\,\kappa\,\kappa_h\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \kappa'_h = \mathcal{W}[\![A]\!]\,\mathsf{handler\_throw}(x)\,\kappa_h\,\kappa_h\,\kappa_h\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \mathit{Fun}(f') = \mathcal{F}_A[\![f]\!]\,\rho\ \mathbf{in} \\
 & & \quad \mathcal{W}[\![A]\!]\,\mathsf{handler}(x)\,(\lambda v.\ f\,\kappa'\,\kappa'\,\kappa'_h\,v)\,\kappa'\,\kappa'_h\,v) \\
\mathcal{E}_A[\![\mathtt{raise}\ e]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e]\!]\,\rho\,\kappa_h\,\kappa_s\,\kappa_h \\
\mathcal{E}_A[\![\mathtt{skip}\ e]\!]\,\rho\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{E}_A[\![e]\!]\,\rho\,\kappa_s\,\kappa_s\,\kappa_h
\end{array}
$$

(Function)

$$
\mathcal{F}_A[\![\mathtt{fun}\ x \to e]\!]\,\rho = \mathit{inFun}(\lambda\kappa\,\kappa_s\,\kappa_h\,v.\,\mathcal{E}_A[\![e]\!]\,([x \to v]\rho)\,\kappa\,\kappa_s\,\kappa_h)
$$

(Weaver)

$$
\begin{array}{rcl}
\mathcal{W}_A[\![\mathtt{advice}\ p = f;\ A']\!]\,\theta\,\kappa\,\kappa_s\,\kappa_h & = & \mathcal{P}[\![p]\!]\,\theta \to \mathbf{let}\ \mathit{Fun}(f') = \mathcal{F}_A[\![f]\!]\ \mathbf{in} \\
 & & \qquad (\lambda v.\ f'\ (\mathcal{W}[\![A']\!]\,\theta\,\kappa\,\kappa_s\,\kappa_h)\,\kappa_s\,\kappa_h\,v) \\
 & & \quad [\!]\ \mathcal{W}[\![A']\!]\,\theta\,\kappa\,\kappa_s\,\kappa_h \\
\mathcal{W}_A[\![\cdot]\!]\,\theta\,\kappa\,\kappa_s\,\kappa_h & = & \kappa
\end{array}
$$

(Pointcut)

$$
\begin{array}{rcl}
\mathcal{P}[\![\mathtt{call}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{call}(x) \\
\mathcal{P}[\![\mathtt{return}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{return}(x) \\
\mathcal{P}[\![\mathtt{throw}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{throw}(x) \\
\mathcal{P}[\![\mathtt{get}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{get}(x) \\
\mathcal{P}[\![\mathtt{handler}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{handler}(x) \\
\mathcal{P}[\![\mathtt{handler\_end}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{handler\_end}(x) \\
\mathcal{P}[\![\mathtt{handler\_throw}(x)]\!]\,\theta & = & \theta\ =\ \mathsf{handler\_throw}(x) \\
\mathcal{P}[\![p_0 \parallel p_1]\!]\,\theta & = & (\mathcal{P}[\![p_0]\!]\,\theta)\ \vee\ (\mathcal{P}[\![p_1]\!]\,\theta)
\end{array}
$$

(Program)

$$
\mathcal{R}[\![e;\ A]\!] = \mathcal{E}_A[\![e]\!]\,[\!]\,(\lambda v.\,v)\,(\lambda v.\,v)\,(\lambda v.\,v)
$$

図 15: 意味関数の定義