The definition of Standard Pancake

(First edition)

The definition of Standard Pancake[1]

Ben Nott
b.nott@student.unsw.edu.au
UNSW, Sydney

July 2023

[1]Where a "Standard Pancake" is understood not to include toppings or syrup.

Preface

Pancake is an imperative programming language designed for the development of formally verified device drivers. The language is built upon the CakeML project, which provides a verified implementation of a substantial subset of Standard ML.

Chapter 1

Introduction

1.1 The language

Pancake is an imperative programming language intended for the development of formally verified device drivers.

1.2 File naming

The names of Pancake source files *must* be suffixed with one of the following filename extensions:

1. `.p`
2. .🥞[1];

It is recommended to use the emoji filename extension (option 2) where operating system support is available.

[1] Where the symbol is the "Pancakes" emoji represented by the UTF-8 codepoint `1F95E`.

Chapter 2

Syntax

2.1 Concrete syntax

The Pancake concrete syntax is defined across the `panLexer` and `panPEG` theories. Here, the lexical analysis and PEG parsing constrain the initial shape and overall form of all Pancake programs.

The Pancake syntax is by design similar to that of the C programming language. The language of Pancake programs $\mathcal{L}_P$ is defined by the following grammar.

$\langle P\rangle$ ::= ‘`fun`’ `IDENT` ‘`(`’ $\langle ParamList\rangle$ ‘`)`’ ‘`{`’ $\langle Prog\rangle$ ‘`}`’
  | $\langle P\rangle$ $\langle P\rangle$

$\langle ParamList\rangle$ ::= $\langle Shape\rangle$ `IDENT`
  | $\langle Shape\rangle$ `IDENT` ‘`,`’ $\langle ParamList\rangle$

$\langle Prog\rangle$ ::= $\langle Block\rangle$
  | $\langle Stmt\rangle$ ‘`;`’

$\langle Block\rangle$ ::= $\langle Dec\rangle$ | $\langle If\rangle$ | $\langle While\rangle$

$\langle Stmt\rangle$ ::= ‘`skip`’
  | $\langle Call\rangle$
  | $\langle Assign\rangle$
  | $\langle Store\rangle$
  | $\langle StoreByte\rangle$
  | ‘`break`’
  | ‘`continue`’

$$
\begin{array}{ll}
| & \langle \mathit{ExtCall} \rangle \\
| & \langle \mathit{Raise} \rangle \\
| & \langle \mathit{Return} \rangle \\
| & \text{`}\texttt{tick}\text{'} \\
| & \text{`}\texttt{\{}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{\}}\text{'}
\end{array}
$$

$$\langle \mathit{Dec} \rangle ::= \text{`}\texttt{var}\text{'}\ \texttt{IDENT}\ \text{`}\texttt{=}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{;}\text{'}\ \langle \mathit{Prog} \rangle$$

$$\langle \mathit{Assign} \rangle ::= \texttt{IDENT}\ \text{`}\texttt{=}\text{'}\ \langle \mathit{Exp} \rangle$$

$$\langle \mathit{Store} \rangle ::= \text{`}\texttt{str}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \mathit{Exp} \rangle$$

$$\langle \mathit{StoreByte} \rangle ::= \text{`}\texttt{strb}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \mathit{Exp} \rangle$$

$$
\begin{array}{ll}
\langle \mathit{If} \rangle ::= & \text{`}\texttt{if}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{\{}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{\}}\text{'} \\
| & \text{`}\texttt{if}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{\{}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{\}}\text{'}\ \text{`}\texttt{else}\text{'}\ \text{`}\texttt{\{}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{\}}\text{'}
\end{array}
$$

$$\langle \mathit{While} \rangle ::= \text{`}\texttt{while}\text{'}\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{\{}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{\}}\text{'}$$

$$
\begin{array}{ll}
\langle \mathit{Call} \rangle ::= & \langle \mathit{Ret} \rangle\ \langle \mathit{Exp} \rangle\ \text{`}\texttt{(}\text{'}\ \langle \mathit{CallArgList} \rangle\ \text{`}\texttt{)}\text{'} \\
| & \langle \mathit{Exp} \rangle\ \text{`}\texttt{(}\text{'}\ \langle \mathit{CallArgList} \rangle\ \text{`}\texttt{)}\text{'}
\end{array}
$$

$$\langle \mathit{CallArgList} \rangle ::= \langle \mathit{ArgList} \rangle \mid \epsilon$$

$$
\begin{array}{ll}
\langle \mathit{Ret} \rangle ::= & \texttt{IDENT}\ \text{`}\texttt{=}\text{'}\ \langle \mathit{Handle} \rangle \\
| & \texttt{IDENT}\ \text{`}\texttt{=}\text{'}
\end{array}
$$

$$\langle \mathit{Handle} \rangle ::= \text{`}\texttt{with}\text{'}\ \texttt{IDENT}\ \text{`}\texttt{in}\text{'}\ \texttt{IDENT}\ \text{`}\texttt{=>}\text{'}\ \langle \mathit{Prog} \rangle\ \text{`}\texttt{handle}\text{'}$$

$$\langle \mathit{Raise} \rangle ::= \text{`}\texttt{raise}\text{'}\ \texttt{IDENT}\ \langle \mathit{Exp} \rangle$$

$$\langle \mathit{Return} \rangle ::= \text{`}\texttt{return}\text{'}\ \langle \mathit{Exp} \rangle$$

$$
\begin{array}{ll}
\langle \mathit{ArgList} \rangle ::= & \langle \mathit{Exp} \rangle \\
| & \langle \mathit{Exp} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \mathit{ArgList} \rangle
\end{array}
$$

$$
\begin{array}{ll}
\langle \mathit{Exp} \rangle ::= & \langle \mathit{EXor} \rangle \\
| & \langle \mathit{EXor} \rangle\ \text{`}\texttt{|}\text{'}\ \langle \mathit{Exp} \rangle
\end{array}
$$

$$
\begin{array}{ll}
\langle \mathit{EXor} \rangle ::= & \langle \mathit{EAnd} \rangle \\
| & \langle \mathit{EAnd} \rangle\ \text{`}\texttt{\^{}}\text{'}\ \langle \mathit{EXor} \rangle
\end{array}
$$

$$
\begin{array}{ll}
\langle \mathit{EAnd} \rangle ::= & \langle \mathit{EEq} \rangle \\
| & \langle \mathit{EEq} \rangle\ \text{`}\texttt{\&}\text{'}\ \langle \mathit{EAnd} \rangle\ \langle \mathit{EAnd} \rangle
\end{array}
$$

$\langle \mathit{EEq} \rangle$:;= $\langle \mathit{ECmp} \rangle$
| $\langle \mathit{ECmp} \rangle$ $\langle \mathit{EqOps} \rangle$ $\langle \mathit{ECmp} \rangle$

$\langle \mathit{ECmp} \rangle$ ::= $\langle \mathit{EShift} \rangle$
| $\langle \mathit{EShift} \rangle$ $\langle \mathit{CmpOps} \rangle$ $\langle \mathit{EShift} \rangle$

$\langle \mathit{EShift} \rangle$ ::= $\langle \mathit{EAdd} \rangle$
| $\langle \mathit{EAdd} \rangle$ $\langle \mathit{ShiftOps} \rangle$ `NUMBER` $\langle \mathit{EShift} \rangle$

$\langle \mathit{EAdd} \rangle$ ::= $\langle \mathit{EMul} \rangle$
| $\langle \mathit{EMul} \rangle$ $\langle \mathit{AddOps} \rangle$ $\langle \mathit{EMul} \rangle$ $\langle \mathit{EAdd} \rangle$

$\langle \mathit{EMul} \rangle$ ::= $\langle \mathit{EBase} \rangle$
| $\langle \mathit{EBase} \rangle$ $\langle \mathit{MulOps} \rangle$ $\langle \mathit{EBase} \rangle$ $\langle \mathit{EMul} \rangle$

$\langle \mathit{EBase} \rangle$ ::= ‘`(`’ $\langle \mathit{Exp} \rangle$ ‘`)`’
| ‘`true`’
| ‘`false`’
| `INTEGER`
| `IDENT`
| $\langle \mathit{Label} \rangle$
| $\langle \mathit{Struct} \rangle$
| $\langle \mathit{Load} \rangle$
| $\langle \mathit{LoadByte} \rangle$
| ‘`@base`’

$\langle \mathit{Label} \rangle$ ::= ‘`!`’ `IDENT`

$\langle \mathit{Struct} \rangle$ ::= ‘`<`’ $\langle \mathit{ArgList} \rangle$ ‘`>`’

$\langle \mathit{Load} \rangle$ ::= ‘`lds`’ $\langle \mathit{Shape} \rangle$ $\langle \mathit{Exp} \rangle$

$\langle \mathit{LoadByte} \rangle$ ::= ‘`ldb`’ $\langle \mathit{Exp} \rangle$

$\langle \mathit{Shape} \rangle$ ::= `INT`
| ‘`{`’ $\langle \mathit{ShapeComb} \rangle$ ‘`}`’

$\langle \mathit{ShapeComb} \rangle$ ::= $\langle \mathit{Shape} \rangle$
| $\langle \mathit{Shape} \rangle$ ‘`,`’ $\langle \mathit{ShapeComb} \rangle$

$\langle \mathit{EqOps} \rangle$ ::= ‘`==`’
| ‘`<>`’

$\langle \mathit{CmpOps} \rangle$ ::= ‘<’
  | ‘>=’
  | ‘>’
  | ‘<=’

$\langle \mathit{ShiftOps} \rangle$ ::= ‘<<’
  | ‘>>>’
  | ‘>>’
  | ‘>>’

$\langle \mathit{AddOps} \rangle$ ::= ‘+’
  | ‘-’

$\langle \mathit{MulOps} \rangle$ ::= ‘*’

The following *keywords* are reserved by the language and may not be used by identifiers.

```
skip str strb if else while break continue
raise return tick var in handle lds ldb @base
true false
```

2.2 Abstract syntax

The abstract syntax is broken into two main categories: *expressions* and *program commands* (commands for short). These are the objects over which the semantics for Pancake are defined. The abstract syntax is defined in the theory `panLang`.

Unlike in the concrete syntax, there is no strict requirement for a Pancake program to take any particular initial shape. We can analyse the abstract syntax of any program fragment at the expression or command level.

Chapter 3

Semantics

The semantics of Pancake programs is defined on two levels; an observational level, which concerns the externally visible events of executed programs; and the evaluation level, which concerns the computation of the program text and its outcome.

Two approaches have been taken to defining a semantics for Pancake. The *functional big-step* semantics [1] describes the meaning of Pancake programs by defining the evaluation result of each term in the abstract syntax. The *interaction tree* semantics defines the meaning of Pancake programs as a infinitely-branching tree of possible behaviour where each leaf is the result of evaluation and internal nodes represent observable events.

3.1 Functional big-step semantics

The functional big-step (FBS) semantics is defined in the `panSem` theory. It describes the meaning of Pancake programs by assigning an evaluation result to each term in the abstract syntax.

This approach requires defining a recursive function for each of the types of program terms: program commands and expressions. Consequently, the semantics uses a clock value, decremented at each recursive operation, to preserve well-foundedness.

3.1.1 Program state

Along with the FBS semantics, the shape of program state is defined by the `state` datatype in the `panSem` theory.

Functions defined in the program text are treated as a map

$$\text{code} : \text{string} \to (\text{arg list} \times \text{prog})$$

and are made available through the `code` field.

3.1.2 Foreign function interface

To permit Pancake programs to execute external code, the language allows for use of a foreign function interface (FFI). The Pancake semantics models the FFI in the `ffi` state field; an element of `ffi_state` defined in the `ffi` theory.

An FFI state contains both a state element and an oracle. The oracle serves to represent the remote end of the FFI and the `call_FFI` method utilises this oracle to compute the outcome of calling a foreign function.

Notably, a non-terminal call to a foreign function, results in an updated FFI state, thus advancing the behaviour of the oracle upon the next FFI call.

3.2 Interaction tree (itree) semantics

Being a language intended for device driver development, it follows that a more natural semantics would be one that captures the reactive nature of typical driver operation. Thus a tree-based semantics was developed for Pancake using interaction trees [2].

The ITree semantics are defined in the `panItreeSem` theory.

3.2.1 Recursive event handlers

The ITree semantics is designed around a mutual recursion combinator first described by Xia [2]

$$\text{mrec} : (\alpha \to (\alpha, \beta + \epsilon, \delta)\ \text{itree}) \to \alpha \to (\gamma, \epsilon, \delta)\ \text{itree}.$$

This combinator allows one to corecursively construct an itree from a seed by appling an algorithm; known as the *recursive event handler*. The combinator was designed to leverage the executable features of the itree `Vis` nodes to allow the modelling of mutual recursion by the corecursive unfolding of a seed into a final itree.

The recursive event handler

$$\text{rh} : (\alpha \to (\alpha, \beta + \epsilon, \delta)\ \text{itree})$$

uses the $\beta + \epsilon$ option type for the argument type to `Vis`. When the recursive event handler returns `Vis (INL` v`)` k, the `mrec` operator continues iteration, taking $v : \alpha$ as the seed, appling it to `rh` and binding the result to $k : (\alpha \to (\alpha, \beta + \epsilon, \delta)$ itree. The constructed itree thus takes as its node for the current depth

$$\texttt{rh}\ v \star k.$$

When the handler returns `Vis (INR` v'`)` k', `mrec` inserts a `Vis` node into the tree with the event $v' : \epsilon$ and ktree $k' : (\gamma, \epsilon, \delta)$ itree, by requiring that `mrec` be reinvoked to produce the result of

$$x\ k'$$

for any $x : \gamma$.

The `mrec` combinator terminates production of the tree when `rh` returns `Ret` r, for $r : \delta$, adding this node as the leaf.

3.2.2 Layout

The itree semantics are separated into two main components

1. the top-level semantics, given by the `itree_semantics` definition; and

2. the evaluation semantics, given by the `itree_evaluate` definition.

The top-level semantics, in the same fashion as the FBS semantics, defines the meaning of Pancake programs in terms of a tail call to a single entrypoint, with a state, the captures namely the loaded program text (in the `code` field) and the FFI state (in the `ffi` field).

The evaluation semantics uses the `mrec` combinator to construct a complete itree from the terms of the abstract syntax. It is most closely related to the `evaluate` definition in the FBS semantics.

3.2.3 Evaluation semantics

The evaluation semantics is defined by `itree_evaluate` and utilises the `mrec` combinator to corecursively construct an itree from a Pancake program and initial state.

The `mrec` operator unfolds a seed into a complete itree using a *recursive event handler*. We define the recursive event handler

$$\text{h\_prog} : \text{prog} \times \text{state} \to (\alpha, \beta, \gamma)\ \text{itree}$$

over the program command terms of the abstract syntax so as to achieve the compositionality desired of a denotational semantics.

To allow recursive behaviour to be expressed as the `Vis` nodes required by the `mrec` operator, we must correspond the type of our seed values with the answer type of the itree. Hence, the itrees produced by the `mrec` operator take as their answer type

$$\alpha = \text{prog} \times \text{state}.$$

Because of this, the ktree of each `Vis` node must have the type

$$k : (\text{prog} \times \text{state}) \to (\text{prog} \times \text{state}, \beta, \gamma) \text{ itree}$$

and so we cannot define a ktree mapping the response of FFI calls to an itree, as we might naturally desire. To overcome this limitation, we embed the desired ktree in the event of the constructed itree and later massage the tree into the desired type. Thus we have that for our desired event type ϵ and answer type α'

$$\beta = \epsilon \times (\alpha' \to (\alpha', \epsilon, \gamma) \text{ itree}).$$

Because we propagate state in the seed value to `mrec`, which uses the bind operator to continue itree production, we must have the argument type to our `Ret` node as

$$\gamma = (\eta \text{ result option} \times \text{state}).$$

Massaging final itree shape

It is in the definition `itree_evaluate` where we massage the type of itree produced by `mrec` into our desired type of

$$(\text{ffi\_behaviour}, \text{sem\_vis\_event}, \eta \text{ result option}) \text{ itree}.$$

We do so by taking our `mrec` produced itree and applying it as a seed to an unfold operation which

1. converts the type γ into $\gamma' = \eta$ result option so that `Ret` nodes no longer contain program state; and which
2. converts `Vis` $(\epsilon \times (\alpha' \to (\alpha', \epsilon, \gamma) \text{ itree}))$ $(\alpha \to (\alpha, \beta, \gamma) \text{ itree})$ nodes into `Vis` ϵ $(\alpha' \to (\alpha', \epsilon, \gamma'))$ nodes.

Compositional definition

Our recursive handler `h_prog` follows nearly identical semantic behaviour for each of the program command terms of the abstract syntax, as occurs in the FBS semantics. In fact, before the massaging of the itree type into its final shape, the type of `Ret` nodes is identical to that of the FBS semantics and the results produced are identical. This is an important component of the design that establishes soundness of the itree semantics.

3.2.4 Top-level semantics

The top-level semantics is defined by `itree_semantics` and follows a nearly identical design to that of the FBS semantics. The meaning of a Pancake program is defined in terms of tail call to a single entrypoint and a state[1].

The type of itree in the top-level semantics is identical to that of the massaged type produced in the evaluation semantics by `itree_evaluate`.

[1] the entrypoint is assumed to be defined in the `code` field of the state

Bibliography

[1] Scott Owens et al. "Functional Big-Step Semantics". In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 589–615. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_23.

[2] Li-Yao Xia. "Executable Denotational Semantics With Interaction Trees". In: (), p. 107.