Filet-o-Fish
When French Cuisine Meets Swiss Fishes

Barrelfish Technical Note 024

Pierre-Evariste Dagand

02.06.2017

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
http://www.barrelfish.org/
Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>02.06.2017</td>
<td>TR</td>
<td>Converted to Technical Note from Pierre’s LHS</td>
</tr>
</tbody>
</table>
Contents

1 The Filet-o-Fish Language 7

1 Filet-o-Fish Syntax 9
  1.1 Filet-o-Fish pure expressions 9
    1.1.1 Types 9
    1.1.2 Pure Expressions 12
  1.2 Filet-o-Fish standard constructs 15
    1.2.1 Functor instance 17

2 Filet-o-Fish Semantics 19
  2.1 Functional core interpreter 19
  2.2 Building the FoF interpreter and compiler 23
    2.2.1 Gluing the Interpreter 23
    2.2.2 Gluing the Compiler 24
  2.3 Plumbing Machinery 25
    2.3.1 The Semantics Monad 26
    2.3.2 Folding the Free Monad 26
    2.3.3 Sequencing in the Free Monad 26

3 Filet-o-Fish Operators 27
  3.1 Arrays 27
    3.1.1 Smart Constructors 27
    3.1.2 Run Instantiation 28
    3.1.3 Compile Instantiation 28
  3.2 Conditionals 29
    3.2.1 Smart Constructors 29
    3.2.2 Compile Instantiation 30
    3.2.3 Run Instantiation 31
  3.3 Enumeration 33
    3.3.1 Smart Constructors 33
    3.3.2 Compile Instantiation 33
    3.3.3 Run Instantiation 34
  3.4 Function Definition 34
    3.4.1 Smart Constructors 34
    3.4.2 Compile Instantiation 35
    3.4.3 Run Instantiation 35
  3.5 Reference Cells 36
    3.5.1 Smart Constructors 36
    3.5.2 Compile Instantiation 37
    3.5.3 Run Instantiation 37
  3.6 Strings 38
    3.6.1 Smart Constructors 38
    3.6.2 Compile Instantiation 38
    3.6.3 Run Instantiation 38
  3.7 Structures Definition 39
3.7.1 Smart Constructors .................................................. 39
3.7.2 Compile Instantiation .............................................. 40
3.7.3 Run Instantiation .................................................... 40
3.8 Type Definition .......................................................... 41
3.8.1 Smart Constructors ............................................... 41
3.8.2 Compile Instantiation ............................................. 42
3.8.3 Run Instantiation .................................................... 42
3.9 Unions Definition ........................................................ 42
3.9.1 Smart Constructors ............................................... 42
3.9.2 Compile Instantiation ............................................. 43
3.9.3 Run Instantiation .................................................... 44
4 Lib-C Operators .......................................................... 45
4.1 Printf ............................................................................. 45
4.1.1 Smart Constructors ............................................... 45
4.1.2 Compile Instantiation ............................................. 45
4.1.3 Run Instantiation .................................................... 46
4.2 Assert ............................................................................ 46
4.2.1 Smart Constructors ............................................... 46
4.2.2 Compile Instantiation ............................................. 46
4.2.3 Run Instantiation .................................................... 46
5 Lib-barrefish Operators .................................................. 47
5.1 Has Descendants .......................................................... 47
5.1.1 Smart Constructors ............................................... 47
5.1.2 Compile Instantiation ............................................. 47
5.1.3 Run Instantiation .................................................... 48
5.2 Mem To Phys ............................................................... 48
5.2.1 Smart Constructors ............................................... 48
5.2.2 Compile Instantiation ............................................. 48
5.2.3 Run Instantiation .................................................... 48
II The Filet-o-Fish Compiler ............................................... 49
6 The FoF Intermediate Language ........................................ 51
6.1 The FoF Intermediate Language .................................... 51
6.2 Translating FoFCode to IL.FoF ...................................... 52
6.2.1 The compiler .......................................................... 52
6.2.2 The machinery ....................................................... 53
6.3 Evaluator ..................................................................... 55
7 The Paka Intermediate Language ....................................... 56
7.1 The Paka Intermediate Language .................................... 56
7.2 Paka building blocks .................................................... 58
7.2.1 Low-level machinery .............................................. 58
7.2.2 Building PakaCode ................................................. 59
7.2.3 Building PakaIntra .................................................. 60
7.2.4 Building ILPaka ........................................................ 60
7.3 Translating IL.FoF to IL.Paka ......................................... 61
7.3.1 Compiling Function definition ................................... 61
7.3.2 Compiling Constant ............................................... 62
7.3.3 Compiling Closing statements ................................... 62
7.3.4 Compiling control-flow operators ................................ 62
7.3.5 Compiling statements ............................................. 64
7.4 Translating IL.Paka to C ................................................ 72
7.4.1 Printing types and expressions ............................................. 72
7.4.2 Names, everywhere ......................................................... 74
7.4.3 Generating C ............................................................... 75
7.5 IL.Paka Code Optimizer ...................................................... 77
7.5.1 Implementation ............................................................. 77
7.5.2 Code predication ............................................................ 79
7.5.3 Code transformation ....................................................... 79

III Appendix

A Future Work

81
82
Introduction

Filet-O-Fish, abbreviated FoF hereafter, is a tool for the working language designer. Developed in the context of Barrelfish[5], FoF aims at easing the development of Domain-Specific Languages (DSL) as well as enhancing their safety. As a side effect of FoF’s design, it also becomes easier for the user of a DSL to understand “what is going on”.

To achieve this goal, Filet-o-Fish defines a set of combinators. A combinator is a Haskell function manipulating some Haskell data-types. In this case, our combinators manipulate an abstraction of the C language constructs, such as integers, floats, structures, arrays, etc. Altogether, this set of combinators defines an embedded language in Haskell. To avoid the confusion with the DSLs we are willing to implement, we term this embedded language the meta-language.

You seems confused now. Listen. The Hamlet compiler is implemented with Filet-o-Fish. Hamlet is a Domain-Specific Language. In Hamlet’s compiler, we use FoF to get the job done, i.e. to get the actual C code out of our capability system description. Hence, the Hamlet compiler is partly developed in the FoF meta-language. Understood?

However, Filet-o-Fish is much more than a language to get the job done: being able to compile the meta-language to C is just one side-effect of our work. By writing a DSL compiler with FoF, you actually define the semantics of the DSL. Whereas the syntax defines the set of legal expressions of a language, the semantics assign a meaning to the terms of the language. Note that the C language does not have any formal semantics. And, no, this is not normal. This is Evil.

For a DSL, the benefit of having a formal semantics is twofold. First, the semantics of your DSL is the most precise and accurate description of the behavior of your domain-specific constructs. An informal, in-English specification of the DSL might fail to capture some specific points. The formal semantics is an ultimate documentation, which doesn’t lie. Second, defining a formal semantics is a necessary step before any compiler correctness proof, be it mechanized or on paper. Therefore, thanks to FoF, you get a formal, mechanized semantics of your DSL. And this is for free.

Finally, this document is the literate Haskell code of Filet-o-Fish: the code described in the following pages is the one that is compiled by the Haskell compiler. Therefore, this is the most accurate, up-to-date documentation of FoF’s internals.

So much marketing, let us look at the code.
Part I

The Filet-o-Fish Language
The Filet-O-Fish Language

Give me back that Filet-O-Fish, Give me back that Filet-O-Fish, …

Frankie the Fish

Filet-o-Fish is organized in a modular way. This is reflected by the definition of the syntax of the language in Chapter 1. Indeed, the language is organized around the purely functional core of C, as described in Section 1.1. This core is extended by several constructs that are the operationally rich building blocks of the language, as described in Section 1.2.

The functional semantics of this language is then implemented in Chapter 2. Following the modular definition of the language, we first implement an interpreter for the core language (Section 2.1). In Section 2.2, we gather the per-construct interpreter under one general function. In Section 2.3, we build the machinery to automatically compute an interpreter and a compiler for the whole language.

Further, in Chapter 3, we implement the interpreter and Filet-o-Fish interpretation of the constructs. Similarly, Chapter 4 and Chapter 5, we define foreign functions mirroring the C library and the barreelfish library. These chapters are bound to be extended as long as foreign functions are needed. This is a natural process made easy by the modular design of the syntax and semantics of Filet-o-Fish.
Chapter 1

Filet-o-Fish Syntax

- None shall pass.
- I have no quarrel with you, good Sir Knight, but I must cross this bridge.
- Then you shall die.  

Monty Python

1.1 Filet-o-Fish pure expressions

The core of Filet-o-Fish is organized around the purely functional core of C. It consists of C types as well as C expressions.

1.1.1 Types

Data-type Definitions

The TypeExpr data-type encompasses the following types:

- Void,
- Integers, of various signedness and size,
- Float,
- Named structures and unions,
- Named pointers, i.e. a pointer recurring in a structure or union,
- Arrays, and
- Pointers

Note that a value of type TInt or TFloat is a constant, like 2, 3/7, or sizeof(struct foo). In FoF meta-language, a C variable is not a value – but a construct. So, the type of the variable x defined by int32_t x = 4 is not TInt Signed TInt32.

\[
data\ TypeExpr = TVoid \\
| TInt\ Signedness\ Size \\
| TFloat \\
| TChar
\]
Functions  A function is represented by an Haskell function, taking a list of arguments and computing the body of the function. In the jargon, this is called an higher-order abstract syntax. So, the function definition is represented by the following type:

\[
\text{data } \text{Function} = \text{Fun} \left( \left[ \text{PureExpr} \right] \rightarrow \text{FoFCode PureExpr} \right)
\]

Because \text{TypeExpr} is showable, \text{Function} has to be showable too. While we could define a more complete \text{Show} instance for \text{Function}, we will not do so here and simply return an opaque name.

\[
\text{instance } \text{Show Function where}
\quad \text{show } _{\_} = "<\text{fun}>"
\]

Concerning equality, this becomes more tricky. We would have to define what “equality” means and if that definition is decidable. Here, we consider syntactic equality and although we could decide whether two functions are syntactically equal or not, we will not do so for the moment. We simply consider functions as always distinct.

\[
\text{instance } \text{Eq Function where}
\quad _{-\equiv-} _{\_} = \text{False}
\]

Composed data-types  Composed data-types have several allocation policies: they might be declared dynamically, using malloc, or statically, on the stack. This is reflected by the following definitions. We chose to use different definitions for each kind of data-type because they are likely to evolve in future versions and diverge from this common scheme.

\[
\begin{align*}
\text{data } \text{AllocStruct} &= \text{StaticStruct} \\
& \quad | \text{DynamicStruct} \\
& \quad \text{deriving } (\text{Eq}, \text{Show}) \\
\text{data } \text{AllocUnion} &= \text{StaticUnion} \\
& \quad | \text{DynamicUnion} \\
& \quad \text{deriving } (\text{Eq}, \text{Show}) \\
\text{data } \text{AllocArray} &= \text{StaticArray } \text{Int} \\
& \quad | \text{DynamicArray} \\
& \quad \text{deriving } (\text{Eq}, \text{Show})
\end{align*}
\]

Both Structures and Unions rely on the \text{TFieldList} synonym. Basically, the type of a Structure corresponds to its name as well as the list of its field names and respective types.

\[
\text{type } \text{TFieldList} = \left[ \left( \text{String}, \text{TypeExpr} \right) \right]
\]
Integers  Signedness and size of integers is defined as usual. An integer is either signed or unsigned and its size may vary from 8 to 64 bits. Interestingly, we derive Ord on these data-types: Ord provides us with a comparison function on the signedness and size. In practice, we can check that a cast is a correct downcasting by enforcing that the sign and size we cast to is bigger than the original sign and size.

```haskell
data Signedness = Unsigned
  | Signed
  deriving (Eq, Ord, Show)
data Size = TInt8
  | TInt16
  | TInt32
  | TInt64
  deriving (Eq, Ord, Show)
```

Pointers  As we understand that the suspense is unbearable, we are going to reveal you the type of \( x \) defined above. Actually, the type of \( x \) is TPointer (TInt Signed TInt32) Avail. A pointer? Indeed, a variable does actually points to a location in memory. This choice allows us to capture the notion of variables and pointers in a single abstraction, called a reference cell.

A reference cell can be in one of the following states: either Available or Read. This distinction makes sense during the compilation process, it can ignored otherwise.

```haskell
data Mode = Avail
  | Read
  deriving (Eq, Show)
```

Smart Constructors  In some circumstances, it is necessary to explicitly write the type of an expression. However, explicitly combining the previously defined types can be quite cumbersome. For example, we can naturally define the base types as follow:

- `voidT :: TypeExpr`
- `voidT = TVoid`
- `uint8T, uint16T, uint32T, uint64T :: TypeExpr`
- `uint8T = TInt Unsigned TInt8`
- `uint16T = TInt Unsigned TInt16`
- `uint32T = TInt Unsigned TInt32`
- `uint64T = TInt Unsigned TInt64`
- `int8T, int16T, int32T, int64T :: TypeExpr`
- `int8T = TInt Signed TInt8`
- `int16T = TInt Signed TInt16`
- `int32T = TInt Signed TInt32`
- `int64T = TInt Signed TInt64`
- `floatT :: TypeExpr`
- `floatT = TFloat`
- `charT :: TypeExpr`
- `charT = TChar`

- `uintptrT :: TypeExpr`
- `uintptrT = TCompPointer "void"`
And, similarly, we can build up composed types by applying them on smaller types:

```plaintext
arrayDT :: TypeExpr → TypeExpr
arrayDT typ = TArray DynamicArray typ
arrayST :: Int → TypeExpr → TypeExpr
arrayST size typ = TArray (StaticArray size) typ
ptrT :: TypeExpr → TypeExpr
ptrT typ = TPointer typ Avail
structDT, unionDT,
   structST, unionST :: String → TFieldList → TypeExpr
structDT name fields = TStruct DynamicStruct name fields
unionDT name fields = TUnion DynamicUnion name fields
structST name fields = TStruct StaticStruct name fields
unionST name fields = TUnion StaticUnion name fields
enumT :: String → [(String, Int)] → TypeExpr
enumT name fields = TEnum name fields
typedef :: TypeExpr → String → TypeExpr
typedef typ name = TTypedef typ name
```

Finally, the named pointer – which is actually a fix-point – takes as input the name of the structure or union it refers to.

```plaintext
cptrT :: String → TypeExpr
cptrT id = TCompPointer id
```

### 1.1.2 Pure Expressions

In a first step, we are going to define the expressions composing FoF meta-language. As for types, this consists in a data-type, `PureExpr`, capturing the syntax of expressions. Then, we also define some smart constructors.

#### Data-type Definitions

An expression is one of the following object:

- `void`, the only object populating the type `Void`,
- an integer, of specific signedness and size,
- a float,
- a reference to an object in memory,
- a unary operation, applied to an object,
- a binary operation, applied on two objects,
- the `sizeof` operator, applied to a type,
- a conditional expression, testing an object against 0, returning one of two objects, and
- a `cast` operator, casting an object to a given type

```plaintext
data PureExpr = Void
               | CLInteger Signedness Size Integer
               | CLFloat Float
               | CLChar Char
```
Variable names  A reference is identified by a name. A Generated name has been forged by FoF. A Provided name has been defined by the compiler designer. An Inherited name results from an operation performed on another variable. We carefully track the origin of names for compilation purpose: for example, if a variable name has been Generated, we should try to eliminate it, to make the compiled code more readable.

```haskell
data VarName = Generated String |
                  Provided String |
                  Inherited Int VarName  
  deriving  (Eq, Show)
```

A reference is also decorated by its origin. This field is used by the compiler to identify the scope of variables. Therefore, the compiler can enforce some safety checks, such as verifying that the address of a local variable is not assigned to a global one, for example. Sadly, this information is not always precisely maintained nor correctly used in the current implementation. More care and more checks should be added in the future, to ensure the correctness of the generated code.

```haskell
data Origin = Local  |
                  Global     |
                  Param |
                  Dynamic |
  deriving  (Eq, Show)
```

Unary operations  The unary operations are either the arithmetic minus operation, or the logic complement operation, or the logic negation operation.

```haskell
data UnaryOp = Minus | Complement | Negation  
  deriving  (Eq, Show)
```

Binary operations  The binary operations are either arithmetic operators (+, -, ×, /, and %), Boolean operators (<<, >>, &, bitwise-or, and '), or comparison operators (<, <=, >, >=, ==, and !=).

```haskell
data BinaryOp = Plus | Sub | Mul | Div | Mod |
                  Shl | Shr | AndBit | OrBit | XorBit |
                  Le | Leq | Ge | Geq | Eq | Neq |
  deriving  (Eq, Show)
```

Smart Constructors

As usual, we define some constructors for the C programmer to feel at home with FoF. Let us start with the constants first:
Then come the unary operators:

- `minus`, `comp`, `neg` :: PureExpr → PureExpr
  
  `minus` = Unary Minus
  `comp` = Unary Complement
  `neg` = Unary Negation

And the binary operators. Note that they are defined infix. Therefore, it becomes possible to write the following code:

- `exampleInfix` :: PureExpr
  
  `exampleInfix` = (uint8 1) .< . ((uint8 2) .+ . (uint8 4))

Although not specified yet, we could have set up the left/right associativity and precedence rules of these operators. This would reduce the parenthesizing overhead. It is just a matter of doing it.

- `.+` = Binary Plus
- `.-` = Binary Sub
- `.*` = Binary Mul
- `./` = Binary Div
- `/%` = Binary Mod
- `.<<` = Binary Shl
- `.<>` = Binary Shr
- `.</>` = Binary AndBit
- `./.` = Binary OrBit
- `./.` = Binary XorBit
- `<.` = Binary Le
- `<=.` = Binary Leq
- `>.` = Binary Ge

FiletOFish - 14 Barrelfish TN-024
Finally, \texttt{sizeof}, conditionals, and \texttt{cast} have their straightforward alter-ego in FoF:

\begin{align*}
\textit{sizeof} &:: \text{TypeExpr} \rightarrow \text{PureExpr} \\
\textit{sizeof t} &\equiv \text{Sizeof } t \\
\textit{test} &:: \text{PureExpr} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr} \\
\text{test} c \text{ ift iff} &\equiv \text{Test } c \text{ ift iff} \\
\textit{cast} &:: \text{TypeExpr} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr} \\
\text{cast t e} &\equiv \text{Cast } t \text{ e}
\end{align*}

When compiling foreign function calls, one might need to turn a (Haskell) string into a FoF quote object. This is achieved by the following combinator. One must avoid using this operation as much as possible: this quotation has no semantic meaning, therefore one should use it only when we are really sure we are not interested in the quoted semantic anymore.

\begin{align*}
\textit{quote} &:: \text{String} \rightarrow \text{PureExpr} \\
\text{quote s} &\equiv \text{Quote } s
\end{align*}

\section{Filet-o-Fish standard constructs}

The FoF language is defined by the syntax tree below. It gathers every constructs defined in the \texttt{Constructs} directory as well as foreign functions defined in the \texttt{Libc} and \texttt{Libbarrelfish} directories.

\begin{verbatim}
data FoFConst a = Assert PureExpr a

Foreign-call to libc Assert:

| Printf String [PureExpr] a

Foreign-call to libc Print:

Foreign-call to libarrelfish has descendants:

| HasDescendants (Maybe String) PureExpr (PureExpr \rightarrow a)

Foreign-call to libarrelfish mem to phys:

| MemToPhys (Maybe String) PureExpr (PureExpr \rightarrow a)

Foreign-call to Hamlet get address:

| GetAddress (Maybe String) PureExpr (PureExpr \rightarrow a)

Support for Union:

| NewUnion (Maybe String) AllocUnion String [(String, TypeExpr)] (String, Data) (Loc \rightarrow a) \\
| ReadUnion Loc String (Data \rightarrow a) \\
| WriteUnion Loc String Data a

Support for Typedef:
\end{verbatim}
typedef TypeExpr a
typedef String TypeExpr a

Support for Structures:

| NewStruct (Maybe String) AllocStruct String [(String, (TypeExpr, Data))] (Loc → a) |
| ReadStruct Loc String (Data → a) |
| WriteStruct Loc String Data a |

Support for Strings:

| NewString (Maybe String) String (Loc → a) |

Support for Reference cells:

| NewRef (Maybe String) Data (Loc → a) |
| ReadRef Loc (Data → a) |
| WriteRef Loc Data a |

Support for Functions:

| NewDef [FunAttr] String Function TypeExpr [(TypeExpr, Maybe String)] (PureExpr → a) |
| CallDef (Maybe String) PureExpr [PureExpr] (PureExpr → a) |
| Return PureExpr |

Support for Enumerations:

| NewEnum (Maybe String) String Enumeration String (Loc → a) |

Support for Conditionals:

| If (FoFCode PureExpr) (FoFCode PureExpr) (FoFCode PureExpr) a |
| For (FoFCode PureExpr) (FoFCode PureExpr) (FoFCode PureExpr) a |
| While (FoFCode PureExpr) (FoFCode PureExpr) a |
| DoWhile (FoFCode PureExpr) (FoFCode PureExpr) a |
| Switch PureExpr [(PureExpr, FoFCode PureExpr)] (FoFCode PureExpr) a |
| Break |
| Continue |

Support for Arrays:

| NewArray (Maybe String) AllocArray [Data] (Loc → a) |
| ReadArray Loc Index (Data → a) |
| WriteArray Loc Index Data a |

The following type synonyms have been used above as a documentation purpose. A Data represents a value used to initialize a data-structure. A Loc represents a reference. An Index is a value used to index an array.

```plaintext
type Data = PureExpr
type Loc = PureExpr
type Index = PureExpr
```
Function attributes  
A function can be characterized by the following attributes, following their C semantics:

```haskell
data FunAttr = Static
| Inline

deriving (Eq)

instance Show FunAttr where
  show Static = "static"
  show Inline = "inline"
```

Enumeration  
When defining an enumeration, we use the following type synonym to describe the list of pair name-value:

```haskell
type Enumeration = [(String, Int)]
```

### 1.2.1 Functor instance

A crucial specificity of FoFConst is that it defines a functor. This functor is defined as follow.

```haskell
instance Functor FoFConst where
  fmap f (Assert a b) = Assert a (f b)
  fmap f (Printf a b c) = Printf a b (f c)
  fmap f (HasDescendants a b c) = HasDescendants a b (f ∘ c)
  fmap f (MemToPhys a b c) = MemToPhys a b (f ∘ c)
  fmap f (GetAddress a b c) = GetAddress a b (f ∘ c)
  fmap f (NewUnion a b c d e g) = NewUnion a b c d e (f ∘ g)
  fmap f (ReadUnion a b c) = ReadUnion a b (f ∘ c)
  fmap f (WriteUnion a b c d) = WriteUnion a b c d (f d)
  fmap f (Typedef a c) = Typedef a (f c)
  fmap f (TypedefE a b c) = TypedefE a b (f c)
  fmap f (NewStruct a b c d e f) = NewStruct a b c d e (f ∘ e)
  fmap f (ReadStruct a b c) = ReadStruct a b (f ∘ c)
  fmap f (WriteStruct a b c d) = WriteStruct a b c d (f d)
  fmap f (NewString a b c) = NewString a b (f c)
  fmap f (NewRef a b c) = NewRef a b (f ∘ c)
  fmap f (ReadRef a b c) = ReadRef a b (f ∘ b)
  fmap f (WriteRef a b c) = WriteRef a b (f c)
  fmap f (NewDef a b c d e f) = NewDef a b c d e (f ∘ e)
  fmap f (CallDef a b c d) = CallDef a b c d (f d)
  fmap f (Return a) = Return a
  fmap f (NewEnum a b c d e) = NewEnum a b c d e (f ∘ e)
  fmap f (If a b c d f) = If a b c (f d)
  fmap f (For a b c d e) = For a b c d e (f e)
  fmap f (While a b c) = While a b (f c)
  fmap f (DoWhile a b c) = DoWhile a b (f c)
  fmap f (Switch a b c d) = Switch a b c d (f d)
  fmap f (Break) = Break
  fmap f (Continue) = Continue
  fmap f (NewArray a b c d) = NewArray a b c d (f d)
  fmap f (ReadArray a b c) = ReadArray a b (f ∘ c)
  fmap f (WriteArray a b c d) = WriteArray a b c d (f d)
```
Thanks to this functor structure, it makes sense to embed \textit{FoFConst} in a \textit{Semantics}: the machinery we build in Chapter 2.3 will take care of transforming this functor into a free monad. Hence the following type synonym.

\begin{verbatim}
  type FoFCode a = Semantics FoFConst a
\end{verbatim}
Chapter 2

Filet-o-Fish Semantics

So, logically...
If...
she...
weighs...
the same as a duck,...
she’s made of wood.

Monty Python

2.1 Functional core interpreter

In this Section, we implement an expression evaluator. Given any (correct) expression, it will compute the corresponding value. The implementation is decomposed in several steps. In Section 2.1, we evaluate top-level expressions. Doing so, we rely on case-specific evaluators. This includes unary operators (Section 2.1), binary operators (Section 2.1), the sizeof operation (Section 2.1), the conditional operation (Section 2.1), and the cast operation (Section 2.1).

Note that the following functions are partial: not all expressions can be successfully evaluated. Indeed, some operations are simply meaningless. For example, computing the sum of a structure and a float is illegal. Currently, we are simply ignore these errors and this might result in run-time errors of the DSL compiler. Satisfactory solutions of this problem exist, though. For example, we could implement a type-checker that would ensure the absence of run-time errors. Another approach would be improve our error handling code.

Top-level Evaluation

The purpose of this section is implement the following function:

\[
symbEval :: PureExpr \rightarrow PureExpr
\]

That reduces a given expression to a value. Hence, for values, this is trivial:

\[
symbEval \ Void = \ Void
\]
\[
symbEval \ x @(CLInteger \_ _ ) = x
\]
\[
symbEval \ x @(CLFloat \_ _ ) = x
\]
\[
symbEval \ x @(CLRef \_ _ ) = x
\]

Then, for inductive constructions, we rely on the specific functions implemented in the following sections.
symbEval (Unary op x) = symbEvalUnary op x'  
  where x' = symbEval x
symbEval (Binary op x y) = symbEvalBinary op x' y'  
  where x' = symbEval x y' = symbEval y
symbEval (Sizeof typ) = symbEvalSizeof typ
symbEval (Test x y z) = symbEvalTest x' y z  
  where x' = symbEval x
symbEval (Cast t x) = symbEvalCast t x'  
  where x' = symbEval x

Unary Operator Evaluation

For unary operators, we need to implement the following function:

\[
\text{symbEvalUnary :: UnaryOp} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr}
\]

Hence the following code:

\[
\text{symbEvalUnary Minus x = case x of }
\text{ CLInteger Signed size x \rightarrow CLInteger Signed size (-x)}
\text{ CLFloat x \rightarrow CLFloat (-x)}
\text{ _ \rightarrow error "symbEvalUnary: minus on wrong type"}
\]
\[
\text{symbEvalUnary Complement x = case x of }
\text{ CLInteger sg sz x \rightarrow CLInteger sg sz (complement x)}
\text{ _ \rightarrow error "symbEvalUnary: complement on wrong type"}
\]
\[
\text{symbEvalUnary Negation x = case x of }
\text{ CLInteger sg sz 0 \rightarrow CLInteger sg sz 1}
\text{ CLInteger sg sz - \rightarrow CLInteger sg sz 0}
\text{ _ \rightarrow error "symbEvalUnary: negation on wrong type"}
\]

Binary Operator Evaluation

For binary operators, here is our goal:

\[
\text{symbEvalBinary :: BinaryOp} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr} \rightarrow \text{PureExpr}
\]

Achieved by the following, messy codes.

**Arithmetic Operations**

\[
\text{symbEvalBinary Plus (CLInteger sg si x) (CLInteger sg' si' y) = }
\text{ | sg \equiv sg' \land si \equiv si' = CLInteger sg si (x + y)}
\text{ | otherwise = error "symbEvalBinary: Plus undefined"}
\]
\[
\text{symbEvalBinary Plus (CLInteger _ _ x) (CLFloat y) =}
\]

FiletOFish - 20

Barrelfish TN-024
More checks should be added here. For examples, we should ensure that the result of the subtraction of two unsigned numbers is still positive, or make it wrap.

```plaintext
symbEvalBinary Mod (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x \( \mod \) y) | otherwise = error "symbEvalBinary: Mod undefined"
symbEvalBinary Div (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x \( \div \) y) | otherwise = error "symbEvalBinary: Div undefined"
symbEvalBinary Mod (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x \( \mod \) y) | otherwise = error "symbEvalBinary: Mod undefined"
```

**Boolean Operations**

```plaintext
symbEvalBinary Shl (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (shiftL x (fromInteger y)) | otherwise = error "symbEvalBinary: Shl undefined"
symbEvalBinary Shr (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (shiftR x (fromInteger y)) | otherwise = error "symbEvalBinary: Shr undefined"
symbEvalBinary AndBit (CLInteger sg si x) (CLInteger sg' si' y) | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x \( \& \) y) | otherwise = error "symbEvalBinary: And undefined"
```
symbEvalBinary AndBit _ _ = error "symbEvalBinary: And undefined"
symbEvalBinary OrBit (CLInteger sg si x) (CLInteger sg' si' y)
    | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x B.&. y)
    | otherwise = error "symbEvalBinary: Or undefined"
symbEvalBinary XorBit (CLInteger sg si x) (CLInteger sg' si' y)
    | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x `xor` y)
    | otherwise = error "symbEvalBinary: Xor undefined"
symbEvalBinary XorBit _ _ = error "symbEvalBinary: Xor undefined"

Comparison Operations

symbEvalBinary op (CLInteger sg si x) (CLInteger sg' si' y)
    | sg ≡ sg' ∧ si ≡ si' = symbEvalComp op x y
    | otherwise = error ("symbEvalBinary: "+ show op ++ " undefined")
symbEvalBinary op (CLInteger _ x) (CLFloat y) =
    symbEvalComp op (fromRational $ toRational x) y
symbEvalBinary op (CLFloat x) (CLInteger _ y) =
    symbEvalComp op x (fromRational $ toRational y)
symbEvalBinary op (CLFloat x) (CLFloat y) = symbEvalComp op x y

symbEvalBinary Le _ _ = error "symbEvalBinary: Le undefined"
symbEvalBinary Leq _ _ = error "symbEvalBinary: Leq undefined"
symbEvalBinary Ge _ _ = error "symbEvalBinary: Leq undefined"
symbEvalBinary Geq _ _ = error "symbEvalBinary: Leq undefined"
symbEvalBinary Eq _ _ = error "symbEvalBinary: Leq undefined"
symbEvalBinary Neq _ _ = error "symbEvalBinary: Leq undefined"

symbEvalComp :: (Ord a, Num a) ⇒ BinaryOp → a → a → PureExpr
symbEvalComp op x y =
    let cmp = case op of
        Le → (<)
        Leq → (≤)
        Ge → (≥)
        Geq → (⩾)
        Eq → (≡)
        Neq → (̸≡) in
    if cmp x y then
        CLInteger Unsigned TInt64 1
    else CLInteger Unsigned TInt64 0

Sizeof Evaluation

Our sizeof operator follows the corresponding C operation:

symbEvalSizeof :: TypeExpr → PureExpr
symbEvalSizeof TVoid = CLInteger Unsigned TInt64 1
symbEvalSizeof (TInt _ TInt8) = CLInteger Unsigned TInt64 1
symbEvalSizeof (TInt _ TInt16) = CLInteger Unsigned TInt64 2
symbEvalSizeof (TInt _ TInt32) = CLInteger Unsigned TInt64 4
symbEvalSizeof (TInt _ TInt64) = CLInteger Unsigned TInt64 8
symbEvalSizeof TFloat = CLInteger Unsigned TInt64 4

FiletOFish - 22
Barrelfish TN-024
symbEvalSizeof (TPointer _) = CLInteger Unsigned TInt64 8
symbEvalSizeof (TCompPointer _) = CLInteger Unsigned TInt64 8
symbEvalSizeof (TArray _, typ) = CLInteger Unsigned TInt64 8
symbEvalSizeof (TStruct _, fields) = CLInteger Unsigned TInt64 8
symbEvalSizeof (TUnion _, fields) = CLInteger Unsigned TInt64 8

Conditionals Evaluation

The semantics of the conditional mimics a restricted version of the C standard: True corresponds to everything which is not a float or integer equal to zero. Hence, we evaluate the corresponding branch accordingly.

symbEvalTest :: PureExpr → PureExpr → PureExpr → PureExpr
symbEvalTest (CLInteger _ _ 0) _ y = symbEval y
symbEvalTest (CLFloat 0) _ y = symbEval y
symbEvalTest _ x _ = symbEval x

Cast Evaluation

Here is our stripped-down version of cast. It will probably deserve some work in the future, as it is quite restrictive. Also, it should ensure that the type modification are reflected on the data: converting a signed, negative number to an unsigned form changes the value of this number. This is currently unsupported.

symbEvalCast :: TypeExpr → PureExpr → PureExpr
symbEvalCast (TInt sg sz) (CLInteger sg' sz' x)
    | sg' < sg ∧ sz' < sz = CLInteger sg sz x
    | otherwise = error "symbEvalCast: illegal integer cast"
symbEvalCast TFloat (CLInteger _ _ x) =
    CLFloat (fromRational $ toRational x)
symbEvalCast TFloat vx@(CLFloat x) = vx
symbEvalCast _ _ _ =
    error "symbEvalCast: Not yet implemented/undefined cast"

2.2 Building the FoF interpreter and compiler

In this section, we glue together the constructs of the FoF language, defined in the Constructs, Libc, and Libbarrelfish directories. This gluing builds a one-step interpreter for FoF, compileAlgebra (Section 2.2.1), and a one-step compiler, compileAlgebra (Section 2.2.2). We rely on the machinery defined in Section 2.3 to automatically build an interpreter and a compiler from these functions.

2.2.1 Gluing the Interpreter

The run-time is actually quite simple. It is described by a heap, in which we first store fresh identifiers, freshLoc, freshSLoc, and freshALoc. When we want to store a value in memory, we pick a fresh identifier and, respectively update the refMap, strMap, or arrayMap with a new map from the identifier to the value. Similarly, we can read and modify these mappings. Intuitively, the Heap is a representation of the machine’s memory.
These different maps have different purposes: refMap maps an identifier to a single value, strMap maps an identifier to a mapping from strings to values (modelling a structure or union), and arrayMap maps an identifier to a bounded array of values.

data Heap = Hp { freshLoc :: Int, 
    refMap :: [(VarName, Data)], 
    freshSLoc :: Int, 
    strMap :: [(VarName, [(String, Data)])], 
    freshALoc :: Int, 
    arrayMap :: [(VarName, [Data])] }

Then, the one-step interpreter takes a FoF term, a Heap, and returns a pair of value and resulting heap. This is simply implemented by matching the term and calling the corresponding construct-specific interpreter.

runAlgebra :: FoFConst (Heap -> (PureExpr, Heap)) -> Heap -> (PureExpr, Heap)
runAlgebra x @ (NewArray _) = runArrays x
runAlgebra x @ (ReadArray _) = runArrays x
runAlgebra x @ (WriteArray _) = runArrays x
runAlgebra x @ (If _) = runConditionals x
runAlgebra x @ (For _) = runConditionals x
runAlgebra x @ (While _) = runConditionals x
runAlgebra x @ (DoWhile _) = runConditionals x
runAlgebra x @ (Switch _) = runConditionals x
runAlgebra x @ (Break) = runConditionals x
runAlgebra x @ (Continue) = runConditionals x
runAlgebra x @ (NewEnum _) = runEnumerations x
runAlgebra x @ (NewDef _) = runFunctions x
runAlgebra x @ (CallDef _) = runFunctions x
runAlgebra x @ (Return _) = runFunctions x
runAlgebra x @ (NewRef _) = runReferences x
runAlgebra x @ (ReadRef _) = runReferences x
runAlgebra x @ (WriteRef _) = runReferences x
runAlgebra x @ (NewEnum _) = runString x
runAlgebra x @ (Typedef _) = runTypedef x
runAlgebra x @ (TypedefE _) = runTypedef x
runAlgebra x @ (NewStruct _) = runStructures x
runAlgebra x @ (WriteStruct _) = runStructures x
runAlgebra x @ (NewUnion _) = runUnions x
runAlgebra x @ (WriteUnion _) = runUnions x
runAlgebra x @ (Assert _) = runAssert x
runAlgebra x @ (Printf _) = runPrintf x
runAlgebra x @ (HasDescendants _) = runHasDescendants x
runAlgebra x @ (MemToPhys _) = runMemToPhys x
runAlgebra x @ (GetAddress _) = runGetAddress x

2.2.2 Gluing the Compiler

Similarly, the one-step compiler is organized around the notion of Binding environment: this environment is carried over the compilation process. Hence, the Binding represents the compiler's state:

- **freshVar** is a free identifier, used to generate unique variable names,
- **def ...** maps the defined structure names with their type
data Binding = Binding { freshVar :: Int,
defStructs :: [(String, TypeExpr)],
defUnions :: [(String, TypeExpr)],
defEnums :: [(String, [(String, Int)])] }

This binding is then modified by the one-step compiler, which takes a term, a binding, and return an FoF expression as well as an updated binding.

```haskell
compileAlgebra :: FoFConst (Binding → (ILFoF, Binding)) →
  (Binding → (ILFoF, Binding))
compileAlgebra x@ (NewArray _ _ _) = compileArrays x
compileAlgebra x@ (ReadArray _ _ _) = compileArrays x
compileAlgebra x@ (WriteArray _ _ _) = compileArrays x
compileAlgebra x@ (If _ _ _) = compileConditionals x
compileAlgebra x@ (For _ _ _) = compileConditionals x
compileAlgebra x@ (While _ _ _) = compileConditionals x
compileAlgebra x@ (DoWhile _ _ _) = compileConditionals x
compileAlgebra x@ (Switch _ _ _) = compileConditionals x
compileAlgebra x@ (Break) = compileConditionals x
compileAlgebra x@ (Continue) = compileConditionals x
compileAlgebra x@ (NewDef _ _ _) = compileFunctions x
compileAlgebra x@ (CallDef _ _ _) = compileFunctions x
compileAlgebra x@ (Return _ _) = compileFunctions x
compileAlgebra x@ (NewEnum _ _ _) = compileEnumerations x
compileAlgebra x@ (NewRef _ _) = compileReferences x
compileAlgebra x@ (ReadRef _ _) = compileReferences x
compileAlgebra x@ (WriteRef _ _) = compileReferences x
compileAlgebra x@ (NewString _ _) = compileString x
compileAlgebra x@ (Typedef _ _) = compileTypedef x
compileAlgebra x@ (TypedefE _ _) = compileTypedef x
compileAlgebra x@ (NewStruct _ _ _) = compileStructures x
compileAlgebra x@ (ReadStruct _ _ _) = compileStructures x
compileAlgebra x@ (WriteStruct _ _ _) = compileStructures x
compileAlgebra x@ (NewUnion _ _ _) = compileUnions x
compileAlgebra x@ (ReadUnion _ _ _) = compileUnions x
compileAlgebra x@ (WriteUnion _ _ _) = compileUnions x
compileAlgebra x@ (Assert _ _) = compileAssert x
compileAlgebra x@ (Printf _ _) = compilePrintf x
compileAlgebra x@ (HasDescendants _ _) = compileHasDescendants x
compileAlgebra x@ (MemToPhys _ _) = compileMemToPhys x
compileAlgebra x@ (GetAddress _ _) = compileGetAddress x
```

## 2.3 Plumbing Machinery

The material presented in this chapter relies on some hairy concepts from Category Theory. If you are curious about these things, Edward Kmett wrote a nice blog post [2] on the subject. The first version of FoF, and in particular this file, relied on Wouter Swierstra solution to the expression problem [6]. However, the burden of this approach on the type-system was unbearable for our users.

Our motivation is to build a monad in which one can naturally write sequential code, just as an imperative language. Each construct of the language is defined in Constructs by the `FoFConst` data-type. Purposely, this data-type implements a functor. The code below generically turn a functor `f` into a `Semantics f` monad. Hence, in Constructs, we apply this machinery to make a monad out of `FoFConst`.

---

Barrelfish TN-024

FiletOfFish - 25
2.3.1 The Semantics Monad

We build a monad \( \text{Semantics} f \) out of a function \( f \) thanks to the following data-type:

\[
\text{data Semantics } f \ a = \text{Pure } a \\
| \text{Impure } (f \ (\text{Semantics } f \ a))
\]

First of all, we show that this defines a functor:

\[
\text{instance Functor } f \Rightarrow \text{Functor } (\text{Semantics } f) \text{ where}
\]

\[
fmap \ f \ (\text{Pure } x) = \text{Pure } (f \ x)
\]

\[
fmap \ f \ (\text{Impure } t) = \text{Impure } (fmap \ (fmap \ f) \ t)
\]

We need to (as of GHC 7.10) implement Applicative

\[
\text{instance } (\text{Functor } f) \Rightarrow \text{Applicative } (\text{Semantics } f) \text{ where}
\]

\[
\text{pure } = \text{return}
\]

\[
(< \ast >) = \text{ap}
\]

Then, we obtain the monad:

\[
\text{instance } \text{Functor } f \Rightarrow \text{Monad } (\text{Semantics } f) \text{ where}
\]

\[
\text{return } = \text{Pure}
\]

\[
(\text{Pure } x) \gg= f = f \ x
\]

\[
(\text{Impure } t) \gg= f = \text{Impure } (fmap \ (\gg= f) \ t)
\]

Terms are embedded into the monad thanks the following function:

\[
\text{inject } :: f \ (\text{Semantics } f \ a) \rightarrow \text{Semantics } f \ a
\]

\[
\text{inject } x = \text{Impure } x
\]

2.3.2 Folding the Free Monad

Finally, once we have built the monad, we will need to manipulate its content. For example, we will be willing to evaluate it, or to compile it, etc. All these operations can be implemented by folding over the monadic code, that is traversing the constructs in their definition order and computing an output of type \( b \). Note that we have to distinguish \( \text{Pure} \) terms, which are simply values, from \( \text{Impure} \) ones, which are the embedded constructs.

\[
foldSemantics :: \text{Functor } f \Rightarrow (a \rightarrow b) \rightarrow (f \ b \rightarrow b) \rightarrow \text{Semantics } f \ a \rightarrow b
\]

\[
foldSemantics \ \text{pure} \ \text{imp} \ (\text{Pure } x) = \text{pure } x
\]

\[
foldSemantics \ \text{pure} \ \text{imp} \ (\text{Impure } t) = \text{imp} \ \text{fmap} \ (\text{foldSemantics} \ \text{pure} \ \text{imp}) \ t
\]

2.3.3 Sequencing in the Free Monad

Provided a list of monadic code, we are able to turn them into a single monadic code returning a list of terms. This corresponds to the \( \text{sequence} \) function in the \( \text{IO} \) monad:

\[
\text{sequenceSem } ms = \text{foldr } k \ (\text{return } []) \ ms
\]

\[
\text{where } k \ m \ m' =
\]

\[
\text{do}
\]

\[
x \leftarrow m
\]

\[
xs \leftarrow m'
\]

\[
\text{return } (x : xs)
\]
Chapter 3

Filet-o-Fish Operators

Listen.
Strange women lying in ponds distributing swords is no basis for a system of government. Supreme executive power derives from a mandate from the masses, not from some farcical aquatic ceremony.

Monty Python

3.1 Arrays

The Array construct, as well as the subsequent constructs, is organized as follow. First, we define some smart constructors, which are directly used by the DSL designer when implementing the compiler. Then, we implement the one-step interpreter and compiler to FoF.

Array offers an abstraction over C arrays, both statically defined or statically allocated. Hence, it offers the possibility to create, read from, and write into arrays.

3.1.1 Smart Constructors

We can create dynamic and static anonymous arrays using the following combinators:

\[
\text{newArray} :: [\text{Data}] \to \text{FoFCode Loc} \\
\text{newArray value} = \text{inject} (\text{NewArray Nothing DynamicArray value return})
\]

\[
\text{newStaticArray} :: [\text{Data}] \to \text{FoFCode Loc} \\
\text{newStaticArray value} = \text{inject} (\text{NewArray Nothing (StaticArray $ length value)} value return)
\]

Similarly, they can be named:

\[
\text{newArrayN} :: \text{String} \to [\text{Data}] \to \text{FoFCode Loc} \\
\text{newArrayN name value} = \text{inject} (\text{NewArray (Just name) DynamicArray value return})
\]

\[
\text{newStaticArrayN} :: \text{String} \to [\text{Data}] \to \text{FoFCode Loc} \\
\text{newStaticArrayN name value} = \text{inject} (\text{NewArray (Just name) (StaticArray $ length value)} value return)
\]

Then, we can read the content of an array:
readArray :: Loc → Index → FoFCode Data
readArray l f = inject (ReadArray l f return)

As well as write some data in a cell:

writeArray :: Loc → Index → Data → FoFCode ()
writeArray l f d = inject (WriteArray l f d (return ()))

3.1.2 Run Instantiation

The interpretation of an array operation is dispatched by the following code.

runArrays :: FoFConst (Heap → (a, Heap)) → (Heap → (a, Heap))
runArrays (NewArray a b c r) heap = uncurry r $ runNewArray b c heap
runArrays (ReadArray a b r) heap = uncurry r $ runReadArray a b heap
runArrays (WriteArray a b c r) heap = r $ runWriteArray a b c heap

Creating, reading, and writing to or from an array are trivially implemented by the following code:

runNewArray :: AllocArray → [Data] → Heap → (Loc, Heap)
runNewArray alloc initData heap =
  let loc = freshALoc heap in
  let sizeInt = length initData in
  let name = makeVarName Dynamic loc in
  let ref = CLRef Dynamic (TArray alloc $ typeOf $ head initData) name in
  let heap1 = heap { freshALoc = loc + 1, arrayMap = (name, initData) : (arrayMap heap) } in
  (ref, heap1)

runReadArray :: Loc → Index → Heap → (Data, Heap)
runReadArray (CLRef _ (TArray _ _)) loc index heap =
  let array = fromJust $ loc `lookup` (arrayMap heap) in
  let (CLInteger _ indexInt) = symbEval index in
  let val = array !! (fromInteger indexInt) in
  (val, heap)

runWriteArray :: Loc → Index → Data → Heap → Heap
runWriteArray (CLRef _ (TArray _ _)) loc dat heap =
  let array = fromJust $ loc `lookup` (arrayMap heap) in
  let (CLInteger _ indexInt) = symbEval index in
  let (arrayBegin, arrayEnd) = splitAt (fromInteger indexInt) array in
  let array1 = arrayBegin ++ (dat : tail arrayEnd) in
  let heap1 = heap { arrayMap = (loc, array1) : arrayMap heap } in
  heap1

3.1.3 Compile Instantiation

Similarly, the compilation of array operations consists in implementing the following function:

compileArrays :: FoFConst (Binding → (ILFoF, Binding)) →
(Binding → (ILFoF, Binding))

The translation from the FoFConst terms to FoF terms is almost automatic. The added value of this
process consists in generating or deriving names for the references.
compileArrays (NewArray name allocArray dat r) binding =
  let scopeVar
    = case allocArray of
        DynamicArray → Dynamic
        StaticArray → Global in
  let (publicName, binding1) = case name of
    Just x → (Provided x, binding)
    Nothing →
      let (loc, binding1) = getFreshVar binding in
      (makeVarName scopeVar loc, binding1) in
  let typeOfDat = typeOf $ head dat in
  let ret = CLRef Dynamic (TArray allocArray typeOfDat) publicName in
  let (cont, binding2) = r ret binding in
  (FStatement (FNewArray publicName allocArray dat) cont,
   binding2)

compileArrays (ReadArray ref@(CLRef origin (TArray arrayAlloc typ) xloc) index r) binding =
  let (loc, name, binding1) = heritVarName binding xloc in
  let ret = CLRef Dynamic (readOf typ) name in
  let (cont, binding2) = r ret binding in
  (FStatement (FReadArray name ref index) cont,
   binding2)

compileArrays (WriteArray ref@(CLRef origin (TArray arrayAlloc typ) xloc) index dat r) binding =
  let (cont, binding1) = r binding in
  (FStatement (FWriteArray ref index dat) cont,
   binding1)

3.2 Conditionals

The Conditionals constructs consist of all control-flow operators defined in the C language, excepted
the goto statement and fall-through switches.

3.2.1 Smart Constructors

We provide the DSL designer with all standard C control-flow operators. Hence, we define the following
combinators: ifc, for, while, doWhile, break, and continue.

ifc :: FoFCode PureExpr →
     FoFCode PureExpr →
     FoFCode PureExpr →
     FoFCode PureExpr
ifc cond ifTrue ifFalse =
  inject (If cond ifTrue ifFalse (return Void))

for :: FoFCode PureExpr →
     FoFCode PureExpr →
     FoFCode PureExpr →
     FoFCode PureExpr →
     FoFCode PureExpr
The `switch` statement is slightly different from the C one: every case is automatically terminated by a `break` statement. Hence, it is impossible to fall through a case.

```plaintext
switch :: PureExpr →
[(PureExpr, FoFCode PureExpr)] →
FoFCode PureExpr →
FoFCode PureExpr
switch cond cases defaultCase =
  inject (Switch cond cases defaultCase (return Void))
```

### 3.2.2 Compile Instantiation

The compilation step is mostly standard. Note that we often have to compile sub-blocks of code. Therefore, we need to carefully update the relevant binding states, so as to ensure the freshness of generated names while respecting the scope of locally defined variables.

```plaintext
compileConditionals (If condi ifTrue ifFalse r) binding =
(FIf compCond compIfTrue compIfFalse cont,
  binding2)
  where (compCond, binding1) = compileSemtoFoF’ condi binding
        (compIfTrue, binding1’) = compileSemtoFoF’ ifTrue binding1
        (compIfFalse, binding1’’) = compileSemtoFoF’ ifFalse
        (binding1’|− > binding1)
        (cont, binding2) = r (binding1’’|− > binding)

compileConditionals (While condW loop r) binding =
(FWhile compCond compLoop cont,
  binding3)
  where (compCond, binding1) = compileSemtoFoF’ condW binding
        (compLoop, binding2) = compileSemtoFoF’ loop binding1
        (cont, binding3) = r (binding2|− > binding)

compileConditionals (DoWhile loop condD r) binding =
(FDoWhile compLoop compCond cont,
  binding3)
  where (compLoop, binding1) = compileSemtoFoF’ loop binding
        (compCond, binding2) = compileSemtoFoF’ condD
        (binding1|− > binding)
```
(cont, binding3) = r (binding2 | − > binding)

compileConditionals (For init test inc loop r) binding =
(FFor compInit compTest compInc compLoop cont, binding5)
  where (compInit, binding1) = compileSemtoFoF' init binding
       (compTest, binding2) = compileSemtoFoF' test binding1
       (compInc, binding3) = compileSemtoFoF' inc binding2
       (compLoop, binding4) = compileSemtoFoF' loop
       (binding1 | − > binding3)
       (cont, binding5) = r (binding4 | − > binding)

compileConditionals (Switch test cases defaultC r) binding =
(FSwitch test compCases compDefault cont, binding3)
  where compileCase (compCodes, binding) (i, code) =
       ((i, compCode) : compCodes,
         (binding1 | − > binding))
       where (compCode, binding1) = compileSemtoFoF' code binding
       (compCases, binding1) =
         foldl' compileCase ([], binding) cases
       (compDefault, binding2) =
         compileSemtoFoF' defaultC (binding1 | − > binding)
       (cont, binding3) = r (binding2 | − > binding)

compileConditionals Break binding =
(FClosing $ FBreak, binding)

compileConditionals Continue binding =
(FClosing $ FContinue, binding)

3.2.3 Run Instantiation

The implementation of the interpreter is straightforward. We start by dispatching calls to construct-specific functions:

runConditionals (If a b c r) heap =
  r $ runIf a b c heap
runConditionals (For a b c d r) heap =
  r $ runFor a b c d heap
runConditionals (While a b r) heap =
  r $ runWhile a b heap
runConditionals (DoWhile a b r) heap =
  r $ runDoWhile a b heap
runConditionals (Switch a b c r) heap =
  r $ runSwitch a b c heap
runConditionals Break heap =
  error "runAlgebra: Break not yet implemented"
runConditionals Continue heap =
  error "runAlgebra: Continue not yet implemented"

Then, we implement the semantics of each of these constructs:

runIf :: FoFCode PureExpr →
FoFCode PureExpr →
FoFCode PureExpr →
Heap → Heap
runIf test ifTrue ifFalse heap =
let (vtest, heap1) = run test heap in
let CLInteger _ _ valVtest = symbEval vtest in
if (valVtest ≠ 0) then
    let (_ , heap2) = run ifTrue heap1 in
    heap2
else
    let (_ , heap2) = run ifFalse heap1 in
    heap2
runFor :: FoFCode PureExpr →
    FoFCode PureExpr →
    FoFCode PureExpr →
    FoFCode PureExpr →
    Heap → Heap
runFor init test incr loop heap =
    let (_ , heap1) = run init heap in
    loopWhile heap1
    where loopWhile heap =
        let (vtest, heap1) = run test heap in
        let CLInteger _ _ valVtest = symbEval vtest in
        if (valVtest ≠ 0) then
            let (_ , heap2) = run loop heap1 in
            let (_ , heap3) = run incr heap2 in
            loopWhile heap3
        else
            heap1
runWhile :: FoFCode PureExpr →
    FoFCode PureExpr →
    Heap → Heap
runWhile test loop heap =
    let (vtest, heap1) = run test heap in
    let CLInteger _ _ valVtest = symbEval vtest in
    if (valVtest ≠ 0) then
        let (_ , heap2) = run loop heap1 in
        runWhile test loop heap2
    else
        heap1
runDoWhile :: FoFCode PureExpr →
    FoFCode PureExpr →
    Heap → Heap
runDoWhile loop test heap =
    let (_ , heap1) = run loop heap in
    let (vtest, heap2) = run test heap in
    let CLInteger _ _ valVtest = symbEval vtest in
    if (valVtest ≠ 0) then
        runDoWhile loop test heap2
    else
        heap2
runSwitch :: PureExpr →
    [(PureExpr, FoFCode PureExpr)] →
    FoFCode PureExpr →
    Heap → Heap
runSwitch test cases defaultCase heap =
    let res = symbEval test in
case res 'lookup' cases of
    Just stmt → let (_ , heap1) = run stmt heap in
                   heap1
    Nothing → let (_ , heap1) = run defaultCase heap in
                   heap1
3.3 Enumeration

The Enumeration construct mirrors the enum data-type of C. It allows us to name a finite number of natural constants and manipulate these names instead of numbers.

3.3.1 Smart Constructors

The newEnum combinator is used to create a member value belonging to one of the fields of nameEnum.

\[
\text{newEnum :: String → Enumeration → String → FoFCode PureExpr} \\
\text{newEnum nameEnum fields value =} \\
\text{inject (NewEnum Nothing nameEnum fields value return)}
\]

Similarly, newEnumN creates a named member of an enumeration.

\[
\text{newEnumN :: String → String → Enumeration → String → FoFCode PureExpr} \\
\text{newEnumN name nameEnum fields value =} \\
\text{inject (NewEnum (Just name) name fields value return)}
\]

3.3.2 Compile Instantiation

A NewEnum is compiled as follow.

\[
\text{compileEnumerations (NewEnum name enumName vals value r) binding =} \\
(\text{FStatement (FNewEnum publicName enumName vals value) cont, binding3}} \\
\text{where (publicName, binding2)} \\
\text{= case name of} \\
\text{Just x → (Provided x, binding)} \\
\text{Nothing → (makeVarName Local loc, binding1)} \\
\text{where (loc, binding1) = getFreshVar binding} \\
\text{ret = CLRef Global uint64T (Provided value)} \\
\text{(cont, binding3) = r ret binding2}
\]

Note that ret is actually the name of the enumerated value: it is treated as a constant and passed as such to the remaining code. A more standard implementation would have been to create a variable containing this constant value and pass the reference to the variable to the subsequent code. However, when switch-ing over an enumerated value, the case would match a variable instead of a constant, which is refused by the C compiler.

Clearly, a clean solution to this implementation must be found. However, the current solution, if not perfect, seems to be good enough.
3.3.3 Run Instantiation

Running a newEnum simply consists in getting the associated value.

```
runEnumerations (NewEnum _ _ enum name r) heap =
    let ref = uint64 $ toInteger $ fromJust $ name 'lookup' enum in
    r ref heap
```

3.4 Function Definition

This module abstracts the function definition and manipulation mechanisms found in C. This consists in a def constructor, to define functions, a call and callN functions to call functions, as well as a returnc combinator to return from a function call.

3.4.1 Smart Constructors

When defining a function, we provide a list of attributes, its name, its body, its return type, and a list of arguments types:

```
def :: [FunAttr] →
    String →
    ([PureExpr] → FoFCode PureExpr) →
    TypeExpr →
    [([TypeExpr, Maybe String]) →
    FoFCode PureExpr

def attr name fun returnT argsT =
    inject (NewDef attr name (Fun fun) returnT argsT return)
```

Then, it is possible to call into a function, provided a list of parameters. The result, if any, can be named by using the callN construct.

Currently, both the interpreter and the compiler are extremely optimistic about their inputs: in the future, we should add more safety checks. For example, we should check that we are calling the functions with the right arguments.

```
call :: PureExpr → [PureExpr] → FoFCode PureExpr
call funRef params =
    inject (CallDef Nothing funRef params return)
callN :: String → PureExpr → [PureExpr] → FoFCode PureExpr
callN varName funRef params =
    inject (CallDef (Just varName) funRef params return)
```

Finally, it is possible to return from a function thanks to the usual return. This should not be confused with the monadic return of Haskell.

```
returnc :: PureExpr → FoFCode PureExpr
returnc value = inject (Return value)
```
3.4.2 Compile Instantiation

Compiling functions is a little bit more tricky than usual. It requires generating or handling arguments, as well as handling the return value, if any. This corresponds to the following code.

```haskell
compileFunctions (NewDef attr nameF (Fun func) return args r)
  binding =
  (FNewDef attr nameF compBody return instanceArgs cont, binding2)
  where instanceArgs = instanciateArgs params
    (compBody, binding1) = compileSemtoFoF' (func instanceArgs) binding
    ref = CLRRef Global (TFun nameF (Fun func) return args) (Provided nameF)
    (cont, binding2) = r ref (binding1 |-> binding)
    instanciateArgs :: (TypeExpr, Maybe String) -> [PureExpr]
    instanciateArgs params = reverse $ foldl' instanciateArg [] $ zip [1..] params
    where instanciateArg l (idx, (typ, mName)) = (CLRRef Param typ name) : l
    where name = case mName of
      Just x -> Provided x
      Nothing -> makeVarName Param idx

compileFunctions (CallDef mName f@(CLRRef _ (TFun nameF func returnT argsT)) args r)
  binding =
  (FStatement (FCallDef name f args) cont, binding2)
  where (name, binding1) = case returnT of
    TVoid -> (Nothing, binding)
    _ -> case mName of
      Just x -> (Just $ Provided x, binding)
      Nothing -> (Just $ makeVarName Local loc, binding')
      where (loc, binding') = getFreshVar binding
  (cont, binding2) = case returnT of
    TVoid -> r Void binding1
    _ -> r (CLRRef Local returnT (fromJust name)) binding1

The translation of the return statement, on the other hand, is trivial.

```haskell
compileFunctions (Return e) binding =
  (FClosing $ FReturn e, binding)
```

3.4.3 Run Instantiation

As usual, we dispatch here:
runFunctions (NewDef f r) heap =
  uncurry r $ runNewDef f heap
runFunctions (CallDef a b r) heap =
  uncurry r $ runCallDef a b heap
runFunctions (Return a) heap =
  runReturn a heap -- OK??

And compute there:

runReturn :: PureExpr -> Heap -> (PureExpr, Heap)
runReturn e heap = (e, heap)
runNewDef :: Function -> Heap -> (PureExpr, Heap)
runNewDef function heap =
  (CLRef Global (TFun ⊥ function ⊥) ⊥), heap)
runCallDef :: PureExpr -> [PureExpr] -> Heap ->
  (PureExpr, Heap)
runCallDef (CLRef (TFun (Fun function) ⊥) args heap =
  let (result, heap1) = run (function args) heap in
  (result, heap1)

3.5 Reference Cells

The reference cell construct provides an abstraction to both variables and C pointers. It composed
by three combinators to create, read from, and write to reference cells. It can be compared to OCaml
references or Haskell IORef.

3.5.1 Smart Constructors

A reference cell is created in an initialized state. The variant newRefN allows the DSL designer to
provide a name to the created variable.

newRef :: Data -> FoFCode Loc
newRef d = inject (NewRef Nothing d return)
newRefN :: String -> Data -> FoFCode Loc
newRefN name d = inject (NewRef (Just name) d return)

Follow primitives to read from and write to these reference cells:

readRef :: Loc -> FoFCode Data
readRef l = inject (ReadRef l return)
writeRef :: Loc -> Data -> FoFCode PureExpr
writeRef l d = inject (WriteRef l d (return Void))

The current implementation lacks lots of sanity checks:

- read and Write on CLRef,
- write from and to compatible types,
- do not write local pointers into param/global ones,
- ...
3.5.2 Compile Instantiation

The compilation is tricky when it comes to computing the pointer type. I wouldn’t be surprised if some bugs were lying there. This concerns `newRef` and `readRef`, which effect on references is not trivial.

```haskell
compileReferences (NewRef refName ref r) binding =
    (FStatement (FNewRef publicName ref) cont, binding2)
    where (publicName, binding1) = case refName of
                Just x → (Provided x, binding)
                Nothing →
                    let (loc, binding1) = getFreshVar binding in
                        (makeVarName Local loc, binding1)
                        ret = CLRef Local (TPointer (typeOf ref) Avail) publicName
                        (cont, binding2) = r ret binding1

compileReferences (ReadRef ref @ (CLRef _ _ zloc) r) binding =
    (FStatement (FReadRef name ref) cont, binding2)
    where (loc, name, binding1) = heritVarName binding zloc
        ret = CLRef Local (unfoldPtrType ref) name
        (cont, binding2) = r ret binding1

writeRef is straightforward.

compileReferences (WriteRef ref d r) binding =
    (FStatement (FWriteRef ref d) cont, binding1)
    where (cont, binding1) = r binding
```

3.5.3 Run Instantiation

On the other hand, the implementation of the interpreter is much simpler. We start with the dispatcher:

```haskell
runReferences (NewRef _ d r) heap = uncurry r $ runNewRef d heap
runReferences (ReadRef _ r) heap = uncurry r $ runReadRef l heap
runReferences (WriteRef _ v r) heap = r $ runWriteRef l v heap

And the per-construct interpreters follow:

```haskell
runNewRef :: Data → Heap → (Loc, Heap)
runNewRef value heap =
    (CLRef Local typeOfVal name, heap2)
    where typeOfVal = typeOf value
        loc = freshLoc heap
        refs = refMap heap
        name = makeVarName Local loc
        heap1 = heap {freshLoc = loc + 1}
        heap2 = heap1 {refMap = (name, value) : refs}

runReadRef :: Loc → Heap → (Data, Heap)
runReadRef (CLRef _ _ location) heap =
    let refs = refMap heap in
    let val = fromJust $ location 'lookup' refs in
        (val, heap)

runWriteRef :: Loc → Data → Heap → Heap
```
runWriteRef (CLRef _ _ location) value heap =
  let refs = refMap heap in
  let refs1 = (location, value) : refs in
  heap { refMap = refs1 }

3.6 Strings

The String construct corresponds to static arrays of characters. However, they are implemented here as a special case as they are specially dealt with by the C compiler.

3.6.1 Smart Constructors

We only provide string creation combinators: accessing a string can be achieved thanks to Arrays combinators. As usual, we provide two combinators: one to create an anonymous string, one to create a named string.

$$\newString ::= \text{String} \to \text{FoFCode Loc}$$
$$\newString \text{ value} = \text{inject} (\text{NewString Nothing value return})$$
$$\newStringN ::= \text{String} \to \text{String} \to \text{FoFCode Loc}$$
$$\newStringN \text{ name value} = \text{inject} (\text{NewString (Just name) value return})$$

3.6.2 Compile Instantiation

The compilation is straightforward, on the model of static array declaration.

$$\text{compileString} (\text{NewString name dat r}) \text{ binding} =$$
$$\text{let (publicName, binding1)}$$
$$= \text{case name of}$$
$$\text{Just x \to (Provided x, binding)}$$
$$\text{Nothing \to}$$
$$\text{let (loc, binding1)} = \text{getFreshVar binding in}$$
$$\text{(makeVarName Global loc,}$$
$$\text{binding1) in}$$
$$\text{let ret = CLRef Global}$$
$$\text{(TArray (StaticArray \$ length dat) TChar) publicName in}$$
$$\text{let (cont, binding2)} = r \text{ ret binding1 in}$$
$$\text{(FStatement (FNewString publicName dat) cont,}$$
$$\text{binding2)}$$

3.6.3 Run Instantiation

Similarly, the interpreter is simple.

$$\text{runString} (\text{NewString a b r}) \text{ heap} = \text{uncurry r \$ runNewString b heap}$$

$$\text{runNewString} ::= \text{String} \to \text{Heap} \to (\text{Loc, Heap})$$
$$\text{runNewString string heap} =$$
let loc = freshALoc heap in
let size = length string in
let name = makeVarName Dynamic loc in
let ref = CLRef Dynamic (TArray (StaticArray size) TChar) name in
let heap1 = heap { freshALoc = loc + 1,
    arrayMap = (name, map cchar string) : (arrayMap heap) } in
(ref, heap1)

3.7 Structures Definition

The Structure construct allows you to mirror the struct data-type of C. It is composed by a newStruct combinator, to instantiate an element of this type, a readStruct combinator, to read a field from a structure, and a writeStruct combinator, to write into a field.

3.7.1 Smart Constructors

As often with instantiation operators, we can chose between statically or dynamically allocating the value. Then, it is possible to chose between an anonymous or a named value. All these choices are provided by the following four combinators.

newStaticStruct :: String →
    [((TypeExpr, String, Data))] →
    FoFCode Loc
newStaticStruct name stt =
    inject (NewStruct Nothing StaticStruct name
        (map (λ(t, n, v) → (n, (t, v))) stt)
        return)

newStaticStructN :: String →
    String →
    [((TypeExpr, String, Data))] →
    FoFCode Loc
newStaticStructN nameStr name stt =
    inject (NewStruct (Just nameStr) StaticStruct name
        (map (λ(t, n, v) → (n, (t, v))) stt)
        return)

newStruct :: String →
    [((TypeExpr, String, Data))] →
    FoFCode Loc
newStruct name stt =
    inject (NewStruct Nothing DynamicStruct name
        (map (λ(t, n, v) → (n, (t, v))) stt)
        return)

newStructN :: String →
    String →
    [((TypeExpr, String, Data))] →
    FoFCode Loc
newStructN nameStr name stt =
    inject (NewStruct (Just nameStr) DynamicStruct name
        (map (λ(t, n, v) → (n, (t, v))) stt)
        return)
Follow the read and write combinators:

\[
\text{readStruct :: Loc } \rightarrow \text{ String } \rightarrow \text{ FoFCode Data} \\
\text{readStruct } l f = \text{ inject (ReadStruct } l f \text{ return)} \\
\text{writeStruct :: Loc } \rightarrow \text{ String } \rightarrow \text{ Data } \rightarrow \text{ FoFCode ()} \\
\text{writeStruct } l f d = \text{ inject (WriteStruct } l f d \text{ (return ()))}
\]

### 3.7.2 Compile Instantiation

Apart from type handling, the compilation naturally follows the definition. As often, computing the \text{CLRef} is a magic voodoo, which is far from being provably correct.

\[
\text{compileStructures (NewStruct refName allocStruct name fields r)} \text{ binding } = \\
(\text{FStatement newS cont, binding2)} \\
\text{ where } (\text{loc, binding1) } = \text{getFreshVar binding} \\
\text{ structName } = \text{case refName of} \\
\text{ Just } x \rightarrow \text{Provided } x \\
\text{ Nothing } \rightarrow \text{makeVarName Dynamic loc} \\
\text{fieldsTypeStr } = \text{[(field, typ)} \\
\text{| (field, (typ, _)) } \leftarrow \text{fields]} \\
\text{typeStr } = \text{TStruct DynamicStruct name fieldsTypeStr} \\
\text{ret } = \text{CLRef Dynamic typeStr structName} \\
\text{(cont, binding2) } = \text{r ret binding1} \\
\text{newS } = \text{FNewStruct structName allocStruct name fields}
\]

\[
\text{compileStructures (ReadStruct ref @(CLRef origin typ @(TStruct alloc name fields)) zloc)} \\
\text{field r) binding } = \\
(\text{FStatement readS cont, binding2)} \\
\text{ where } (\text{loc, varName, binding1) } = \text{heritVarName binding zloc} \\
\text{typeField } = \text{fromJust } $ \text{field lookup fields} \\
\text{ret } = \text{CLRef (allocToOrigin alloc) (readOf typeField) varName} \\
\text{(cont, binding2) } = \text{r ret binding1} \\
\text{readS } = \text{FReadStruct varName ref field} \\
\text{allocToOrigin StaticStruct } = \text{Local} \\
\text{allocToOrigin DynamicStruct } = \text{Dynamic}
\]

\[
\text{compileStructures (WriteStruct ref @(CLRef origin typ @(TStruct alloc name fields)) zloc)} \\
\text{field value r) binding } = \\
(\text{FStatement writeS cont, binding1)} \\
\text{ where } (\text{cont, binding1) } = \text{r binding} \\
\text{writeS } = \text{FWriteStruct ref field value}
\]

### 3.7.3 Run Instantiation

The interpreter follows with a dispatcher:
runStructures \((\text{NewStruct } a b c r)\) heap =
uncurry r \$ runNewStruct a b c heap
runStructures \((\text{ReadStruct } a b r)\) heap =
uncurry r \$ runReadStruct a b heap
runStructures \((\text{WriteStruct } a b c r)\) heap =
r \$ runWriteStruct a b c heap

And the per-construct implementation:

runNewStruct :: AllocStruct \(ightarrow\) String \(ightarrow\) [(String, (TypeExpr, Data))] \(ightarrow\) Heap \(ightarrow\) (Loc, Heap)
runNewStruct alloc name struct heap =
let structT = map (\(x1, (x2, \_)) \rightarrow (x1, x2)) struct in
let structD = map (\(x1, (\_, x2)) \rightarrow (x1, x2)) struct in
let loc = freshLoc heap in
let structs = strMap heap in
let varName = makeVarName Local loc in
let heap1 = heap \{ freshLoc = loc + 1 \} in
let heap2 = heap1 \{ strMap = \(\text{varName}, \text{structD} : \text{structs} \} in
(CLRef Local (TStruct alloc name structT) varName, heap2)

runReadStruct :: Loc \(ightarrow\) String \(ightarrow\) Heap \(ightarrow\) (Data, Heap)
runReadStruct (CLRef _ location) field heap =
let structs = strMap heap in
let struct = fromJust $ location 'lookup' structs in
let val = fromJust $ field 'lookup' struct in
(val, heap)

runWriteStruct :: Loc \(ightarrow\) String \(ightarrow\) Data \(ightarrow\) Heap \(ightarrow\) Heap
runWriteStruct (CLRef _ location) field value heap =
let structs = strMap heap in
let struct = fromJust $ location 'lookup' structs in
let struct1 = (field, value) : struct in
let structs1 = (location, struct1) : structs in
heap \{ strMap = structs1 \}

3.8 Type Definition

The \textit{Typedef} construct provides a similar service than the C \texttt{typedef}.

3.8.1 Smart Constructors

In particular, \textit{Typedef} offers two combinators. The first one, \texttt{alias} allows you to locally define a type alias.

\texttt{alias} :: TypeExpr \rightarrow FooCode PureExpr
\texttt{alias typedef} = inject (Typedef typedef (return void))

The other one, \texttt{aliasE} allows you to mention an aliasing declared in an external library, such as \texttt{<stdbool.h>} that declares a \texttt{bool} as an integer.

\texttt{aliasE} :: String \rightarrow
TypeExpr \rightarrow
3.8.2 Compile Instantiation

The compilation to FoF is straightforward:

```plaintext
compileTypedef (Typedef (TTypedef typ aliasName) r) binding =
let (cont, binding1) = r binding in
(FStatement (FTypedef typ aliasName) cont,
 binding1)
```

```plaintext
compileTypedef (TypedefE inclDirective typeDef (TTypedef typ aliasName) r) binding =
let (cont, binding1) = r binding in
(FStatement (FTypedefE inclDirective typeDef) cont,
 binding1)
```

3.8.3 Run Instantiation

These operations occurring at the type-level, the interpreter doesn’t pay any attention to them:

```plaintext
runTypedef (Typedef r) heap = r heap
runTypedef (TypedefE r) heap = r heap
```

3.9 Unions Definition

The `Union` constructs abstracts the union data-type of C.

3.9.1 Smart Constructors

Hence, creating an union is available in four flavors, statically or dynamically allocated, and anonymous or named.

```plaintext
newStaticUnion :: String →
[(TypeExpr, String)] →
String →
Data →
FoFCode Loc
newStaticUnion name fields field dat =
inject (NewUnion Nothing StaticUnion name
(mapStateToProps (λ(s1, s2) → (s2, s1)) fields)
(field, dat)
return)
newStaticUnionN :: String →
String →
[(TypeExpr, String)] →
String →
Data →
FoFCode Loc
```
newStaticUnionN nameU name fields field dat =
  inject (NewUnion (Just nameU) StaticUnion name
  (map (λ(s1, s2) → (s2, s1)) fields)
  (field, dat)
  return)

newUnion :: String →
  [(TypeExpr, String)] →
  String →
  Data →
  FoFCode Loc
newUnion name fields field dat =
  inject (NewUnion Nothing DynamicUnion name
  (map (λ(s1, s2) → (s2, s1)) fields)
  (field, dat)
  return)

newUnionN :: String →
  String →
  [(TypeExpr, String)] →
  String →
  Data →
  FoFCode Loc
newUnionN nameU name fields field dat =
  inject (NewUnion (Just nameU) DynamicUnion name
  (map (λ(s1, s2) → (s2, s1)) fields)
  (field, dat)
  return)

Reading and writing follow the usual scheme:

readUnion :: Loc → String → FoFCode Data
readUnion l f = inject (ReadUnion l f return)

writeUnion :: Loc → String → Data → FoFCode ()
writeUnion l f d = inject (WriteUnion l f d (return ()))

3.9.2 Compile Instantiation

As usual the difficulty of the compilation stands in not messing up created and read types. Apart from that, it is a simple translation.

\[
\text{compileUnions} \left( \text{NewUnion refName allocUnion nameU fields (initField, initData) } r \right) \text{ binding} =
\left( \text{FStatement newU cont, binding2} \right)
\text{where typeUnion} = \text{TUnion DynamicUnion nameU fields}
\left( \text{loc, binding1} \right) = \text{getFreshVar binding}
\text{name} = \text{case refName of}
\text{Nothing} → \text{makeVarName Dynamic loc}
\text{Just x} → \text{Provided x}
\text{ret} = \text{CLRef Dynamic typeUnion name}
\left( \text{cont, binding2} \right) = r \text{ ret binding1}
\text{newU} = \text{FNewUnion name allocUnion nameU fields (initField, initData)}
\text{compileUnions} \left( \text{ReadUnion ref @(CLRef _ typeU @(TUnion alloc}
\right)
\]
nameU
fields) xloc)
field r) binding =
(FStatement readU cont,
  binding2)
  where (loc, name, binding1) = heritVarName binding xloc
    typeField = fromJust $ field 'lookup' fields
    origin = allocToOrigin alloc
    ret = CLRef origin (readOf typeField) name
    (cont, binding2) = r ret binding1
    readU = FReadUnion name ref field
    allocToOrigin StaticUnion = Local
    allocToOrigin DynamicUnion = Dynamic
compileUnions (WriteUnion ref @ (CLRef origin
  typ @(TUnion alloc _ fields)
  xloc)
  field
  value r) binding =
  (FStatement writeU cont,
    binding1)
  where (cont, binding1) = r binding
    writeU = FWriteUnion ref field value

### 3.9.3 Run Instantiation

This part has not been implemented yet. Hence, the interpreter will blow up in presence of unions. To get an idea of the desired implementation, take a look at the reference cells interpreter. It should be similarly easy.

```
runUnions (NewUnion - a b c d r) heap = error "runUnions: not yet implemented"
runUnions (ReadUnion a b r) heap = error "runUnions: not yet implemented"
runUnions (WriteUnion a b c r) heap = error "runUnions: not yet implemented"
```
Chapter 4

Lib-C Operators

4.1 Printf

The Printf constructs is a simple foreign function wrapper around the C library printf.

4.1.1 Smart Constructors

Provided with a format string and a list of parameters, the printf Pcombinator emulates printf.

\[
\text{printf :: String} \rightarrow [\text{PureExpr}] \rightarrow \text{FoFCode PureExpr}
\]
\[
\text{printf format params} = \text{inject (Printf format params (return Void))}
\]

4.1.2 Compile Instantiation

Compilation is a natural foreign function call. Note the quoting of format: we sacrify the semantics of the format string. We could possibly apply some tricks to recover it, or to get it in a "nice" format thanks to the printf combinator. However, for simplicity, we drop its semantics for now.

\[
\text{compilePrintf (Printf format params r) binding =}
\text{let (cont, binding1) = r binding in}
\text{(FStatement (FFICall "printf" ((quote format) : params)) cont, binding1)}
\]
4.1.3 Run Instantiation

For the reason mentioned above, it is a pain to recover the semantics of the printf. Hence, we drop its side-effect when interpreting it.

\[
\text{runPrintf (Printf } a \ b \ r) \ heap = r \ heap
\]

An esthetically satisfying solution would be to store this (and others) side-effecting operations in a stream, along with its arguments. Hence, we could compare side-effecting programs by their so-called trace. By ignoring the effect of printf here, we consider that side-effects have no semantic significance. This is kind of lie when interpreting an imperative language.

4.2 Assert

The construct Assert embeds the C assert function into FoF.

4.2.1 Smart Constructors

The use of assert is obvious, by its definition.

\[
\text{assert :: PureExpr } \rightarrow \text{ FoFCode PureExpr}
\]

\[
\text{assert test} = \text{inject (Assert test (return Void))}
\]

4.2.2 Compile Instantiation

The compilation is a direct translation into a foreign function:

\[
\text{compileAssert (Assert } e \ r) \ binding =
\]

\[
\text{let (cont, binding1) } = r \ binding \ \text{in}
\]

\[
(\text{FStatement (FFFICall "assert" [e]) cont, binding1})
\]

4.2.3 Run Instantiation

As mentioned with Printf, we take here the easy option of ignoring the run-time behaviour of an assertion.

\[
\text{runAssert (Assert } a \ r) \ heap = r \ heap
\]

Being able to capture the semantics of that operation would be helpful when debugging a compiler. So, some efforts are worth being devoted here.
Chapter 5

Lib-barrefish Operators

Here may be found the last words of Joseph of Aramathea. He who is valiant and pure of spirit may find the Holy Grail in the Castle of uuggggggh

Monty Python

5.1 Has Descendants

The construct HasDescendants embeds the libarrefish function has_descendants into FoF.

5.1.1 Smart Constructors

This function is provided in two flavors: an anonymous one, which stores its result in an anonymous variable, and a named one, which allows you to name the resulting variable.

```plaintext
has_descendants :: PureExpr → FoFCode PureExpr
has_descendants cte = inject (HasDescendants Nothing cte return)
has_descendantsN :: String → PureExpr → FoFCode PureExpr
has_descendantsN name cte = inject (HasDescendants (Just name) cte return)
```

5.1.2 Compile Instanciation

This function is translated into a foreign function definition, as usual:

```plaintext
compileHasDescendants (HasDescendants mName arg r) binding =
  let (loc, binding1) = getFreshVar binding in
  let name = case mName of
    Nothing → makeVarName Local loc
    Just x → Provided x in
  let ref = CLRef Local uint64T name in
  let (cont, binding2) = r ref binding1 in
  (FStatement (FFFICall "has_descendants" [ref, arg]) cont, binding2)
```
5.1.3 Run Instantiation

As for libc functions, we have not yet implemented the semantics of that operation. A trace-based semantics would make sense, too.

\[
\text{runHasDescendants} (\text{HasDescendants} \_ a r) \text{ heap} = \text{error} \ "\text{HasDescendants: eval not implemented}"
\]

5.2 Mem To Phys

This construct embeds the libarrellfish function mem_to_phys into FoF.

5.2.1 Smart Constructors

As for HasDescendants, both named and anonymous function are provided. They are direct wrappers around the mem_to_phys function.

\[
\begin{align*}
\text{mem_to_phys} & : \text{PureExpr} \to \text{FoFCode PureExpr} \\
\text{mem_to_phys} \text{ cte} & = \text{inject (MemToPhys Nothing cte return)} \\
\text{mem_to_physN} & : \text{String} \to \text{PureExpr} \to \text{FoFCode PureExpr} \\
\text{mem_to_physN} \text{ name cte} & = \text{inject (MemToPhys (Just name) cte return)}
\end{align*}
\]

5.2.2 Compile Instantiation

Compiling is straightforward: just declare a foreign function.

\[
\begin{align*}
\text{compileMemToPhys} (\text{MemToPhys mName arg r}) \text{ binding} = \\
\text{let loc, binding1} = \text{getFreshVar binding in} \\
\text{let name} = \text{case mName of} \\
\quad \text{Just x} \to \text{Provided x} \\
\quad \text{Nothing} \to \text{makeVarName Local loc in} \\
\text{let ref} = \text{CLRef Local uint64T name in} \\
\text{let cont, binding2} = r \text{ ref binding1 in} \\
\text{(FStatement (FFFICall "mem_to_phys" [ref, arg]) cont, binding2)
\end{align*}
\]

5.2.3 Run Instantiation

However, the semantics remains to be defined.

\[
\text{runMemToPhys} (\text{MemToPhys} \_ a r) \text{ heap} = \text{error} \ "\text{MemToPhys: eval not implemented}"
\]
Part II

The Filet-o-Fish Compiler
The Filet-O-Fish Compiler(s)

I’m French!
Why do think I have this outrageous accent,
you silly king-a?!

Monty Python

The Filet-o-Fish to C compiler is major component of Filet-o-Fish. Major in the sense that it is a big chunk of code, which correctness is critical. So, when playing with this part of the code, better be cautious. The high-level specification of the compiler is straightforward: given a Filet-o-Fish code, it should translate it into a semantically equivalent C code. Well, it is a compiler, after all.

However, from a usability point of view, it is vital to be able to understand what the generated code is doing: think of a debugging session that needs to go through some code generated by Filet-o-Fish. Hence, we have implemented some so-called optimizations that tidy up the generated code. In order to ease the implementation of these optimizations we rely on two standard compiler techniques: first, we define a bunch of intermediate languages (IL) to tackle a specific optimization issue, second we implement the optimizer as a data-flow analysis solver. The current state of affair is not as idyllic and the reader is referred to Chapter A to get an overview of my dreams.

Let us sketch the compilation process.

```plaintext
compile :: Semantics FoFConst PureExpr → PakaCode
compile sem =
  optimizePaka $!
  compileFoFtoPaka $!
  compileSemtoFoF sem
```

First of all, The compiler is provided a value of type `Semantics FoFConst PureExpr`, built by the operators of Chapter 3. While this structure has a nice functional definition, making it convenient for interpretation, it is bothersome to navigate on it. Therefore, the first pass of the compiler is to reify this data-structure, as explained in Chapter 6.

At the end of this compilation pass, the initial input has been translated into an (hopefully) equivalent one in the FoF intermediate language. In order to remove unnecessary variable assignments, a second pass of the compiler translate the FoF code into Paka code. In a nutshell, the Paka language only captures variable assignments, ignoring the computational parts of statements. Hence, seeking and simplifying redundant assignments is made easy: it corresponds to an optimization phase applied to the resulting Paka code.

Because different optimizations will focus on different aspects of the code, one could imagine several intermediate languages and refinements between them. FoF and Paka are just an example of what could be done. The name Paka comes from a retired hurricane: to pursue that tradition, you can look up the list of retired hurricane names [1]. There is fair amount of ILs to be implemented.
Chapter 6

The FoF Intermediate Language

6.1 The FoF Intermediate Language

The FoF IL is nothing more than a direct translation of the Filet-o-Fish operators. In retrospect, calling it FoF might be confusing, Never forget that lives in the IL/ directory, so it is simply not the abbreviation for Filet-o-Fish, and that’s it.

Having said that, it is also obvious that, essentially, FoF is Filet-o-Fish: it is a dumb translation of the Filet-o-Fish constructs into a data-type. Hence, an ILFoF term is the reification of the language constructs:

```data ILFoF
  = FConstant PureExpr
   | FStatement FStatement ILFoF
   | FClosing FClosing
   | FNewDef [FunAttr] String ILFoF TypeExpr [PureExpr] ILFoF
   | IF IFoF ILFoF ILFoF ILFoF ILFoF
   | FFor ILFoF ILFoF ILFoF ILFoF ILFoF
   | FWhile ILFoF ILFoF ILFoF
   | FDoWhile ILFoF ILFoF ILFoF
   | FSwitch PureExpr [(PureExpr, ILFoF)] ILFoF ILFoF
```

Where an FStatement is one of the sequential statement of the Filet-o-Fish language, that is:

- [...] For, since the tragic death of her father –
- He’s not quite dead!
- Since the near fatal wounding of her father –
- He’s getting better!
- For, since her own father… who, when he
  seemed about to recover, suddenly felt the
  icy hand of death upon him,…
- Oh, he’s died!
- And I want his only daughter to look upon
  me… as her own dad – in a very real, and
  legally binding sense. And I feel sure that the
  merger – uh, the union – between the
  Princess and the brave, but dangerous, Sir
  Launcelot of Camelot...

Monty Python
data FStatement
  = FNewUnion VarName AllocUnion String [(String, TypeExpr)] (String, Data)
  | FReadUnion VarName Loc String
  | FWriteUnion VarName Loc String Data
  | FTypedef TypeExpr String
  | FTypedefE String TypeExpr
  | FNewStruct VarName AllocStruct String [(String, (TypeExpr, Data))] (String, Data)
  | FReadStruct VarName Loc String
  | FWriteStruct Loc String Data
  | FNewString VarName String
  | FReadRef VarName Data
  | FWriteRef VarName Loc
  | FNewRef VarName Data
  | FCallDef (Maybe VarName) PureExpr [PureExpr]
  | FFFICall String [PureExpr]

And an FClosing is a standard C end of something statement:

data FClosing
  = FReturn PureExpr
  | FBreak
  | FContinue

6.2 Translating FoFCode to IL.FoF

6.2.1 The compiler

We already know how to translate individual statements of the FoFCode language, by using the one
step compiler compileAlgebra defined in ./Expressions.lhs and provided a Binding capturing the
state of the compiler. The game is then to chain up these compilation steps into a single one. Here,
foldSemantics nicely comes to the rescue and automatically build this compiler.

\[
\text{compileSemtoFoF'} :: \text{FoFCode PureExpr} \rightarrow \text{Binding} \rightarrow (\text{ILFoF, Binding})
\]

\[
\text{compileSemtoFoF'} = \text{foldSemantics compilePure compileAlgebra}
\]

Where compilePure is used to compile pure expressions. Pure expressions are, by definition, constants
and returned as such. This is used when generating tests for conditional expressions: the computational
part is generated above the test handler and only the (pure) result is tested.

\[
\text{compilePure :: PureExpr} \rightarrow \text{Binding} \rightarrow (\text{ILFoF, Binding})
\]

\[
\text{compilePure } x \text{ binding} = (FConstant } x, \text{ binding}
\]

For our convenience, we can define the following compileSemToFoF function that takes a closed FoFCode
and compiles it in the empty environment: that’s our compiler for self-contained expressions.

\[
\text{compileSemtoFoF :: FoFCode PureExpr} \rightarrow \text{ILFoF}
\]

\[
\text{compileSemtoFoF } \text{term} = \text{fst }$\text{ compileSemtoFoF'} \text{term emptyBinding}
\]

\[
\text{where emptyBinding} = \text{Binding } \{\text{freshVar} = 1,
\text{defStructs} = []},
\text{defUnions} = []},
\text{defEnums} = []\}
\]
6.2.2 The machinery

Manipulating the compiler environment

We very often need to generate fresh names, while keeping the freshness invariant of the compiler environment. The following function just does that:

\[
\text{getFreshVar} :: \text{Binding} \rightarrow (\text{Int}, \text{Binding})
\]

\[
\text{getFreshVar} \text{ binding} = (\text{loc}, \text{binding}1)
\]

where \( \text{loc} = \text{freshVar} \text{ binding} \)  
\( \text{binding}1 = \text{binding} \{ \text{freshVar} = \text{loc} + 1 \} \)

Note that a clever implementation would be something of type:

\[
\text{better_getFreshVar} :: \text{Binding} \rightarrow (\text{Int} \rightarrow \text{Binding} \rightarrow t) \rightarrow t
\]

\[
\text{better_getFreshVar} \text{ binding} \ f = \bot
\]

Which enforces the fact that the function \( f \) is provided a synchronized compiler state. This ensures that people don’t inadvertently mess up the compiler state. This remark holds for too many functions below, I’m a bit sad about that.

In order to ensure the freshness of names across bindings, we define the following \( \text{passFreshVar} \) function that builds a \( \text{stableBinding} \) whose fresh variables are ensured not to clash with the one generated using \( \text{upBinding} \). Similarly, it carries the structures defined in \( \text{upBinding} \).

\[
\text{passFreshVar} :: \text{Binding} \rightarrow \text{Binding} \rightarrow \text{Binding}
\]

\[
\text{passFreshVar} \text{ upBinding} \text{ stableBinding} =
\]

\[
\text{stableBinding} \{ \text{freshVar} = \text{freshVar} \text{ upBinding},
\]

\[
\text{defStructs} = \text{defStructs} \text{ upBinding},
\]

\[
\text{defUnions} = \text{defUnions} \text{ upBinding},
\]

\[
\text{defEnums} = \text{def Enums} \text{ upBinding}
\]

\[
(|\rightarrow>) = \text{passFreshVar}
\]

From variable identifier and an origin, we can later make \( \text{VarName} \). In a craze of Hungarian naming, the origin dictates the name of variables.

\[
\text{makeVarName} :: \text{Origin} \rightarrow \text{Int} \rightarrow \text{VarName}
\]

\[
\text{makeVarName} \text{ orig loc} = \text{Generated} \{ \text{makeVarName'} orig loc
\]

where \( \text{makeVarName'} \) Local \( x = "\text{fof}_x" \# \text{show} \ x \)  
\( \text{makeVarName'} \) Param \( x = "\text{fof}_y" \# \text{show} \ x \)  
\( \text{makeVarName'} \) Dynamic \( x = "\text{fof}_d" \# \text{show} \ x \)  
\( \text{makeVarName'} \) Global \( x = "\text{fof}_g" \# \text{show} \ x \)

The Hungarian fever can go further: when a variable is somehow related to another \( \text{VarName} \), the \( \text{heritVarName} \) makes it explicit at the name level by deriving a fresh name from the previous one.

\[
\text{heritVarName} :: \text{Binding} \rightarrow \text{VarName} \rightarrow (\text{Int}, \text{VarName}, \text{Binding})
\]

\[
\text{heritVarName} \text{ binding name} = (\text{loc}, \text{Inherited loc name}, \text{binding}1)
\]

where \( (\text{loc}, \text{binding}1) = \text{getFreshVar} \text{ binding} \)

From Expressions to Types

Let us be honest: the code which follows is tricky. Change something there and the generated code will be wrong, if it is not already. I’m looking at you \text{readOf} \ and \text{liftType}. They came to life during the implementation of References and its painful compiler. After a lot of work, I came to the conclusion (and
proof) that they are correct. The question is now: are they correct when mixed with complex data-types, such as structs and arrays. The practician seems to say “yes”, the theoretician remains proofless.

The intrinsic difficulty is that a Reference abstracts both a C variable and a C pointer. However, in C, both concepts are quite distinct. Hence, the compiler needs to be clever to translate the unified notion of Reference in two semantically different objects. Hence that horrible machinery.

typeOf: Obviously, there exists a map going from each well-typed element of PureExpr to an element of TypeExpr. Hence, this map assigns a type to a given, well-typed expression. As for ill-typed expressions, we simply return an error message.

Computing the type of base values as well as of unary operations is straightforward:

```
typeOf :: PureExpr → TypeExpr
typeOf (Void) = TVoid

typeOf (CLInteger sign size _) = TInt sign size

typeOf (CLFloat _) = TFloat

typeOf (CLRef _ typ _) = typ

typeOf (Unary _ x) = typeOf x
```

A binary operation is well-typed if and only if both sub-terms are well-typed and of same type. The same goes for the branches of a conditional expression:

```
typeOf (Binary _ x y) =
  if (typeOfx ≡ typeOfy) then
typeOfx
  else error "typeOf: Binop on distinct types."
  where typeOfx = typeOf x
  typeOfy = typeOf y

typeOf (Test _ t1 t2) =
  if (typeOft1 ≡ typeOft2) then
  typeOft1
  else error "typeOf: Test exits on distinct types"
  where typeOft1 = typeOf t1
  typeOft2 = typeOf t2
```

By convention, the value returned by sizeof is an unsigned 64 bits integer:

```
typeOf (Sizeof _) = TInt Unsigned TInt64
```

Finally, the type of a casted expression is the assigned type. Note that we do not judge of the legality of this cast here. This aspect is handled by the dynamic semantics of FoF’s meta-language.

```
typeOf (Cast t _) = t
```

readOf and unfoldPtrType: When we read the content of the reference cell, of type TPointer typeCell modeCell, the type of the object read is either:

- A constant of type typeCell, or
- A reference cell of type typeCell, in a Read mode

We can distinguish both cases thanks to typeCell. If typeCell is a TPointer itself (first case, below), this means that we are dealing with a reference cell. If typeCell is a base type (second case), this means that this is a constant.
readOf :: TypeExpr → TypeExpr
readOf (TPointer typ _) = TPointer typ Read
readOf x = x

unfoldPtrType :: PureExpr → TypeExpr
unfoldPtrType (CLRef _ (TPointer typ _)) = readOf typ

liftType :: TypeExpr → TypeExpr
liftType (TPointer x) = x

deref :: PureExpr → String
deref (CLRef _ (TPointer _ _)) = "&"
deref _ = ""

deref :: PureExpr → String
deref (CLRef _ (TPointer _ _)) = "&"
deref _ = ""

6.3 Evaluator

Just as for the compiler, described in the previous section, the implementation of the Filet-o-Fish interpreter is automatically derived from the one-step interpreters. Again, foldSemantics comes to the rescue and computes the interpreter:

run :: Semantics FoFConst PureExpr → Heap → (PureExpr, Heap)
run = foldSemantics (.) runAlgebra
Chapter 7

The Paka Intermediate Language

Listen, lad.
I’ve built this kingdom up from nothing.
When I started here, all there was was swamp. All the kings said I was daft to build a castle in a swamp, but I built it all the same, just to show ’em. It sank into the swamp.
So, I built a second one. That sank into the swamp.
So I built a third one. That burned down, fell over, then sank into the swamp.
But the fourth one stayed up. An’ that’s what your gonna get, lad – the strongest castle in these islands.

Monty Python

7.1 The Paka Intermediate Language

The purpose of Paka is to ease the task of tracking down unnecessary variable assignment in the to-be-generated C code. Therefore, its syntax is extremely close to C and focused on intra-procedural statements. This is reflected by the definition of PakaCode: the structure of the C file is almost here, with includes, type definitions and prototypes, function prototypes, and function definitions, in this order.

Note that they are all defined by a Map or associative list from String to something else. The String plays the role of an identifier which should be compiled only once in the C code. Typically, a type definition should appear only once, otherwise the C compiler will complain. Map is used when the definition order is not important, associative list is used when we want to keep it (when a declaration might be defined in term of another declaration defined earlier).

```plaintext
data PakaCode
    = PakaCode { includes :: Map.Map String Doc,
                   types :: Map.Map String Doc,
                   declarations :: [(String, Doc)],
                   prototypes :: Map.Map String Doc,
                   globalVars :: [(String, Doc)],
                   functions :: Map.Map String (Doc, Doc, String, Doc, PakaIntra, ILPaka) }

emptyCode = PakaCode { includes = Map.empty,
```
Each function is defined by a `PakaIntra` record, which stands for *intra-procedural*. In there, we find local variable definitions and, potentially, a constant. This constant is used to carry the result of a side-effecting test: the side-effecting is compiled before the test-handler and the constant is tested instead.

```haskell
data PakaIntra
  = PakaIntra { localVars :: Map.Map String Doc,
               expr :: (Maybe PureExpr) }
  deriving Show
emptyIntra = PakaIntra { localVars = Map.empty,
                         expr = Nothing }
```

As part of the definition of functions, we find the body of the function. This is presented as an `ILPaka` data-type. This is a strip-down version of the `FoF` IL: we have kept most of the control-flow structures (at the exception of the `for` loop, translated into `while` loops) and statements. Because we are describing intra-procedural code, we have removed the function definition construct.

```haskell
data ILPaka
  = PVoid
  | PClosing PakaClosing
  | PStatement PakaStatement ILPaka
  | PIf ILPaka PureExpr ILPaka ILPaka ILPaka
  | PWhile ILPaka PureExpr ILPaka ILPaka ILPaka
  | PDoWhile ILPaka ILPaka PureExpr ILPaka
  | PSwitch PureExpr [(PureExpr, ILPaka)] ILPaka ILPaka
```

However, the major specificity of Paka is its definition of a statement: a statement is either an assignment or an instruction. An assignment `PAssign x t ys` is a term `t` in which the variable `x` is assigned a value computed from the variables `ys`. On the other hand, an instruction `PInstruction t ys` is a side-effecting operation `t` making use of the variables `ys`.

In a nutshell, when chasing redundant assignments, we will track down raw assignment `PAssign x t [y]`, remove the assignment, and replace all use of `x` by `y`.

```haskell
data PakaStatement
  = PAssign PakaVarName Term [PakaVarName]
  | PInstruction Term [PakaVarName]
```

A `Term` is an almost valid C statement, with holes in it. The holes correspond to the variable names: provided with the list of variable names, it computes a C statement.

Hence, when we have settled the input and output variables of a `PAssign x t ys`, we obtain the corresponding C statement by applying `t x : xs`. Similarly, we get the C code from an instruction `PInstruction t ys` by computing `t ys`.

```haskell
type Term = [Doc] → Doc
```

However, things are not that simple. First, we need more information about the variable: are they raw C variables, or pointers, or dereferenced from somewhere else? This information is vital to avoid aliasing issues.

Similarly, when a variable `y` is used in some operationally non-trivial term `t`, we cannot simply replace `x` by `y`: we would have to compute some sort of `t y` to be correct. Although it would be doable, we do not support that at the moment and tag the variable name as `Complex`, meaning “non prone to simplification”.

---

Barrelfish TN-024
Finally, constants are a gold opportunity we don’t want to miss, hence we explicitly carry the value instead of variable name. Therefore, we are able to do some naive constant propagation for free.

```haskell
data PakaVarName
    = Var String
    | Ptr PakaVarName
    | Deref PakaVarName
    | Complex PakaVarName
    | K PureExpr
    deriving (Show, Eq)

data PakaClosing
    = PReturn PureExpr
    | PBreak
    | PContinue
    deriving Show
```

## 7.2 Paka building blocks

I’m particularly proud of the Paka code generation architecture. To build a Paka term, we simply call some builders functions which are chained up together with the `#` operator. These builders take care of inserting the definitions in the right place in `PakaCode`, `PakaIntra`, or sequentially extend the `ILPaka` code. Thanks to that machinery, we don’t have to explicitly build these data-structures, we just call functions.

Hence, a builder is just putting a brick in the `PakaBuilding` wall:

```haskell
type PakaBuilding = (ILPaka -> ILPaka, PakaCode, PakaIntra)
```

That is, operations taking some arguments and extending a `PakaBuilding` into a new one.

### 7.2.1 Low-level machinery

To give a feeling of “sequential code”, the `#` operator is simply an inversed composition operation:

```
f \# g = \lambda x \rightarrow g (f x)
```

Using `#`, we will compose our builders with a sequential feeling.

Because most, if not all, operations modify one element of the `PakaBuilding` triple, we define the following combinators:

- `first :: (a -> b) -> (a, c, d) -> (b, c, d)`
- `first f (a, b, c) = (f a, b, c)`
- `second :: (a -> b) -> (c, a, d) -> (c, b, d)`
- `second f (a, b, c) = (a, f b, c)`
- `third :: (a -> b) -> (c, d, a) -> (c, d, b)`
- `third f (a, b, c) = (a, b, f c)`
7.2.2 Building PakaCode

We can add new C includes:

```plaintext
class String → PakaBuilding → PakaBuilding
class id = second $ include' id

where include' id globalEnv =
  case id 'Map.lookup' incls of
    Nothing → globalEnv { incls = Map.insert id decl incls }
    Just _ → globalEnv
where decls = incls globalEnv
    decl = text "#include" < + > text id
```

We can declare new C types:

```plaintext
class String → Doc → PakaBuilding → PakaBuilding
declare id typ decl = second $ declare' id typ decl

where declare' id typ decl globalEnv =
  case id 'Map.lookup' typings of
    Nothing → globalEnv { declarations = (id, decl) : decls,
                        types = Map.insert id typ types }
    Just _ → globalEnv
where decls = declarations globalEnv
    types = types globalEnv
```

We can declare global variables:

```plaintext
class String → Doc → PakaBuilding → PakaBuilding
globalVar id def = second $ globalVar' id def

where globalVar' id def globalEnv =
  case id 'lookup' vars of
    Nothing → globalEnv { vars = (id, def) : vars }
    Just _ → globalEnv
where vars = globalVars globalEnv
```

We can add function prototypes:

```plaintext
class String → Doc → PakaBuilding → PakaBuilding
prototype id proto = second $ prototype' id proto

where prototype' id proto globalEnv =
  case id 'lookup' protos of
    Nothing → globalEnv { protos = Map.insert id proto protos }
    Just _ → globalEnv
where protos = prototypes globalEnv
```

And we can define new functions:

```plaintext
class Doc → Doc → String → Doc → PakaIntra → ILPaka → PakaBuilding → PakaBuilding
function returnT attrs funName funArgs lEnv body =
  second $ function' returnT attrs funName funArgs lEnv body

where function' returnT attrs funName funArgs lEnv body gEnv =
  case funName 'Map.lookup' functions' of
    Nothing → gEnv { functions = Map.insert funName (returnT, attrs, funName, funArgs, lEnv, body) functions }
    Just _ → gEnv
where functions' = functions gEnv
```
7.2.3 Building PakaIntra

As for global variables in the PakaCode, we can add local variables in the PakaIntra environment:

```haskell
localVar :: String → Doc → PakaBuilding → PakaBuilding
localVar id def = third $ localVar' id def
  where localVar' id def localEnv
    = case id `Map.lookup` vars of
        Nothing → localEnv (localVars = Map.insert id def vars)
        Just _ → localEnv
    where vars = localVars localEnv
```

And we can bring a constant in the PakaIntra:

```haskell
constant :: PureExpr → PakaBuilding → PakaBuilding
constant e = third $ constant' e
  where constant' e lEnv = lEnv { expr = Just e }
```

7.2.4 Building ILPaka

Obviously, the serious stuff happens in ILPaka, or more precisely ILPaka → ILPaka: this code is seriously continuation-passing. The plan is that we want to build a ILPaka value. However, we note that, for instance, to build a PStatement value, we need to know the remaining code. But we don’t know it yet, as we are compiling it! So, we return a continuation that waits for that uncompiled chunk and plug it in the right place. Continuation-passing style, yay!

As an example of that technique in action, take a look at instr and assgn below. Apart from that CPS detail, they are computationally trivial, bringing their arguments in the right place of the constructor and returning by calling the continuation.

```haskell
instr :: Term → [PakaVarName] → PakaBuilding → PakaBuilding
instr instruction vars = first $ instr' instruction vars
  where instr' instruction varNames k
    = λc →
      k $ PStatement (PInstruction instruction varNames) c
assgn :: PakaVarName → Term → [PakaVarName] → PakaBuilding → PakaBuilding
assgn wVarName assgnmt rVarNames = first $ assgn' wVarName assgnmt rVarNames
  where assgn' wVarName assgnmt rVarNames k
    = λc →
      k $ PStatement (PAssign wVarName assgnmt rVarNames) c
```

As you can expect, we need to stop “continuating” at some point. This naturally fits with the role of closing terms:

```haskell
close :: PakaClosing → PakaBuilding → PakaBuilding
close c = first $ close' c
  where close' c = λk _ → k (PClosing c)
```

Similarly, the control-flow operators closes all their branches and only continue downward:

```haskell
pif :: ILPaka → PureExpr → ILPaka → ILPaka → PakaBuilding → PakaBuilding
pif cond test ifTrue ifFalse = first $ pif' cond test ifTrue ifFalse
  where pif' cond test ifTrue ifFalse cont = λc →
      cont $ PIf cond test ifTrue ifFalse c
pwhile :: ILPaka → PureExpr → ILPaka → PakaBuilding → PakaBuilding
pwhile cond test loop = first $ pwhile' cond test loop
```

FileOfFish - 60

Barrelyfish TN-024
where pwhile' cond test loop cont = \lambda c \rightarrow
cost PWhile cond test loop c
pdoWhile :: ILPaka → ILPaka → PureExpr → PakaBuilding → PakaBuilding
pdoWhile loop cond test = first $ pdoWhile' loop cond test
where pdoWhile' loop cond test cont = \lambda c \rightarrow
cost PDOWhile loop cond test c
pswitch :: PureExpr → [(PureExpr, ILPaka)] → ILPaka → PakaBuilding → PakaBuilding
pswitch test cases defaultCase = first $ pswitch' test cases defaultCase
where pswitch' test cases defaultCase cont = \lambda c \rightarrow
cost PSwitch test cases defaultCase c

7.3 Translating IL.FoF to IL.Paka

To translate IL.FoF code, we simply iterate over it and build the corresponding IL.Paka term.

compileFoFtoPaka :: ILFoF → PakaCode
compileFoFtoPaka code = ccode
where (_, ccode, _) = compileFoFtoPaka' code (id, emptyCode, emptyIntra)

The translation is often trivial, because both languages are very similar in structure. The major novelty is that intra-procedural and extra-procedural code are translated into different data-structures: building an ILPaka term for the former, defining a PakaCode record for the latter. At the same time, we carry a PakaIntra environment during intra-procedural compilations. All these details are abstracted away by the builders we have defined in the previous section and that we abuse in this section.

At this stage, the compiler simply dispatches to construct-specific compilers. Hence the following code:

compileFoFtoPaka' :: ILFoF → PakaBuilding → PakaBuilding
compileFoFtoPaka' (FStatement stmt k) = compileFoFtoPakaStmt stmt k
compileFoFtoPaka' (@(FIf _ _ _)) = compileFoFtoPakaIf t
code
compileFoFtoPaka' (@(FNewDef _ _ _ _ _ _ _)) = compileFoFtoPakaFunDef t
code
compileFoFtoPaka' (@(FWhile _ _ _ _)) = compileFoFtoPakaWhile t
code
compileFoFtoPaka' (@(FDoWhile _ _ _ _)) = compileFoFtoPakaDoWhile t
code
compileFoFtoPaka' (@(FFor _ _ _ _ _ _ _)) = compileFoFtoPakaFor t
code
compileFoFtoPaka' (@(FSwitch _ _ _ _ _ _ _)) = compileFoFtoPakaSwitch t
code
compileFoFtoPaka' (FConstant e) = compileFoFtoPakaCst e

7.3.1 Compiling Function definition

The compilation of a function definition consists in building a prototype, compiling the body of the function, building it, and pursuing with the next definition.

compileFoFtoPakaFunDef :: ILFoF → PakaBuilding → PakaBuilding
compileFoFtoPakaFunDef (FNewDef funAttrs
funName
body
returnT
args
k) (cont, gEnv, lEnv) =
prototype funName (attr \< + > returnT\< + > text funName \< > parens functionArgs \< > semi)
7.3.2 Compiling Constant

This one is directly handled by the so-called builder:

\[
\text{compileFoFtoPakaCst :: PureExpr} \rightarrow \text{PakaBuilding} \rightarrow \text{PakaBuilding}
\]

\[
\text{compileFoFtoPakaCst} = \text{constant}
\]

7.3.3 Compiling Closing statements

As for closing statements, this is not much more difficult:

\[
\text{compileFoFtoPakaClosing :: FClosing} \rightarrow \text{PakaBuilding} \rightarrow \text{PakaBuilding}
\]

\[
\text{compileFoFtoPakaClosing (FReturn expr)} = \text{close $ PReturn expr}
\]

\[
\text{compileFoFtoPakaClosing (FBreak)} = \text{close PBreak}
\]

\[
\text{compileFoFtoPakaClosing (FContinue)} = \text{close PContinue}
\]

7.3.4 Compiling control-flow operators

The mechanics of control-flow operators does not vary much between operators, so they are all here, together.

Some points worth mentioning. First, sub-branches are compiled down with \text{compileFoFtoPaka}', as one would expect. Second, to get a \text{ILPaka} value out of an \text{ILPaka} \rightarrow \text{ILPaka} continuation \text{k}, we call \text{k pVoid}: \text{void} is the ultimate closing statement, after all. Third, an expression computing a tested value \text{must} return a pure expression, which we can grab \text{fromJust $ expr intraEnv}. This is an invariant, if not respected \text{fromJust} will blow up.

Finally, it’s all fine and good to compile sub-branches privately (inside \text{where} statements) but \text{don’t forget} to bring the resulting global and local environments in the public setting. This corresponds to the use of \text{second (const globalEnv)} and \text{third (const localEnv)} in the public flow. Also, \text{don’t forget} to thread these environments in your private compilations, too. Someone should think of a less error-prone solution.

\[
\text{compileFoFtoPakaIf :: ILFoF} \rightarrow \text{PakaBuilding} \rightarrow \text{PakaBuilding}
\]

\[
\text{compileFoFtoPakaIf (FIf cond)
ifTrue
ifFalse
k)} \text{ (cont, gEnv, lEnv)} =
\]

\[
\text{pif cond test cifTrue cifFalse
# second (const gEnv3)}
\]

# third (const lEnv3)
# compileFoFtoPaka' k
$ (cont, gEnv3, lEnv3)

where (cond_, gEnv1, lEnv1) = compileFoFtoPaka' cond (id, gEnv, lEnv)
    ccond = ccond_, PVoid
test = fromJust $ expr lEnv1
    (cifTrue_, gEnv2, lEnv2) = compileFoFtoPaka' ifTrue (id, gEnv1, lEnv1)
    cifTrue = cifTrue_, PVoid
    (cifFalse_, gEnv3, lEnv3) = compileFoFtoPaka' ifFalse (id, gEnv2, lEnv2)
    cifFalse = cifFalse_, PVoid

compileFoFtoPakaWhile (FWhile cond loop)
k) (cont, gEnv, lEnv) =
    while ccond test cloop
    # second (const gEnv2)
    # third (const lEnv2)
    # compileFoFtoPaka' k
$ (cont, gEnv2, lEnv2)

where (cond_, gEnv1, lEnv1) = compileFoFtoPaka' cond (id, gEnv, lEnv)
    ccond = ccond_, PVoid
test = fromJust $ expr lEnv1
    (cloop_, gEnv2, lEnv2) = compileFoFtoPaka' loop
        # compileFoFtoPaka' cond
        $(id, gEnv1, lEnv1)
    cloop = cloop_, PVoid

compileFoFtoPakaDoWhile (FDoWhile loop cond)
k) (cont, gEnv, lEnv) =
    pdoWhile cloop ccond test
    # second (const gEnv2)
    # third (const lEnv2)
    # compileFoFtoPaka' k
$ (cont, gEnv2, lEnv2)

where (cond_, gEnv1, lEnv1) = compileFoFtoPaka' cond (id, gEnv, lEnv)
    ccond = ccond_, PVoid
test = fromJust $ expr lEnv1
    (cloop_, gEnv2, lEnv2) = compileFoFtoPaka' loop
        # compileFoFtoPaka' cond
        $(id, gEnv1, lEnv1)
    cloop = cloop_, PVoid

compileFoFtoPakaSwitch (FSwitch test cases)
defaultCase
k) (cont, gEnv, lEnv) =
    pswitch test cases cdefaultCase
    # second (const gEnv2)
    # third (const lEnv2)
    # compileFoFtoPaka' k
$ (cont, gEnv, lEnv)

where (cdefaultCase_, gEnv1, lEnv1) = compileFoFtoPaka' defaultCase (id, gEnv, lEnv)
    cdefaultCase = cdefaultCase_, PVoid
    (codes, fcases) = unzip cases
    (ccases_, gEnv2, lEnv2) = compileCases fcases gEnv1 lEnv1
    ccases = zip codes ccases_
    compileCases [] x y = (][, x, y)


```
compileCases (fcase : fcases) gEnv lEnv =
  -- cfcase 'deepSeq' codes 'deepSeq'
  (cfcase : codes, gEnv2, lEnv2)
  where (cfcase, gEnv1, lEnv1) = compileFoFtoPaka' fcase (id, gEnv, lEnv)
    cfcase = cfcase' PVoid
  (codes, gEnv2, lEnv2) = compileCases fcases gEnv1 lEnv1
```

For my personal convenience, for loops are compiled into while loops. If you’re not happy with that, go ahead and implement that. However, I have to warn you that dealing with computations inside the indices is not a joy.

```
compileFoFtoPakaFor (FFor init
test
inc
loop
k) (cont, gEnv, lEnv) =
pwhile ccond etest cloop
  # second (const gEnv2)
  # third (const lEnv2)
  # compileFoFtoPaka' k
$ (cont, gEnv2, lEnv2)
  where (ccond, gEnv1, lEnv1) = compileFoFtoPaka' init
      # compileFoFtoPaka' test
      $ (id, gEnv, lEnv)
    ccond = ccond' PVoid
  etest = fromJust $ expr lEnv1
  (cloop, gEnv2, lEnv2) = compileFoFtoPaka' loop
    # compileFoFtoPaka' inc
    # compileFoFtoPaka' test
    $ (id, gEnv1, lEnv1)
  cloop = cloop' PVoid
```

7.3.5 Compiling statements

The real stuff happens below: compiling these damned statements. And there is a lot of them. That was for the bad news. The good news is that, individually, these functions are quite easy to understand.

The careful reader will notice that Terms are not using all their arguments. Honestly, I just wanted the basic Optimizer to be done, so I dropped everything not necessary. So, you have the architecture, now fill the holes if you want to do something clever. Therefore, when you see a term defined with (\(\lambda[zs, \_xss] \rightarrow \_\ldots\)), this means that the ignored variable is hard-coded in the term, and cannot be actually replaced. This is ok with my simple optimizer, that would probably need to be changed if you are to do something more clever.

Compiling References

As a starting, non frightening example, here is the code to compile references. Honestly, it is self-explanatory, isn’t it?

```
compileFoFtoPakaStmt (FNewRef varName dat) k =
  localVar mvarName (toC (typeOf dat) <+ toC varName <+ <> semi)
# assign pvarName (\(\lambda[\_e] \rightarrow toC varName <+ <> char '"' <+ <> e <+ <> <> semi)
  [pakaVarName dat]
# compileFoFtoPaka' k
```
where mvarName = mkPakaVarName varName
    peerName = Var $ mkPakaVarName varName

compileFoFtoPakaStmt (FReadRef varName ref) k =
    localVar mvarName (toC (unfoldPtrType ref) < + > toC varName <> semi)
    # assign peerName (λ[_, e] →
        toC varName < + > char '=' < + > e <> semi)
    [pakaValName ref]
    # compileFoFtoPaka' k
where mvarName = mkPakaVarName varName
    peerName = Var $ mkPakaVarName varName

compileFoFtoPakaStmt (FWriteRef ref dat) k =
    assign (pakaValName ref)
    (λ[_, e] → toC ref < + > char '=' < + > e <> semi)[pakaVarName dat]
    # compileFoFtoPaka' k

Compiling Arrays

Similarly, compiling arrays work the same way. There is minor nitpick in the current implementation:
it doesn’t support dynamic array (that is, malloc’ed arrays).

Actually, I suspect that if you are reading this file, it is because your code is using a dynamic array and
the compiler blew up when you use it. Well, the code needs to be written. It is remotely similar to
static arrays, with the additional need to malloc memory and initialize the data. If you are looking for
a word to describe your situation, I think that “screwed” is appropriate. Hint: a dynamic array should
be defined by a single initial element and an integer variable specifying (at run time) the length of the
array.

compileFoFtoPakaStmt (FNewArray varName alloc@ (StaticArray size) dat) k =
    globalVar mvarName (toC typeOfDat < + > toC varName <> brackets Pprinter.empty
                      < + > char '=' < + > braces (nest 4$
                      fsep (punctuate comma
                          [text (deref val) <> toC val
                          | val ← dat|]) <>
                      ) semi)
    # compileFoFtoPaka' k
where mvarName = mkPakaVarName varName
    typeOfDat = typeOf $ head dat

compileFoFtoPakaStmt (FReadArray varName (CLRef origin (TArray (StaticArray size) typ)
                      xloc) index) k =
    localVar mvarName (toC typ < + > toC varName <> semi)
    # (case symbEval index of
    CLInteger x →
        if x < (toInteger size) then
            assign peerName (λ[_,_] →
                toC varName < + > char '='
                < + > toC xloc <> brackets (toC index) <> semi)
            [Complex $ Var $ mkPakaVarName xloc]
else
    instr (\_\_\_\_ \_, \_, e) →
    text "assert" <> parens (text "! "ReadArray: Out of bound\"") <> semi
[
→
    assgn pvarName (λ[\_, e] →
    text "if" < + > parens (e
    < + > char '<'
    < + > int size) <> lbrace
$ +$
    next 4 (toC varName < + > char '='
    < + > toC xloc <> brackets e <> semi)
$ +$
    rbrace < + > text "else" < + > lbrace $ +$
    next 4 (text "assert" <> parens (text "! "ReadArray: Out of bound\"") <> semi
$ +$ toC varName < + > char '=' < + > text "NULL" <> semi)
$ +$
    rbrace)
    [ Complex $ Var $ mkPakaVarName xloc,
      pakaValName index ]
# compileFoFtoPaka' k
where mvarName = mkPakaVarName varName
    pvarName = Var $ mkPakaVarName varName
compileFoFtoPakaStmt (FWriteArray ref @(CLRef origin
    (TArray (StaticArray size) typ)
    xloc)
index
    dat) k =
    assgn pxloc (\_, e, f) →
    text "if" < + > parens (f < + > char '<' < + > int size) <> lbrace
$ +$
    next 4 (toC xloc <> brackets f
    < + > char '=' < + > e <> semi)
$ +$
    rbrace < + > text "else" < + > lbrace
$ +$
    next 4 (text "assert" <> parens (text "! "Out of bound\"") <> semi)
$ +$
    rbrace) [pakaValName dat, pakaValName index]
# compileFoFtoPaka' k
where pxloc = Var $ mkPakaVarName xloc

Compiling Strings

Building a new string is as simple as building a new static array:

    compileFoFtoPakaStmt (FNewString varName dat) k =
    globalVar mvarName (toC TChar < + > toC varName <> text "[]"
    < + > char '='
    < + > doubleQuotes (text dat) <> semi)
    # compileFoFtoPaka' k
where mvarName = mkPakaVarName varName

Compiling Function call

As for function call, there is no black magic either:
We can safely compile enumerations:

```
compileFoFtoPakaStmt (FNewEnum varName
  nameEnum
  fields
  initVal) k =
  declareEnum nameEnum fields
  # compileFoFtoPaka' k
  where mVarName = mkPakaVarName varName
       pvarName = Var $ mkPakaVarName varName
```

### Compiling Union

It is not a big deal to compile union operations either:

```
compileFoFtoPakaStmt (FNewUnion name
  DynamicUnion
  nameUnion
  fields
  (initField, initData)) k =
  declareRecursive (TUnion DynamicUnion nameUnion fields)
  # localVar (mkPakaVarName name) (text "union" <+ > text nameUnion <> char ',*' <> + > toC name <> semi)
  # assgn varName (\[\] ->
    toC name <+ > char ',*' <+ >
    parens (text "union" <+ > text nameUnion <> char ',*')
    <+ > text "malloc" <+ > parens (text "sizeof" <+ > parens (text "union" <+ > text nameUnion))
    <> semi) []
```
Compiling Structs

Quite the same goes for structure operations:

```haskell
compileFoFtoPakaStmt (FNewStruct varName DynamicStruct nameStruct) k = declareRecursive typeU

# assign pxloc (λ[_, e] ->
toC zloc <> ptrSigUnion alloc <> text field <> semi)

[ pakaVarName value

# compileFoFtoPaka' k

where pxloc = Var $ mkPakaVarName xloc
```
fields) k =
declareRecursive (TStruct DynamicStruct nameStruct fieldsTypeStr)

# localVar mVarName (text "struct" < + > text nameStruct < + > toC varName <> semi)
# (assign pVarName (λ_[_] ->
toC varName < + > char '=',
  < + > parens (text "struct" < + > text nameStruct < + > char '*/')
  < + > text "malloc"
  < > parens (text "sizeof"
  < > parens (text "struct" < + > text nameStruct)
  < > semi) []))
# foldl' (#) id [assign pVarName (λ_[_] e ->
toC varName <> text "->" <> text field
  < + > char '='
  < + > e <> semi) [pakaVarName val]
  | (field, (typ, val)) <- fields]
  <> semi)
where mVarName = mkPakaVarName varName
  pVarName = Var $ mkPakaVarName varName
  fieldsTypeStr = [(field, typ)
    | (field, (typ, _)) <- fields]

calculateFoFtoPakaStmt (FNewStruct varName
  StaticStruct
  nameStruct
  fields) k =
declareRecursive (TStruct StaticStruct nameStruct fieldsTypeStr)

# localVar mVarName (text "struct" < + > text nameStruct < + > toC varName
  < + > char '='
  < + > braces (nest 4 $)
  hcat (punctuate comma
    [text (deref val) <> toC val
    | (_(_, val)) <- fields])
  <> semi)
# compileFoFtoPaka' k
where mVarName = mkPakaVarName varName
  typeField = fromJust $ field 'lookup' fields
  mVarName = mkPakaVarName varName
  pVarName = Var $ mkPakaVarName varName

calculateFoFtoPakaStmt (FReadStruct ref
  @ (CLRef origin
    typeS@(TStruct alloc
      nameStruct
      fields)
    xloc)
  field) k =
declareRecursive typeS

# localVar mVarName (toC typeField < + > toC varName <> semi)
# assign pVarName (λ_[_] ->
toC varName < + > char '=',
  < + > toC xloc <> ptrSigStruct alloc <> text field <> semi)
[Complex $ Var $ mkPakaVarName xloc]
# compileFoFtoPaka' k
where typeField = fromJust $ field 'lookup' fields
  mVarName = mkPakaVarName varName
  pVarName = Var $ mkPakaVarName varName

calculateFoFtoPakaStmt (FWriteStruct ref @ (CLRef origin
  typeS@(TStruct alloc
  nameStruct

---

Barrelfish TN-024 FiletOFish - 69
fields
zloc
field
value) k =
declareRecursive typeS
  # assign pxloc (λzloc, e) →
toC zloc <> ptrSigStruct alloc <> text field
  < + > char '=' < + > e <> semi)
  [pakaVarName value]
  # compileFoFtoPaka' k
  where pxloc = Var $ mkPakaVarName zloc

Compiling Typedef

And we can even get typedefs:

\[
\text{compileFoFtoPakaStmt (FTypedef typ aliasName) k =}
\text{declareRecursive typ}
\]
  # declare aliasName Pprinter.empty
  (text "typedef" < + > toC typ < + > text aliasName <> semi)
  # compileFoFtoPaka' k

\[
\text{compileFoFtoPakaStmt (FTypedefE inclDirective}
\text{(TTypedef typ aliasName)) k =}
\text{include inclDirective}
\]
  # compileFoFtoPaka' k

Compiling Foreign calls

It is always the same story for foreign function calls. If you have extended Filet-o-Fish with a new
foreign-function, don’t look further: you should put your foreign call here!

So, as often, we have an inoffensive dispatcher. Don’t touch it.

\[
\text{compileFoFtoPakaStmt (FFFICall nameCall args) k =}
\text{compileFFI nameCall args}
\]
  # compileFoFtoPaka' k

And the dispatched function, in which you should add your foreign code generator. This is just like
writing C code, so don’t be shy.

\[
\text{compileFFI nameCall params | nameCall ≡ "printf" =}
\text{include "<stdio.h>"}
\]
  # instr (\_ → text "printf" <> parens (heat (punctuate comma (map toC params))) <> semi)
  (map (Complex ◦ pakaVarName) params)

\[
\text{compileFFI nameCall [e] | nameCall ≡ "assert" =}
\text{include "<assert.h>"}
\]
  # instr (λ[e] → text "assert" <> parens e <> semi) [pakaValName e]

\[
\text{compileFFI nameCall [varName, param] | nameCall ≡ "has_descendants" =}
\text{include "<mdb/mdb.h>"}
\]
  # include "<capabilities.h>"
  # include "<stdbool.h>"
  # localVar (show $ toC varName)
  (text "bool" < + > toC varName <> semi)
Declaring types

Above, we have dealt with the compilation of operations on complex structures, such as enums, structs, and unions. When compiling a code operating on such structure, we need to make sure that the corresponding type is defined.

Hence, we provide an advanced builder to declare a struct or an union:

\[
\text{declareStructUnion kind name fields} = \\
\text{declare name (text kind < + > text name <> semi)} \\
\text{(text kind < + > text name < + > braces (}
\text{nest 4 (vcat' [toC typ < + > text field <> semi}
\text{-- special case for static array?}
\text{| (field, typ) ← fields]))) <> semi)
\]

And similarly for declaring an enum, however without the forward declaration:

\[
\text{declareEnum nameEnum fields} = \\
\text{declare nameEnum empty}
\text{(text "enum" < + > text nameEnum < + > lbrace}
\text{\$ + \$ nest 4 (vcat' \$ punctuate comma}
\text{[(text name < + > char ^= < + > int val}
\text{| (name, val) ← fields)])}
\text{\$ + \$ rbrace <> semi)}
\]
However, that does not solve the problem: a structure or an union might be defined in term of other structures or unions. Hence, we need to declare the dependencies before defining the concerned object. This is handled by \texttt{declareRecursive}:

\begin{verbatim}
declareRecursive = declareRecursive'
  where declareRecursive' (TStruct name fields) (code, gEnv, lEnv) =
    case name `Map.lookup' types gEnv of
      Just → (code, gEnv, lEnv)
      Nothing →
        foldl' (#) id [declareRecursive' typ | (_, typ) ← fields]
        # declareStructUnion "struct" name fields
        $ (code, gEnv, lEnv)

declareRecursive' (TUnion name fields) (code, gEnv, lEnv) =
  case name `Map.lookup' types gEnv of
      Just → (code, gEnv, lEnv)
      Nothing →
        foldl' (#) id [declareRecursive' typ | (_, typ) ← fields]
        # declareStructUnion "union" name fields
        $ (code, gEnv, lEnv)

declareRecursive' (TEnum name fields) t =
  declareEnum name fields t
  declareRecursive' t = id t
\end{verbatim}

These two functions have also been handy above, even though they are not fundamentally clever. Depending on the allocation policy of the data-structure, they choose to dereference and access it, or directly access it.

\begin{verbatim}
ptrSigUnion :: AllocUnion → Doc
ptrSigUnion DynamicUnion = text "->"
ptrSigUnion StaticUnion = char ';

ptrSigStruct :: AllocStruct → Doc
ptrSigStruct DynamicStruct = text "->"
ptrSigStruct StaticStruct = char ';
\end{verbatim}

\section{Translating IL.Paka to C}

This file could as well be called ./IL/C/C.lhs but I felt guilty of introducing yet another confusing IL. So, it is here but feel free to move it around.

\subsection{Printing types and expressions}

Because we are good kids, we create a type-class called \texttt{Compileable}. A data-type satisfying \texttt{Compileable} can be pretty-printed to something vaguely looking like a bunch of C code.

\begin{verbatim}
class Compileable a where
toC :: a → Doc
\end{verbatim}

Part of the \texttt{Compileable} class are FoF’s types \texttt{TypeExpr} and FoF’s pure expressions \texttt{PureExpr}.

There is nothing but boiler plate code to get the job done for pure expressions:

\begin{verbatim}
instance Compileable PureExpr where
toC (Quote s) = doubleQuotes $ text s
\end{verbatim}
toC Void = empty
toC (CLInteger _ x) = integer x
toC (CLFloat x) = Pprinter.float x
toC (CLRef origin (TPPointer _ Avail) loc) = toC loc
toC (CLRef origin (TPPointer _ Read) loc) = char '*' <> toC loc
toC (CLRef origin _) = toC loc
toC (Unary op x) = parens $ toC op < + > toC x
toC (Binary op x y) = parens $ toC x < + > toC op < + > toC y
toC (Sizeof t) = text "sizeof" <> (parens $ toC t)
toC (Test t1 t2 t3) = parens $ parens (toC t1) < + > char '?'> < + > parens (toC t2) < + > char ':'> < + > parens (toC t3)
toC (Cast t e) = parens $ parens (toC t) < + > toC e

instance Compileable UnaryOp where
toC Minus = char '
'toC Complement = char '!
toC Negation = char '!' 

instance Compileable BinaryOp where
toC Plus = text "+
toC Sub = text "-
toC Mul = text "*
toC Div = text "/
toC Mod = text "%
toC Shl = text "<<
toC Shr = text ">>
toC AndBit = text "&
toC OrBit = text "|
toC XorBit = text "^'
toC Le = text "<
toC Leq = text "<=
toC Ge = text ">
toC Geq = text ">=
toC Eq = text "==
toC Neq = text "!='

And similarly for types:

instance Compileable TypeExpr where

toC (TInt Signed TInt8) = text "int8_t"
toC (TInt Signed TInt16) = text "int16_t"
toC (TInt Signed TInt32) = text "int32_t"
toC (TInt Signed TInt64) = text "int64_t"
toC (TInt Unsigned TInt8) = text "uint8_t"
toC (TInt Unsigned TInt16) = text "uint16_t"
toC (TInt Unsigned TInt32) = text "uint32_t"
toC (TInt Unsigned TInt64) = text "uint64_t"
toC TFloat = text "float"
toC TVoid = text "void"
toC TChar = text "char"
toC (TArray DynamicArray typ) = toC typ <> char '*' 
toC (TArray (StaticArray size) typ) = toC typ <> char '*'
toC (TPPointer x _) = toC x <> char '*'
toC (TStruct DynamicStruct name fields) = text "struct " < + > text name < + > char '*'
toC (TStruct StaticStruct name fields) = text "struct " < + > text name
toC (TUnion DynamicUnion name fields) = text "union " < + > text name < + > char '*"
toC (TUnion StaticUnion name fields) = text "union " <+ > text name
toC (TCompPointer name) = text "uintptr_t"
toC (TTypedef typ name) = text name
toC (TEnum name _) = text "enum" <+ > text name

The picky reader will have noticed the absence of printer for function types. This is hardly a problem at
the moment because we do not support function pointers, so we are not going to declare function types
anytime soon. Note that this argument might well be circular: if we do not support function pointers,
it is because it is a pain to write their type, among other things (if I remember correctly). Oh well.

Printing variable names is dead easy:

instance Compileable VarName where
toC x = text $ mkPakaVarName x

7.4.2 Names, everywhere

I am not very proud of that section, and of the way I abused these functions in IL/Paka/Paka.lhs. I
beg your pardon for that. There must some abstraction to bust here but I was not able to catch it.

Provided a FoF VarName, we turn it into a string with a bit of Hungarianism, but very little. Why this
function is called mkPakaVarName when it does not deal with PakaVarName? I have no clue.

mkPakaVarName :: VarName -> String
mkPakaVarName (Generated x) = "." <+ x
mkPakaVarName (Provided x) = x
mkPakaVarName (Inherited y x) = mkPakaVarName x + "." <+ show y

Then, we have to functions turning a PureExpr into a PakaVarName. PakaValName provides you with the
value described by the PureExpr. On the other hand, PakaVarName works one level below and
gives you the value contained in the PureExpr.

I have to admit that I am not myself convinced by this explanation. Basically, I would have to look at
the former code, the typeOf, deref, readOf functions, and the new code. Then, I might be able to make
more sense of that. However, intrinsically, references are a non-sense.

pakaValName :: PureExpr -> PakaVarName
pakaValName (CLRef origin (TPointer Avail) loc) = Var $! mkPakaVarName loc
pakaValName (CLRef origin (TPointer Read) loc) = Ptr $! Var $ mkPakaVarName loc
pakaValName (CLRef _ loc) = Var $! mkPakaVarName loc
pakaValName x = K x

pakaVarName :: PureExpr -> PakaVarName
pakaVarName (CLRef origin (TPointer _ Avail) loc) = Deref (Var $ mkPakaVarName loc)
pakaVarName (CLRef origin (TPointer _ Read) loc) = Var $ mkPakaVarName loc
pakaVarName (CLRef _ loc) = Var $ mkPakaVarName loc
pakaVarName x = K x

Finally, we need to be able to print these PakaVarName into meaning C code. Here you go.

instance Compileable PakaVarName where
toC (Deref x) = char '&' <+ toC x
toC (Var x) = text x
toC (Ptr x) = char '*' <+ toC x
toC (Complex _) = error "Cannot convert a Complex var name to C"
toC (K x) = toC x

FiletOfFish - 74 Barrelfish TN-024
7.4.3 Generating C

The following is a small addendum to the pretty-printer library. We don't know why it is not defined there.

\[
\text{\texttt{vcat'}} :: \texttt{[Doc]} \rightarrow \texttt{Doc} \\
vcat' [] = \text{\texttt{empty}} \\
vcat' (x : xs) = \text{\texttt{l}} \cdot \text{\texttt{seq}} \cdot r' \cdot \text{\texttt{seq}} \cdot \text{\texttt{r}} \\
\textbf{where} \ l = vcat' \ xs \\
\text{\texttt{r}} = x + \text{\texttt{r}} + \text{\texttt{l}}
\]

For once, I will do a bottom-up presentation. So, I will describe the implementation of pretty-printers from Paka code to C.

The first step consists in printing closing terms:

\[
\text{\texttt{pprintClosing}} :: \texttt{PakaClosing} \rightarrow \texttt{Doc} \\
\text{\texttt{pprintClosing}} (\texttt{PReturn} \ e) = \text{\texttt{text}} \text{\texttt{"return"}} < + > \text{\texttt{parens}} (\text{\texttt{toC}} \ e) <> \text{\texttt{semi}} \\
\text{\texttt{pprintClosing}} \text{\texttt{PBreak}} = \text{\texttt{text}} \text{\texttt{"break"}} \\
\text{\texttt{pprintClosing}} \text{\texttt{PContinue}} = \text{\texttt{text}} \text{\texttt{"continue"}}
\]

Then, we print statements. As you remember, we need to build the final code by applying the variables to the term:

\[
\text{\texttt{pprintStmt}} :: \texttt{PakaStatement} \rightarrow \texttt{Doc} \\
\text{\texttt{pprintStmt}} (\texttt{PAssign} \ dst \ x \ \texttt{srcs}) = x \ (\text{\texttt{toC}} \ dst : \text{\texttt{map}} \ \text{\texttt{toC}} \ \texttt{srcs}) \\
\text{\texttt{pprintStmt}} (\texttt{PInstruction} \ x \ \texttt{srcs}) = x \ (\text{\texttt{map}} \ \text{\texttt{toC}} \ \texttt{srcs})
\]

The next step consists in compiling intra-procedural code. This is rather simple and quite directly follows from the Paka definitions:

\[
\text{\texttt{pprintPaka}} :: \texttt{ILPaka} \rightarrow \texttt{Doc} \\
\text{\texttt{pprintPaka}} \text{\texttt{PVoid}} = \text{\texttt{empty}} \\
\text{\texttt{pprintPaka}} (\texttt{PClosing} \ c) = \text{\texttt{pprintClosing}} \ c \\
\text{\texttt{pprintPaka}} (\texttt{PStatement} \ stmt \ k) = \\
\text{\texttt{pprintStmt}} \ stmt + \text{\texttt{pprintPaka}} \ k \\
\text{\texttt{pprintPaka}} (\texttt{PIf} \ cond \ test \ ifTrue \ ifFalse \ k) = \\
\text{\texttt{pprintPaka}} \ cond + \text{\texttt{text}} \text{\texttt{"if"}} < + > \text{\texttt{parens}} (\text{\texttt{toC}} \ test) <> \text{\texttt{brace}} + \text{\texttt{brace}} < + > \text{\texttt{text}} \text{\texttt{"else"}} < + > \text{\texttt{brace}} + \text{\texttt{brace}} + \text{\texttt{pprintPaka}} \ ifTrue + \text{\texttt{pprintPaka}} \ ifFalse + \text{\texttt{pprintPaka}} \ k \\
\text{\texttt{pprintPaka}} (\texttt{PWhile} \ cond \ test \ loop \ k) = \\
\text{\texttt{pprintPaka}} \ cond + \text{\texttt{text}} \text{\texttt{"while"}} <> \text{\texttt{parens}} (\text{\texttt{toC}} \ test) <> \text{\texttt{brace}} + \text{\texttt{brace}} + \text{\texttt{pprintPaka}} \ loop + \text{\texttt{pprintPaka}} \ k \\
\text{\texttt{pprintPaka}} (\texttt{PDoWhile} \ loop \ cond \ test \ k) = \\
\text{\texttt{pprintPaka}} \ cond + \text{\texttt{text}} \text{\texttt{"do"}} < + > \text{\texttt{brace}} + \text{\texttt{brace}} + \text{\texttt{pprintPaka}} \ loop + \text{\texttt{pprintPaka}} \ k \\
\text{\texttt{pprintPaka}} (\texttt{PSwitch} \ test \ cases \ defaultCase \ k) = \\
\text{\texttt{pprintPaka}} \ cond + \text{\texttt{text}} \text{\texttt{"switch"}} < + > \text{\texttt{parens}} (\text{\texttt{toC}} \ test) < + > \text{\texttt{brace}} + \text{\texttt{brace}} + \text{\texttt{pprintPaka}} \ defaultCase
(nest 4 $ vcat' $ map compileCase cases) $ + $
(nest 4 (text "default:" < + > lbrace $ + $
  (nest 4! pprintPaka defaultCase) $ + $
  rbrace)) $ + $
  rbrace $ + $
printPaka k

where compileCase (i, code) =
text "case" < + > toC i <> colon < + > lbrace $ + $
  (nest 4! (pprintPaka code) $ + $
  text "break" <> semi)) $ + $
  rbrace

Finally, we can pretty-print a complete PakaCode by iterating over each section, and, in each section, pretty-printing each element.

pprint :: PakaCode → Doc
pprint code =
text "/* Includes: */" $ + $
space $ + $
text "#include <stdint.h>" $ + $
vcat' (extractM $ includes code) $ + $
space $ + $
(case Map.null $ types code of
  True → empty
  _ → text "/* Type Declarations: */" $ + $
space $ + $
vcat' (extractM $ types code) $ + $
vcat' (extractL $ declarations code) $ + $
space) $ + $
(case null $ globalVars code of
  True → empty
  _ → text "/* Global Variables: */" $ + $
space $ + $
vcat' (map (λ y → text "static" < + > y) $ 
  extractL $ 
  globalVars code) $ + $
space) $ + $
(case Map.null $ prototypes code of
  True → empty
  _ → text "/* Prototypes: */" $ + $
space $ + $
vcat' (extractM $ prototypes code) $ + $
space) $ + $
(case Map.null $ functions code of
  True → empty
  _ → text "/* Function Definitions: */" $ + $
space $ + $
vcat' (map (λ (returnT, attrs, name, args, lEnv, body) → 
  returnT < + > attrs < + > text name <> parens args < + > lbrace $ + $
  (nest 4 $ vcat' $ extractM $ localVars lEnv) $ + $
  space $ + $
  (nest 4 $ vcat' $ extractM $ localVars lEnv) $ + $
  space) 
  $ extractM 
  $ functions code) $ + $

FiletOFish - 76

Barrelfish TN-024
We note the use of two extraction functions: these functions remove the keys from the associative structure in use, and simply return the content. When an order was maintained, ie. an associative list was used, the definition order is carefully restored by reversing the list.

\[
\begin{align*}
\text{extractL} & :: \text{Eq } a \Rightarrow [(a, b)] \rightarrow [b] \\
\text{extractL} & = (\text{map snd}) \circ \text{reverse} \\
\text{extractM} & :: \text{Map.Map } a \ b \rightarrow [b] \\
\text{extractM} & = \text{Map.elems}
\end{align*}
\]

Because we have worked very hard, we are rewarded by the right to instantiate these \textit{PakaCode} in the \textit{Show}.

\[
\text{instance } \text{Show } \textit{PakaCode} \text{ where} \\
\text{show} = \text{render} \circ \text{pprint}
\]

### 7.5 IL.Paka Code Optimizer

The currently implemented optimizer is a naive redundant assignment simplifier, which happens to do constant propagation at the same time. It is naive in the several dimensions. An important one is that it is entirely hard-coded, while we all know that optimization is simply a matter of dataflow analysis. So, at some point, we should use a more generic framework for that. It is also naive because it does not try to reach a fix-point: it is single phase, while it is obvious that more assignments could still be eliminated in subsequent phases. Finally, it is naive because any case that was not easy to deal with have been discarded: more redundant assignments could be removed if the logic were more precise.

The purpose of that module is to show that “it is possible to do optimization”. It is a proof of concept. Now, it is Future Work (Chapter A) to get a clever optimization framework. The ease I had in implementing that stuff convince me that we are not far from this heaven.

So, if you want optimized Paka code, you will only get a slightly less redundant code:

\[
\begin{align*}
\text{optimizePaka} & :: \textit{PakaCode} \rightarrow \textit{PakaCode} \\
\text{optimizePaka} & = \text{optimizeAssgmtElim}
\end{align*}
\]

Because this analysis is intra-procedural, we go over each function and apply an intra-procedural optimizer:

\[
\begin{align*}
\text{optimizeAssgmtElim} & :: \textit{PakaCode} \rightarrow \textit{PakaCode} \\
\text{optimizeAssgmtElim} \text{ code} & = \text{code} \{ \text{functions} = \text{optFunc} \} \\
\text{where} & \text{ func} = \text{functions code} \\
\text{optFunc} & = \text{Map.mapMaybe } (\lambda(b, c, d, e, f, \text{fun}) \rightarrow \text{Just } (b, c, d, e, f, \text{assgmtElim fun})) \text{ func}
\end{align*}
\]

### 7.5.1 Implementation

This optimizer is quite easy to implement, assuming we have the right tools at hand. That is, assuming that we are able to replace a variable \(x\) by a variable \(y\) in a code \(k\) – using \textit{replace} (\textit{Var} \(x\)) (\textit{Var} \(y\)) \(k\) –, that we are able to say if a variable \(y\) is either a constant or never used in a code \(k\) – using \textit{isUsed flatten} \(y\) \(k\) –, and that we are able to say if a variable \(x\) is used without side-effects in a code \(k\) – using \textit{isUsed flattenS} \(x\) \(k\).
The intra-procedural optimizer will turn an ILPaka into a better ILPaka:

\[ \text{assgmtElim} :: \text{ILPaka} \rightarrow \text{ILPaka} \]

The interesting case is obviously the variable assignment: a value \( y \) is assigned to a variable \( x \). We remove that assignment and replace \( x \) by \( y \) if and only if \( y \) is never used again and \( x \) is not involved in some weird computation. Otherwise, we go ahead.

A small issue here is that we ask for \( y \) to be never used again. That’s quite restrictive. This results in being able to carry only 4 assignment eliminations on today’s Hamlet and Fugu inputs. This is shame, compared to the numerous opportunities. To solve that issue, we would have to extend or re-design \text{isUsed} to allow the definition of more fine-grained predicates, such as “is overwritten”.

\[
\text{assgmtElim} (PStatement \ a@((PAssign \ (Var \ x)) \rightarrow [Var \ y]) \ k) =
\begin{align*}
& \text{if } (\neg (\text{isUsed flatten y k})) \\
& \quad \land (\neg (\text{isUsed flattenS x k})) \text{ then} \\
& \quad \text{assgmtElim } $ replace (Var \ x) (Var \ y) k \\
& \text{else} \\
& \quad PStatement \ a $ \\
& \quad \text{assgmtElim } k
\end{align*}
\]

All other assignments that do not fit this scheme, or the instructions are skipped:

\[
\text{assgmtElim} (PStatement \ a \ k) = \\
\quad PStatement \ a $ \\
\quad \text{assgmtElim } k
\]

Finally, control-flow operators are simply iterated over:

\[
\begin{align*}
\text{assgmtElim} (PIf \ c \ t \ ifT \ ifF \ k) &= \\
\quad PIf \\
& \quad (\text{assgmtElim } c) \\
& \quad t \\
& \quad (\text{assgmtElim } ifT) \\
& \quad (\text{assgmtElim } ifF) \\
& \quad (\text{assgmtElim } k)
\end{align*}
\]

\[
\begin{align*}
\text{assgmtElim} (PWhile \ c \ t \ l \ k) &= \\
\quad PWhile \\
& \quad (\text{assgmtElim } c) \\
& \quad t \\
& \quad (\text{assgmtElim } l) \\
& \quad (\text{assgmtElim } k)
\end{align*}
\]

\[
\begin{align*}
\text{assgmtElim} (PDoWhile \ l \ c \ t \ k) &= \\
\quad PDoWhile \\
& \quad (\text{assgmtElim } l) \\
& \quad (\text{assgmtElim } c) \\
& \quad t \\
& \quad (\text{assgmtElim } k)
\end{align*}
\]

\[
\begin{align*}
\text{assgmtElim} (PSwitch \ t \ cases \ d \ k) &= \\
\quad PSwitch \\
& \quad t \\
& \quad (map (\lambda(a, b) \rightarrow (a, \text{assgmtElim } b)) \ cases) \\
& \quad (\text{assgmtElim } d) \\
& \quad (\text{assgmtElim } k) \\
& \quad \text{assgmtElim } x = x
\end{align*}
\]
7.5.2 Code predication

First, if I correctly remember my Software Testing lecture, a *use site* is a place where a variable is read. In opposition to a *def site* where a variable is written to. Well, then the following is misleading.

\( \text{isUsed } f \ x \ k \) tells you that \( x \) has been found in a use or def site of \( k \) in a situation where it played a role caught by \( f \). To simplify, \( \text{isUsed flatten} \) will catch any kind of use or def. \( \text{isUsed flattenS} \) will catch a use or def in a Complex state.

As for the implementation, it is simply going over ILPaka terms and doing the necessary on PStatement.

\[
\begin{align*}
\text{isUsed} & \colon (\text{PakaVarName} \to \text{Maybe String}) \to \text{String} \to \text{ILPaka} \to \text{Bool} \\
isUsed p var PVoid &= False \\
isUsed p var (PClosing (PReturn k)) &= Just \ var \equiv (\text{flatten } $ pakaValName k) \\
isUsed p var (PClosing _) &= False \\
isUsed p var (PStatement s k) &= \text{isUsedStmt } s \lor \text{isUsed } p \ var \ k \\
\quad \text{where} \ isUsedStmt (PAssign t ls) &= \\
\quad \quad \quad \quad \text{Just } \ var \in \text{map } \text{flatten} \ (t : ls) \\
\quad \text{isUsedStmt} (PInstruction ls) &= \\
\quad \quad \quad \quad \text{Just } \ var \in \text{map } \text{flatten } ls \\
isUsed p var (PIf c t ifT ifF k) &= (\text{Just } \ var \equiv (\text{flatten } $ pakaValName t)) \lor \\
\quad (\text{isUsed } p \ var \ c \lor \text{isUsed } p \ var \ ifT \lor \text{isUsed } p \ var \ ifF \lor \text{isUsed } p \ var \ k) \\
isUsed p var (PWhile c t l k) &= (\text{Just } \ var \equiv (\text{flatten } $ pakaValName t)) \lor \\
\quad \text{isUsed } p \ var \ c \lor \text{isUsed } p \ var \ l \lor \text{isUsed } p \ var \ k \\
isUsed p var (PDoWhile l c t k) &= (\text{Just } \ var \equiv (\text{flatten } $ pakaValName t)) \lor \\
\quad \text{isUsed } p \ var \ c \lor \text{isUsed } p \ var \ l \lor \text{isUsed } p \ var \ k \\
isUsed p var (PSwitch t c d k) &= (\text{Just } \ var \equiv (\text{flatten } $ pakaValName t)) \lor \\
\quad \text{foldl}' (\lambda a (\_ , b) \to a \lor \text{isUsed } p \ var \ b) \ False \ c \lor \text{isUsed } p \ var \ d \lor \text{isUsed } p \ var \ k \\
\end{align*}
\]

In light of the explanation above, the definition of flatten and flattenS should be obvious. Aren’t they?

\[
\begin{align*}
\text{flatten} & \colon \text{PakaVarName} \to \text{Maybe String} \\
\text{flatten} \ (\text{Var } s) &= \text{Just } s \\
\text{flatten} \ (\text{Ptr } x) &= \text{flatten } x \\
\text{flatten} \ (\text{Deref } x) &= \text{flatten } x \\
\text{flatten} \ (\text{Complex } x) &= \text{flatten } x \\
\text{flatten} \ (K _) &= \text{Nothing} \\
\text{flattenS} & \colon \text{PakaVarName} \to \text{Maybe String} \\
\text{flattenS} \ (\text{Var } s) &= \text{Nothing} \\
\text{flattenS} \ (\text{Ptr } x) &= \text{Nothing} \\
\text{flattenS} \ (\text{Deref } x) &= \text{Nothing} \\
\text{flattenS} \ (\text{Complex } x) &= \text{flatten } x \\
\text{flattenS} \ (K _) &= \text{Nothing}
\end{align*}
\]

7.5.3 Code transformation

As for replace, it is by now standard: go over the terms, hunt the dest, and kill it with source. It is surgical striking, in its full glory.
replace :: PakaVarName → PakaVarName → ILPaka → ILPaka
replace dest source (PStatement (PAssign dst stmt srcs) k) =  
PStatement (PAssign dst stmt srcs')
  (replace dest source k)
  where srcs' = replaceL dest source srcs
replace dest source (PStatement (PInstruction stmt srcs) k) =  
PStatement (PInstruction stmt srcs')
  (replace dest source k)
  where srcs' = replaceL dest source srcs
replace dest source (PIf c t ifT ifF k) =  
PIf (replace dest source c) t
  (replace dest source ifT)
  (replace dest source ifF)
  (replace dest source k)
replace dest source (PWhile c t l k) =  
PWhile (replace dest source c)
  t
  (replace dest source l)
  (replace dest source k)
replace dest source (PDoWhile l c t k) =  
PDoWhile (replace dest source l)
  (replace dest source c)
  t
  (replace dest source k)
replace dest source (PSwitch t cases d k) =  
PSwitch t
  (map (λ(a, b) → (a, replace dest source b)) cases)
  (replace dest source d)
  (replace dest source k)
replace dest source x = x
replaceL x y = map (λz → if z ≡ x then y else z)
Part III

Appendix
Appendix A

Future Work

Follow! But! follow only if ye be men of valor, for the entrance to this cave is guarded by a creature so foul, so cruel that no man yet has fought with it and lived! Bones of four fifty men lie strewn about its lair. So, brave knights, if you do doubt your courage or your strength, come no further, for death awaits you all with nasty big pointy teeth.

— Monty Python

This is going to look like a brain dump, despite any effort to make it understandable by the Outside World.

**Module import clean-up:** for historical reasons, some imports might be completely useless now. Similarly, imports such as —Debug.Trace— should disappear too;

**Paka terms with real holes:** in Section 7.3, we have seen that Paka terms are ignoring most of their holes by using hard-coded values;

**More efficient redundant assignment optimizer:** in Chapter 7.5, we have seen that the optimizer is quite conservative, making it quite useless in practice;

**Supporting function pointers:** preventing Filet-o-Fish users to abuse function pointers is a violation to Geneva convention. I do not think that there is some deep technical difficulty to get that. But printing the type of such pointer was a first trouble, if I remember correctly;

**Implementing the interpreter in the Agda language:** this was already one of my goal initially, but the NICTA people insisted that without an in-theorem-prover semantics, the dependability argument is just bullsh*t. Ha, these Australians . . . ;

**Code generator back-back-end:** following the steps of FoF and Paka, we need a more principled back-back-end, generating (correct) out of —FoFCode—;

**Hoopl-based optimization framework:** the Hoopl [4] framework is a promising tool to implement any kind of data-flow analysis and optimization. Instead of developing our own crappy optimizer, we should use that stuff, when the source is released. This is the reason why @IL.Paka.Optimizer@ is such a joke: it must be dropped asap;

**Translation validation infrastructure:** because we claim dependability but our compiler is such a tricky mess, we need a good bodyguard. Translation validation [3] is an affordable technique that tells you, when you run your compiler, if it has barfed (and where), or not. If it has not failed, then you know for sure that the generated code is correct;
More stringent syntactic tests: it is very easy to build ill-formed Filet-o-Fish terms, because the types of constructs have not been engineered to ensure their invariants, and there is little or no run-time checks. It is just a matter of putting more run-time checks, a lot more.

Compiling to macros: that’s an interesting topic: we are able to generate C code. We might need to generate C macro at some point. How would that fit into Filet-o-Fish?

Compiling with assertions: assuming that Filet-o-Fish-generated C code is correct, we are ensured that it must never failed at run-time, except if it is provided with bogus input data. Being able to specify what is a valid input data and translating that into assertions might be useful. Similarly, when reading in an array, for example, we probably want to ensure that we are not going out of bounds, and an assert should fail if this is the case.
References