その言語、定義されてますか

びしょ〜じょ

Dec 15, 2019

1. はじめに

こんにちは、びしょ〜じょです。この記事は言語実装 Advent Calendar 2019 の 15 日目の記事です。

皆さん、言語実装してますか。ある人は C コンパイラ、またある人は JS のエンジン……。そしてある人は自作言語の実装。しかしその言語は実装が仕様になってませんか？“実装が仕様” の危うさは今更言うまでもないですが、プログラム言語においては、型システムなどを考えるためには仕様がカチッと定まっていないとナンセンスですね。

本記事では、以前作った言語 λ_{eff} を題材に、実装前夜として言語の定義について簡単に述べます。言語**実装**アドベントカレンダーなので恐縮ですが、ほとんど仕様の話になります。

1.1. 言語実装手順、または本記事の進み方

言語が何に先立つかは議論の余地がありますが、筆者のおこなう言語実装プロセスは次のようになります。

1. まずはじめに、作りたい言語をイメージする。general purpose な言語を実装しようと思ったことがないので、とりあえず一番実装したい機能を考え、その機能を持つ言語 (だいたいいつも ML(ラムダ計算 + `let`くらい)) をイメージする。
2. 次に構文と意味論、型システムなどを考える。この辺は前のフェーズでイメージした言語をコンクリートにしていく。
3. 最後に実装を与える。パーザの実装が面倒くさかったり意味論のバグが発覚したりすると、1 つ前のフェーズに戻る。このイテレーションを複数回繰り返すことがある。
   フロントエンドの実装なんて面倒だしええわになるとパーザを実装せず AST をそのまま渡しちゃう。

特に筆者が定義する程度の小さな言語なら、ちゃんと 2.でカチッと定義できていれば 3.から戻ってまた実装してのイテレーションは回さなくて良いのですが、筆者程度に適当な人間だと雑に定義して雑に実装して NG な部分を直すのを繰り返すことになります。

本記事は、実際に上記のプロセスに沿って実装していった λ_{eff} を見ていきます。

2. 作りたい言語をイメージしよう

では早速作りたい言語を考えます。

とにかく代数的効果をもつ言語を実装したかったのでそういう言語を考えました[*1]。とりあえずハンドラとエフェクトの発生の構文を考えた。プログラムの最大単位が文のリストなのはサクッと言語作るには微妙だな〜ということでエフェクトの定義も式として定義する。ベースとなる言語はとりあえずラムダ計算 + `let`、つまり ML でいいでしょう。

具体的な構文は次の手順で考えますが、このフェーズでも多少は構文のことを念頭に置いてイメージする必要があります。遅延評価で代数的効果はワカランので正格評価で良いか〜などもふんわりと決めています。

Listing 1　こんな感じに……

```
let double = inst() in
```

[*1] この言語に関してはこれは嘘で、ほぼ完全に Eff 言語と同じものを考えた。

```
with 2 + perform double 10 handle
| double x k -> k (x * x)
(* ==> (2 + [ ])(10*10) = 102 *)
```

このフェーズはふわっとしてるので適当にイマジンしていただいて結構ですが、最初から大きな言語を作ろうとしてここで巨大構想を練るとおそらく次の段階で頓挫するので、ベースになる言語 +1、2 機能くらいで考えておきましょう。

べつにいつ決めても良いし決めなくてもいいんですが、とりあえず今作ろうとしている言語の名前を決めておきましょう。うーん、ラムダ計算に代数的効果だから “λ_{eff}” で!w [*2]

3. イメージの具体化 (1): 構文

統語論が意味論に先立つのか、意味論が統語論に先立つのかは議論の余地がありますが、多くの論文がそうであるように、議論したいプログラム言語の形をハッキリさせるために構文から定義します。はいこちら。

$$
\begin{array}{rcl}
x & \in & \mathit{Variables} \\
\mathit{eff} & \in & \mathit{Effects} \\
\\
v & ::= & x \mid h \mid \lambda x.e \mid \mathit{eff} \\
e & ::= & v \mid e\ e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \\
 & \mid & \mathtt{inst}\ () \mid \mathtt{with}\ v\ \mathtt{handle}\ e \\
 & \mid & \mathtt{perform}\ e\ e \\
h & ::= & \mathtt{handler}\ v\ (\mathtt{val}\ x \to e)\ ((x, k) \to e)
\end{array}
$$

図 1　λ_{eff} の構文

BNF っぽい定義方法ですね。筆者独自のものではなく、多少の違いはありますが、多くの論文などで用いられる定義方法です。

“$x \in \mathit{Variables}$” は、“x は変数集合 $\mathit{Variables}$ に含まれる変数名を表すメタ変数” と読みます。**メタ変数**って何やねんですが、これは今定義せんとしている λ_{eff} の変数ではなく、λ_{eff} の要素を指すために付けた名前です。なので “メタ”。記述 (実装) 言語をメタ言語と呼ぶのと同じですね。ちなみに記述する対象は対象言語です。だいたいメタ言語の要素と対象言語の要素は書体で書き分けられ、前者は *italic*、後者は `typewriter` がち。

話が長くなりましたが、3 行目から続く実際の構文定義を見ましょう。主に 3 つの要素 v、e、h で構成されていることが分かるでしょうか。そしてこれらは相互再帰的な構造になっています。だいたい e と v は頻出のメタ変数で、それぞれ *expression* と *value* を指します。そして、明示されてませんが、トップレベルは expression です。h は *handler* の略です。Listing 1には整数や二項演算が出てきますが、自明そうなので omit しときます。

構文をちゃんと定義しておくことで、(実装もちゃんとしてれば) いちいちパーザの実装を見なくてもどういった文字列をプログラムとして受理してくれるのかが分かります。

[*2] effectful computation を扱う論文の core calculus の名前とだいたいかぶってる

4. イメージの具体化 (2): 意味論

皆さんお待ちかねの意味論を定義します。意味論には色々なスタイルがありますが、目的や好みに応じて適当に選んでください。一番メジャーなパターンが**操作的意味論**だと思います。対象言語の項自体や、言語の実行モデルとしての抽象機械の状態遷移で表すやつです (雑)。λ_{eff} は代数的効果を持つ言語なので、コントロールフローをシュッとやる定義にする必要があります。ともすれば CEK **マシン**あたりが定石でしょうか。コントロールフローをガンガン扱うような機能 (例外とか、if やパターンマッチも) は評価コンテキストをガッと取ってワーッとやると楽に定義できます。

意味論の定義のための抽象機械で使う構文を追加します (図 2)。

$$
\begin{array}{rcl}
F & ::= & e\,\square \mid \square\,v \mid \texttt{let}\ x = \square\ \texttt{in}\ e \\
 & \mid & \texttt{with}\ v\ \texttt{handle}\ \square \mid \texttt{perform}\ \square\ e \mid \texttt{perform}\ v\ \square \\
s & ::= & [] \mid F :: s
\end{array}
$$

図 2　構文 (続)

F は "$Frame$" を表します。スタックフレームの 1 要素ですね。穴 $\square$ があいており、$F\,[e]$ と書くことで穴に e が放り込まれる、と考えてください。s はスタックです。

では早速意味論を……といきたいところですが、補助関数を定義します (図 3)。

$$
\begin{aligned}
\mathit{flatfn}\ [] &= \lambda x.x \\
\mathit{flatfn}\ (F :: s) &= \lambda x.\mathit{flatfn}\ s\ (F\,[x])
\end{aligned}
$$

図 3　補助関数 flatfn

スタックフレームから継続を生成したいのですが、その処理を再帰的に定義するために補助関数として切り出します。

補助関数も揃ったので、いよいよ本丸にいきます。$\langle e;\ s;\ es \rangle$ という三組 (es はスタックを表すメタ変数) で表される抽象機械の状態遷移による操作的意味論を示します (図 4)。λ_{eff} を作っていたときは知恵がやや足りてなかったので、謎の抽象機械の状態遷移による操作的意味論を与えています。`let` 式などの操作が無いのは、`let` 式がフレームによって表され、フレームの操作が Push 規則で定義されてるためです。そういえば思い出したけどスコープあたりがバグってます。しっかり考えて作られた CEK マシンなどを使いましょう。

ここに書かれてないような状態になるとランタイムエラーになります。未定義動作というのは、処理系実装者が勝手に忖度した場合と、自明なので各位適当に実装してくれという言語デザイナの意図を汲んだ場合の 2 通りがあると思います。前者は主に C で、後者は例えば OCaml のタプルの評価順序などが挙げられそうです。

なにはともあれ、意味論を定義することで、実装のバグなのか仕様のバグなのかがはっきりします。この場合、変数のスコープがおかしいのは仕様のバグであることが分かります。

4.1. 型システムとの連携

型は素人なのですみませんが subsection で簡潔に述べる程度にとどめておきます。

意味論の定義は型システムを考える場合も重要です。型システムがやりたいのは、変な計算をするプログラムを事前にはじくことです。変な計算とは? それは評価が進むとランタイムエラーになるような計算です。計算がエラーになるということを定義するために、意味論は必要になります。逆に、型が付く項はエラーにならないということも言えると嬉しそうです。

5. 実装

$$\langle F\,[e]\,;\,s;\,es\rangle \mapsto \langle e;\,F :: s;\,es\rangle \qquad \text{(Push)}$$

$$\langle v;\,F :: s;\,es\rangle \mapsto \langle F\,[v]\,;\,s;\,es\rangle \qquad \text{(Pop)}$$

$$\langle \lambda x.e;\,(\square\ v) :: s;\,es\rangle \mapsto \langle e\,[x = v]\,;\,s;\,es\rangle \qquad \text{(Apply)}$$

$$\langle \texttt{inst}\ ()\,;\,s;\,es\rangle \mapsto \langle \mathit{eff};\,s;\,es\rangle \qquad \text{(Instanciate)}$$

$$\langle \texttt{perform}\ \mathit{eff}\ v;\,F :: s;\,es\rangle \mapsto \langle \texttt{perform}\ \mathit{eff}\ v;\,s;\,F :: es\rangle \qquad \text{(Rethrow)}$$

$$\langle \texttt{perform}\ \mathit{eff}\ v;\,F :: s;\,es\rangle \mapsto \langle e_{\mathit{eff}}\ [x = v, k = \mathit{flatfn}\ es]\,;\,F :: s;\,[]\rangle$$
$$\text{where}\quad F = \texttt{with}\ h\ \texttt{handle}\ \square \qquad \text{(Handle}_{\text{eff}}\text{)}$$
$$h = \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((x, k) \to e_{\mathit{eff}})$$

$$\langle v;\,F :: s;\,es\rangle \mapsto \langle e_v[x = v];\,s;\,es\rangle$$
$$\text{where}\quad F = \texttt{with}\ h\ \texttt{handle}\ \square \qquad \text{(Handle}_{\text{v}}\text{)}$$
$$h = \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((x, k) \to e_{\mathit{eff}})$$

図 4　λ_{eff} の意味論

あとはやるだけ。……といいたいところですが、コンパイラを実装したろ〜という場合はまだ定義すべきものがあります。それは今定義した対象言語から、コンパイル先の言語への変換です。これもちゃんと定義することで、コンパイラの実装がバグってるのか、変換がバグってるのかが明らかとなり、デバッグが比較的容易になります。

とりあえず λ_{eff} はインタプリタを実装しました。小ステップ意味論で定義しても、インタプリタの実装はだいたい大ステップ意味論での実装になりがちです。なんといってもそのほうが簡単なので。しかし今回は素直に定義に沿って実装したい気持ちがなんとなくあったので、小ステップ意味論で実装していきました。

5.1. 最適化

ウッ頭が……

6. おわりに

TaPL などで言語の定義に用いられる記号とかそういったものの解説とかをしようと思ったのですが、なんかとりとめのない謎の記事になりました。そのうえ時間がギリギリアウトです (この文は 23 時 59 分に書かれた)。