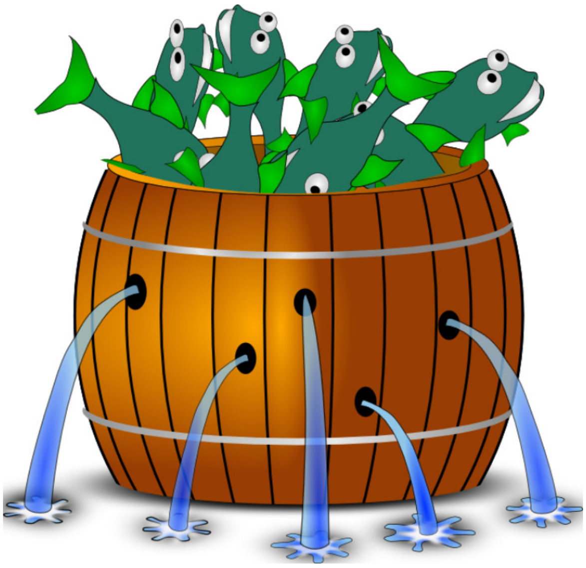
Barrelfish Project
ETH Zurich



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

Revision	Date	Author(s)	Description
1.0	02.06.2017	TR	Converted to Technical Note from Pierre's LHS

Contents

Introduction

> The Filet-o-Fish contains a battered fish patty made mostly from pollock and/or hoki.
>
> Wikipedia

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*, ie. 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 barrelfish 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 $\mathit{TypeExpr}$ data-type encompasses the following types:

- Void,
- Integers, of various signedness and size,
- Float,
- Named structures and unions,
- Named pointers, ie. 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* $\mathit{TInt}\ \mathit{Signed}\ \mathit{TInt32}$.

$$\begin{aligned}
&\textbf{data}\ \mathit{TypeExpr} = \mathit{TVoid}\\
&\quad \mid \mathit{TInt}\ \mathit{Signedness}\ \mathit{Size}\\
&\quad \mid \mathit{TFloat}\\
&\quad \mid \mathit{TChar}
\end{aligned}$$

```
    | TStruct AllocStruct String TFieldList
    | TUnion AllocUnion String TFieldList
    | TCompPointer String
    | TEnum String [(String, Int)]
    | TArray AllocArray TypeExpr
    | TPointer TypeExpr Mode
    | TTypedef TypeExpr String
    | TFun String Function TypeExpr [(TypeExpr, Maybe String)]
      deriving (Eq, Show)
```

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:

```
data Function = Fun ([PureExpr] → FoFCode PureExpr)
```

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

```
instance Show Function where
  show _ = "<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.

```
instance Eq Function where
  _ ≡ _ = 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 differents definitions for each kind of data-type because they are likely to evolve in future versions and diverge from this common scheme.

```
data AllocStruct = StaticStruct
   | DynamicStruct
     deriving (Eq, Show)
data AllocUnion = StaticUnion
   | DynamicUnion
     deriving (Eq, Show)
data AllocArray = StaticArray Int
   | DynamicArray
     deriving (Eq, Show)
```

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

```
type TFieldList = [(String, TypeExpr)]
```

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.

```
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 *Avail*able or *Read*. This distinction makes sense during the compilation process, it can ignored otherwise.

```
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:

$$
\begin{array}{l}
\mathit{arrayDT} :: \mathit{TypeExpr} \to \mathit{TypeExpr} \\
\mathit{arrayDT}\ \mathit{typ} = \mathit{TArray}\ \mathit{DynamicArray}\ \mathit{typ} \\
\mathit{arrayST} :: \mathit{Int} \to \mathit{TypeExpr} \to \mathit{TypeExpr} \\
\mathit{arrayST}\ \mathit{size}\ \mathit{typ} = \mathit{TArray}\ (\mathit{StaticArray}\ \mathit{size})\ \mathit{typ} \\
\mathit{ptrT} :: \mathit{TypeExpr} \to \mathit{TypeExpr} \\
\mathit{ptrT}\ \mathit{typ} = \mathit{TPointer}\ \mathit{typ}\ \mathit{Avail} \\
\mathit{structDT}, \mathit{unionDT}, \\
\quad \mathit{structST}, \mathit{unionST} :: \mathit{String} \to \mathit{TFieldList} \to \mathit{TypeExpr} \\
\mathit{structDT}\ \mathit{name}\ \mathit{fields} = \mathit{TStruct}\ \mathit{DynamicStruct}\ \mathit{name}\ \mathit{fields} \\
\mathit{unionDT}\ \mathit{name}\ \mathit{fields} = \mathit{TUnion}\ \mathit{DynamicUnion}\ \mathit{name}\ \mathit{fields} \\
\mathit{structST}\ \mathit{name}\ \mathit{fields} = \mathit{TStruct}\ \mathit{StaticStruct}\ \mathit{name}\ \mathit{fields} \\
\mathit{unionST}\ \mathit{name}\ \mathit{fields} = \mathit{TUnion}\ \mathit{StaticUnion}\ \mathit{name}\ \mathit{fields} \\
\mathit{enumT} :: \mathit{String} \to [(\mathit{String}, \mathit{Int})] \to \mathit{TypeExpr} \\
\mathit{enumT}\ \mathit{name}\ \mathit{fields} = \mathit{TEnum}\ \mathit{name}\ \mathit{fields} \\
\mathit{typedef} :: \mathit{TypeExpr} \to \mathit{String} \to \mathit{TypeExpr} \\
\mathit{typedef}\ \mathit{typ}\ \mathit{name} = \mathit{TTypedef}\ \mathit{typ}\ \mathit{name}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{cptrT} :: \mathit{String} \to \mathit{TypeExpr} \\
\mathit{cptrT}\ \mathit{id} = \mathit{TCompPointer}\ \mathit{id}
\end{array}
$$

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, $\mathit{PureExpr}$, 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

$$
\begin{array}{l}
\mathbf{data}\ \mathit{PureExpr} = \mathit{Void} \\
\quad \mid \mathit{CLInteger}\ \mathit{Signedness}\ \mathit{Size}\ \mathit{Integer} \\
\quad \mid \mathit{CLFloat}\ \mathit{Float} \\
\quad \mid \mathit{CLChar}\ \mathit{Char}
\end{array}
$$

```
        | CLRef Origin TypeExpr VarName
        | Unary UnaryOp PureExpr
        | Binary BinaryOp PureExpr PureExpr
        | Sizeof TypeExpr
        | Test PureExpr PureExpr PureExpr
        | Cast TypeExpr PureExpr
        | Quote String
          deriving (Eq, Show)
```

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.

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

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.

```
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.

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

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

```
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:

```
void :: PureExpr
void = Void

int8, int16, int32, int64 :: Integer → PureExpr
int8 x = CLInteger Signed TInt8 x
int16 x = CLInteger Signed TInt16 x
int32 x = CLInteger Signed TInt32 x
int64 x = CLInteger Signed TInt64 x

uint8, uint16, uint32, uint64 :: Integer → PureExpr
uint8 x = CLInteger Unsigned TInt8 x
uint16 x = CLInteger Unsigned TInt16 x
uint32 x = CLInteger Unsigned TInt32 x
uint64 x = CLInteger Unsigned TInt64 x

charc :: Char → PureExpr
charc x = CLInteger Unsigned TInt8 (toInteger $ ord x)

float :: Float → PureExpr
float x = CLFloat x

cchar :: Char → PureExpr
cchar x = CLChar x


opaque :: TypeExpr → String → PureExpr
opaque t s = CLRef Local t (Provided s)
```

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.

```
(. + .), (. − .), (. ∗ .), (./.), (.%.),
  (. << .), (. >> .), (.&.), (.|.), (.^.),
  (. < .), (. <= .), (. > .),
  (. >= .), (. == .), (.! = .) :: PureExpr → PureExpr → PureExpr
(. + .) = 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
```

$$(. >= .) = Binary\ Geq$$
$$(. == .) = Binary\ Eq$$
$$(.! = .) = Binary\ Neq$$

Finally, $sizeof$, conditionals, and $cast$ have their straightforward alter-ego in FoF:

$$sizeof :: TypeExpr \rightarrow PureExpr$$
$$sizeof\ t = Sizeof\ t$$

$$test :: PureExpr \rightarrow PureExpr \rightarrow PureExpr \rightarrow PureExpr$$
$$test\ c\ ift\ iff = Test\ c\ ift\ iff$$

$$cast :: TypeExpr \rightarrow PureExpr \rightarrow PureExpr$$
$$cast\ t\ e = Cast\ t\ e$$

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.

$$quote :: String \rightarrow PureExpr$$
$$quote\ s = Quote\ s$$

1.2 Filet-o-Fish standard constructs

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

data $FoFConst\ a$

Foreign-call to libc Assert:

$$= Assert\ PureExpr\ a$$

Foreign-call to libc Printf:

$$|\ Printf\ String\ [PureExpr]\ a$$

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:

$\mid$ *Typedef TypeExpr a*
$\mid$ *TypedefE String TypeExpr a*

Support for Structures:

$\mid$ *NewStruct* (*Maybe String*) *AllocStruct String* [(*String*, (*TypeExpr*, *Data*))] (*Loc* $\rightarrow$ *a*)
$\mid$ *ReadStruct Loc String* (*Data* $\rightarrow$ *a*)
$\mid$ *WriteStruct Loc String Data a*

Support for Strings:

$\mid$ *NewString* (*Maybe String*) *String* (*Loc* $\rightarrow$ *a*)

Support for Reference cells:

$\mid$ *NewRef* (*Maybe String*) *Data* (*Loc* $\rightarrow$ *a*)
$\mid$ *ReadRef Loc* (*Data* $\rightarrow$ *a*)
$\mid$ *WriteRef Loc Data a*

Support for Functions:

$\mid$ *NewDef* [*FunAttr*] *String Function TypeExpr* [(*TypeExpr*, *Maybe String*)]
(*PureExpr* $\rightarrow$ *a*)
$\mid$ *CallDef* (*Maybe String*) *PureExpr* [*PureExpr*]
(*PureExpr* $\rightarrow$ *a*)
$\mid$ *Return PureExpr*

Support for Enumerations:

$\mid$ *NewEnum* (*Maybe String*) *String Enumeration String* (*Loc* $\rightarrow$ *a*)

Support for Conditionals:

$\mid$ *If* (*FoFCode PureExpr*)
(*FoFCode PureExpr*)
(*FoFCode PureExpr*) *a*
$\mid$ *For* (*FoFCode PureExpr*)
(*FoFCode PureExpr*)
(*FoFCode PureExpr*)
(*FoFCode PureExpr*) *a*
$\mid$ *While* (*FoFCode PureExpr*)
(*FoFCode PureExpr*) *a*
$\mid$ *DoWhile* (*FoFCode PureExpr*)
(*FoFCode PureExpr*) *a*
$\mid$ *Switch PureExpr*
[(*PureExpr*, *FoFCode PureExpr*)]
(*FoFCode PureExpr*) *a*
$\mid$ *Break*
$\mid$ *Continue*

Support for Arrays:

$\mid$ *NewArray* (*Maybe String*) *AllocArray* [*Data*] (*Loc* $\rightarrow$ *a*)
$\mid$ *ReadArray Loc Index* (*Data* $\rightarrow$ *a*)
$\mid$ *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.

type *Data* = *PureExpr*
type *Loc* = *PureExpr*
type *Index* = *PureExpr*

Function attributes A function can be characterized by the following attributes, following their C semantics:

$$
\begin{aligned}
&\textbf{data}\ \mathit{FunAttr} = \mathit{Static}\\
&\quad \mid \mathit{Inline}\\
&\quad \textbf{deriving}\ (\mathit{Eq})
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Show}\ \mathit{FunAttr}\ \textbf{where}\\
&\quad \mathit{show}\ \mathit{Static} = \texttt{"static"}\\
&\quad \mathit{show}\ \mathit{Inline} = \texttt{"inline"}
\end{aligned}
$$

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

$$\textbf{type}\ \mathit{Enumeration} = [(\mathit{String}, \mathit{Int})]$$

1.2.1 Functor instance

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

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{FoFConst}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{Assert}\ a\ b) = \mathit{Assert}\ a\ (f\ b)\\
&\quad \mathit{fmap}\ f\ (\mathit{Printf}\ a\ b\ c) = \mathit{Printf}\ a\ b\ (f\ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{HasDescendants}\ a\ b\ c) = \mathit{HasDescendants}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{MemToPhys}\ a\ b\ c) = \mathit{MemToPhys}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{GetAddress}\ a\ b\ c) = \mathit{GetAddress}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{NewUnion}\ a\ b\ c\ d\ e\ g) = \mathit{NewUnion}\ a\ b\ c\ d\ e\ (f \circ g)\\
&\quad \mathit{fmap}\ f\ (\mathit{ReadUnion}\ a\ b\ c) = \mathit{ReadUnion}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{WriteUnion}\ a\ b\ c\ d) = \mathit{WriteUnion}\ a\ b\ c\ (f\ d)\\
&\quad \mathit{fmap}\ f\ (\mathit{Typedef}\ a\ c) = \mathit{Typedef}\ a\ (f\ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{TypedefE}\ a\ b\ c) = \mathit{TypedefE}\ a\ b\ (f\ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{NewStruct}\ a\ b\ c\ d\ e) = \mathit{NewStruct}\ a\ b\ c\ d\ (f \circ e)\\
&\quad \mathit{fmap}\ f\ (\mathit{ReadStruct}\ a\ b\ c) = \mathit{ReadStruct}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{WriteStruct}\ a\ b\ c\ d) = \mathit{WriteStruct}\ a\ b\ c\ (f\ d)\\
&\quad \mathit{fmap}\ f\ (\mathit{NewString}\ a\ b\ c) = \mathit{NewString}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{NewRef}\ a\ b\ c) = \mathit{NewRef}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{ReadRef}\ a\ b) = \mathit{ReadRef}\ a\ (f \circ b)\\
&\quad \mathit{fmap}\ f\ (\mathit{WriteRef}\ a\ b\ c) = \mathit{WriteRef}\ a\ b\ (f\ c)\\
&\quad \mathit{fmap}\ g\ (\mathit{NewDef}\ a\ b\ c\ d\ e\ f) = \mathit{NewDef}\ a\ b\ c\ d\ e\ (g \circ f)\\
&\quad \mathit{fmap}\ f\ (\mathit{CallDef}\ a\ b\ c\ d) = \mathit{CallDef}\ a\ b\ c\ (f \circ d)\\
&\quad \mathit{fmap}\ f\ (\mathit{Return}\ a) = \mathit{Return}\ a\\
&\quad \mathit{fmap}\ f\ (\mathit{NewEnum}\ a\ b\ c\ d\ e) = \mathit{NewEnum}\ a\ b\ c\ d\ (f \circ e)\\
&\quad \mathit{fmap}\ f\ (\mathit{If}\ a\ b\ c\ d) = \mathit{If}\ a\ b\ c\ (f\ d)\\
&\quad \mathit{fmap}\ f\ (\mathit{For}\ a\ b\ c\ d\ e) = \mathit{For}\ a\ b\ c\ d\ (f\ e)\\
&\quad \mathit{fmap}\ f\ (\mathit{While}\ a\ b\ c) = \mathit{While}\ a\ b\ (f\ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{DoWhile}\ a\ b\ c) = \mathit{DoWhile}\ a\ b\ (f\ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{Switch}\ a\ b\ c\ d) = \mathit{Switch}\ a\ b\ c\ (f\ d)\\
&\quad \mathit{fmap}\ f\ \mathit{Break} = \mathit{Break}\\
&\quad \mathit{fmap}\ f\ \mathit{Continue} = \mathit{Continue}\\
&\quad \mathit{fmap}\ f\ (\mathit{NewArray}\ a\ b\ c\ d) = \mathit{NewArray}\ a\ b\ c\ (f \circ d)\\
&\quad \mathit{fmap}\ f\ (\mathit{ReadArray}\ a\ b\ c) = \mathit{ReadArray}\ a\ b\ (f \circ c)\\
&\quad \mathit{fmap}\ f\ (\mathit{WriteArray}\ a\ b\ c\ d) = \mathit{WriteArray}\ a\ b\ c\ (f\ d)
\end{aligned}
$$

Thanks to this functor structure, it makes sense to embed $FoFConst$ in a $Semantics$: the machinery we build in Chapter 2.3 will take care of transforming this functor into a free monad. Hence the following type synonym.

$$\textbf{type}\ FoFCode\ a = Semantics\ FoFConst\ a$$

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.

$$
\begin{aligned}
&\mathit{symbEval}\ (\mathit{Unary}\ \mathit{op}\ x) = \\
&\quad \mathit{symbEvalUnary}\ \mathit{op}\ x' \\
&\qquad \textbf{where}\ x' = \mathit{symbEval}\ x \\
&\mathit{symbEval}\ (\mathit{Binary}\ \mathit{op}\ x\ y) = \\
&\quad \mathit{symbEvalBinary}\ \mathit{op}\ x'\ y' \\
&\qquad \textbf{where}\ x' = \mathit{symbEval}\ x \\
&\qquad\quad y' = \mathit{symbEval}\ y \\
&\mathit{symbEval}\ (\mathit{Sizeof}\ \mathit{typ}) = \mathit{symbEvalSizeof}\ \mathit{typ} \\
&\mathit{symbEval}\ (\mathit{Test}\ x\ y\ z) = \\
&\quad \mathit{symbEvalTest}\ x'\ y\ z \\
&\qquad \textbf{where}\ x' = \mathit{symbEval}\ x \\
&\mathit{symbEval}\ (\mathit{Cast}\ t\ x) = \\
&\quad \mathit{symbEvalCast}\ t\ x' \\
&\qquad \textbf{where}\ x' = \mathit{symbEval}\ x
\end{aligned}
$$

Unary Operator Evaluation

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

$$\mathit{symbEvalUnary} :: \mathit{UnaryOp} \rightarrow \mathit{PureExpr} \rightarrow \mathit{PureExpr}$$

Hence the following code:

$$
\begin{aligned}
&\mathit{symbEvalUnary}\ \mathit{Minus}\ x = \\
&\quad \textbf{case}\ x\ \textbf{of} \\
&\qquad \mathit{CLInteger}\ \mathit{Signed}\ \mathit{size}\ x \rightarrow \mathit{CLInteger}\ \mathit{Signed}\ \mathit{size}\ (-x) \\
&\qquad \mathit{CLFloat}\ x \rightarrow \mathit{CLFloat}\ (-x) \\
&\qquad \_ \rightarrow \mathit{error}\ \texttt{"symbEvalUnary: minus on wrong type"} \\
&\mathit{symbEvalUnary}\ \mathit{Complement}\ x = \\
&\quad \textbf{case}\ x\ \textbf{of} \\
&\qquad \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ x \rightarrow \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ (\mathit{complement}\ x) \\
&\qquad \_ \rightarrow \mathit{error}\ \texttt{"symbEvalUnary: complement on wrong type"} \\
&\mathit{symbEvalUnary}\ \mathit{Negation}\ x = \\
&\quad \textbf{case}\ x\ \textbf{of} \\
&\qquad \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ 0 \rightarrow \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ 1 \\
&\qquad \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ \_ \rightarrow \mathit{CLInteger}\ \mathit{sg}\ \mathit{sz}\ 0 \\
&\qquad \_ \rightarrow \mathit{error}\ \texttt{"symbEvalUnary: negation on wrong type"}
\end{aligned}
$$

Binary Operator Evaluation

For binary operators, here is our goal:

$$\mathit{symbEvalBinary} :: \mathit{BinaryOp} \rightarrow \mathit{PureExpr} \rightarrow \mathit{PureExpr} \rightarrow \mathit{PureExpr}$$

Achieved by the following, messy codes.

Arithmetic Operations

$$
\begin{aligned}
&\mathit{symbEvalBinary}\ \mathit{Plus}\ (\mathit{CLInteger}\ \mathit{sg}\ \mathit{si}\ x)\ (\mathit{CLInteger}\ \mathit{sg}'\ \mathit{si}'\ y) \\
&\quad \mid \mathit{sg} \equiv \mathit{sg}' \wedge \mathit{si} \equiv \mathit{si}' = \mathit{CLInteger}\ \mathit{sg}\ \mathit{si}\ (x + y) \\
&\quad \mid \mathit{otherwise} = \mathit{error}\ \texttt{"symbEvalBinary: Plus undefined"} \\
&\mathit{symbEvalBinary}\ \mathit{Plus}\ (\mathit{CLInteger}\ \_\ \_\ x)\ (\mathit{CLFloat}\ y) =
\end{aligned}
$$

```
  CLFloat ((fromRational $ toRational x) + y)
symbEvalBinary Plus (CLFloat x) (CLInteger _ _ y) =
  CLFloat (x + (fromRational $ toRational y))
symbEvalBinary Plus (CLFloat x) (CLFloat y) = CLFloat (x + y)
symbEvalBinary Plus _ _ = error "symbEvalBinary: Plus undefined"
```

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.

```
symbEvalBinary Sub (CLInteger sg si x) (CLInteger sg' si' y)
  | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x − y)
  | otherwise = error "symbEvalBinary: Sub undefined"
symbEvalBinary Sub (CLInteger _ _ x) (CLFloat y) =
  CLFloat ((fromRational $ toRational x) − y)
symbEvalBinary Sub (CLFloat x) (CLInteger _ _ y) =
  CLFloat (x − (fromRational $ toRational y))
symbEvalBinary Sub (CLFloat x) (CLFloat y) = CLFloat (x − y)
symbEvalBinary Sub _ _ = error "symbEvalBinary: Sub undefined"
symbEvalBinary Mul (CLInteger sg si x) (CLInteger sg' si' y)
  | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x * y)
  | otherwise = error "symbEvalBinary: Mul undefined"
symbEvalBinary Mul (CLInteger _ _ x) (CLFloat y) =
  CLFloat ((fromRational $ toRational x) * y)
symbEvalBinary Mul (CLFloat x) (CLInteger _ _ y) =
  CLFloat (x * (fromRational $ toRational y))
symbEvalBinary Mul (CLFloat x) (CLFloat y) = CLFloat (x * y)
symbEvalBinary Mul _ _ = error "symbEvalBinary: Mul 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 Div (CLInteger _ _ x) (CLFloat y) =
  CLFloat ((fromRational $ toRational x) / y)
symbEvalBinary Div (CLFloat x) (CLInteger _ _ y) =
  CLFloat (x / (fromRational $ toRational y))
symbEvalBinary Div (CLFloat x) (CLFloat y) = CLFloat (x / y)
symbEvalBinary Div _ _ = 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"
symbEvalBinary Mod _ _ = error "symbEvalBinary: Mod undefined"
```

Boolean Operations

```
  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 Shl _ _ = 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 Shr _ _ = error "symbEvalBinary: Shr undefined"
symbEvalBinary AndBit (CLInteger sg si x) (CLInteger sg' si' y)
  | sg ≡ sg' ∧ si ≡ si' = CLInteger sg si (x B..|. 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 OrBit _ _ = 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
```

$$symbEvalSizeof\ (TPointer\ \_\ \_) = CLInteger\ Unsigned\ TInt64\ 8$$
$$symbEvalSizeof\ (TCompPointer\ \_) = CLInteger\ Unsigned\ TInt64\ 8$$
$$symbEvalSizeof\ (TArray\ \_\ typ) = CLInteger\ Unsigned\ TInt64\ 8$$
$$symbEvalSizeof\ (TStruct\ \_\ \_\ fields) = CLInteger\ Unsigned\ TInt64\ 8$$
$$symbEvalSizeof\ (TUnion\ \_\ \_\ fields) = CLInteger\ Unsigned\ TInt64\ 8$$

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.

$$symbEvalTest :: PureExpr \to PureExpr \to PureExpr \to PureExpr$$
$$symbEvalTest\ (CLInteger\ \_\ \_\ 0)\ \_\ y = symbEval\ y$$
$$symbEvalTest\ (CLFloat\ 0)\ \_\ y = symbEval\ y$$
$$symbEvalTest\ \_\ x\ \_ = symbEval\ x$$

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.

$$symbEvalCast :: TypeExpr \to PureExpr \to PureExpr$$
$$symbEvalCast\ (TInt\ sg\ sz)\ (CLInteger\ sg'\ sz'\ x)$$
$$\quad |\ sg' < sg \land sz' < sz = CLInteger\ sg\ sz\ x$$
$$\quad |\ otherwise = error\ \texttt{"symbEvalCast: illegal integer cast"}$$
$$symbEvalCast\ TFloat\ (CLInteger\ \_\ \_\ x) =$$
$$\quad CLFloat\ (fromRational\ \$\ toRational\ x)$$
$$symbEvalCast\ TFloat\ vx@(CLFloat\ x) = vx$$
$$symbEvalCast\ \_\ \_ =$$
$$\quad error\ \texttt{"symbEvalCast: Not yet implemented/undefined cast"}$$

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.

$$
\begin{array}{l}
\textbf{data}\ Heap = Hp\ \{freshLoc :: Int, \\
\quad refMap :: [(VarName, Data)], \\
\quad freshSLoc :: Int, \\
\quad strMap :: [(VarName, [(String, Data)])], \\
\quad freshALoc :: Int, \\
\quad arrayMap :: [(VarName, [Data])]\}
\end{array}
$$

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.

$$
\begin{array}{l}
runAlgebra :: FoFConst\ (Heap \to (PureExpr, Heap)) \to Heap \to (PureExpr, Heap) \\
runAlgebra\ x@(NewArray\ \_\ \_\ \_\ \_) = runArrays\ x \\
runAlgebra\ x@(ReadArray\ \_\ \_\ \_) = runArrays\ x \\
runAlgebra\ x@(WriteArray\ \_\ \_\ \_\ \_) = runArrays\ x \\
runAlgebra\ x@(If\ \_\ \_\ \_\ \_) = runConditionals\ x \\
runAlgebra\ x@(For\ \_\ \_\ \_\ \_\ \_) = runConditionals\ x \\
runAlgebra\ x@(While\ \_\ \_\ \_) = runConditionals\ x \\
runAlgebra\ x@(DoWhile\ \_\ \_\ \_) = runConditionals\ x \\
runAlgebra\ x@(Switch\ \_\ \_\ \_\ \_) = runConditionals\ x \\
runAlgebra\ x@Break = runConditionals\ x \\
runAlgebra\ x@Continue = runConditionals\ x \\
runAlgebra\ x@(NewEnum\ \_\ \_\ \_\ \_\ \_) = runEnumerations\ x \\
runAlgebra\ x@(NewDef\ \_\ \_\ \_\ \_\ \_\ \_) = runFunctions\ x \\
runAlgebra\ x@(CallDef\ \_\ \_\ \_\ \_) = runFunctions\ x \\
runAlgebra\ x@(Return\ \_) = runFunctions\ x \\
runAlgebra\ x@(NewRef\ \_\ \_\ \_) = runReferences\ x \\
runAlgebra\ x@(ReadRef\ \_\ \_) = runReferences\ x \\
runAlgebra\ x@(WriteRef\ \_\ \_\ \_) = runReferences\ x \\
runAlgebra\ x@(NewString\ \_\ \_\ \_) = runString\ x \\
runAlgebra\ x@(Typedef\ \_\ \_) = runTypedef\ x \\
runAlgebra\ x@(TypedefE\ \_\ \_\ \_) = runTypedef\ x \\
runAlgebra\ x@(NewStruct\ \_\ \_\ \_\ \_\ \_) = runStructures\ x \\
runAlgebra\ x@(ReadStruct\ \_\ \_\ \_) = runStructures\ x \\
runAlgebra\ x@(WriteStruct\ \_\ \_\ \_\ \_) = runStructures\ x \\
runAlgebra\ x@(NewUnion\ \_\ \_\ \_\ \_\ \_\ \_) = runUnions\ x \\
runAlgebra\ x@(ReadUnion\ \_\ \_\ \_) = runUnions\ x \\
runAlgebra\ x@(WriteUnion\ \_\ \_\ \_\ \_) = runUnions\ x \\
runAlgebra\ x@(Assert\ \_\ \_) = runAssert\ x \\
runAlgebra\ x@(Printf\ \_\ \_\ \_) = runPrintf\ x \\
runAlgebra\ x@(HasDescendants\ \_\ \_\ \_) = runHasDescendants\ x \\
runAlgebra\ x@(MemToPhys\ \_\ \_\ \_) = runMemToPhys\ x \\
runAlgebra\ x@(GetAddress\ \_\ \_\ \_) = runGetAddress\ x
\end{array}
$$

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.

```
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*.

2.3.1 The Semantics Monad

We build a monad *Semantics f* out of a function *f* thanks to the following data-type:

$$\textbf{data}\ \mathit{Semantics}\ f\ a = \mathit{Pure}\ a \\ \quad \mid \mathit{Impure}\ (f\ (\mathit{Semantics}\ f\ a))$$

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

$$\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (\mathit{Semantics}\ f)\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Pure}\ x) = \mathit{Pure}\ (f\ x) \\ \quad \mathit{fmap}\ f\ (\mathit{Impure}\ t) = \mathit{Impure}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ t)$$

We need to (as of GHC 7.10) implement Applicative

$$\textbf{instance}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Applicative}\ (\mathit{Semantics}\ f)\ \textbf{where} \\ \quad \mathit{pure} = \mathit{return} \\ \quad (< * >) = \mathit{ap}$$

Then, we obtain the monad:

$$\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Semantics}\ f)\ \textbf{where} \\ \quad \mathit{return} = \mathit{Pure} \\ \quad (\mathit{Pure}\ x) \gg\!= f = f\ x \\ \quad (\mathit{Impure}\ t) \gg\!= f = \mathit{Impure}\ (\mathit{fmap}\ (\gg\!= f)\ t)$$

Terms are embedded into the monad thanks the following function:

$$\mathit{inject} :: f\ (\mathit{Semantics}\ f\ a) \rightarrow \mathit{Semantics}\ f\ a \\ \mathit{inject}\ x = \mathit{Impure}\ x$$

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 *Pure* terms, which are simply values, from *Impure* ones, which are the embedded constructs.

$$\mathit{foldSemantics} :: \mathit{Functor}\ f \Rightarrow (a \rightarrow b) \rightarrow (f\ b \rightarrow b) \rightarrow \mathit{Semantics}\ f\ a \rightarrow b \\ \mathit{foldSemantics}\ \mathit{pure}\ \mathit{imp}\ (\mathit{Pure}\ x) = \mathit{pure}\ x \\ \mathit{foldSemantics}\ \mathit{pure}\ \mathit{imp}\ (\mathit{Impure}\ t) = \mathit{imp}\ \$\ \mathit{fmap}\ (\mathit{foldSemantics}\ \mathit{pure}\ \mathit{imp})\ t$$

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 *sequence* function in the IO monad:

```
sequenceSem ms = foldr k (return []) ms
  where k m m' =
    do
      x ← m
      xs ← m'
      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:

$$\begin{array}{l} newArray :: [Data] \rightarrow FoFCode\ Loc \\ newArray\ value = inject\ (NewArray\ Nothing\ DynamicArray\ value\ return) \end{array}$$

$$\begin{array}{l} newStaticArray :: [Data] \rightarrow FoFCode\ Loc \\ newStaticArray\ value = inject\ (NewArray\ Nothing\ (StaticArray\ \$\ length\ value)\ value\ return) \end{array}$$

Similarly, they can be named:

$$\begin{array}{l} newArrayN :: String \rightarrow [Data] \rightarrow FoFCode\ Loc \\ newArrayN\ name\ value = inject\ (NewArray\ (Just\ name)\ DynamicArray\ value\ return) \end{array}$$

$$\begin{array}{l} newStaticArrayN :: String \rightarrow [Data] \rightarrow FoFCode\ Loc \\ newStaticArrayN\ name\ value = inject\ (NewArray\ (Just\ name)\ (StaticArray\ \$\ length\ value)\ value\ return) \end{array}$$

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) index 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 binding1 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
```

$$
\begin{array}{l}
\mathit{for}\ \mathit{init}\ \mathit{cond}\ \mathit{incr}\ \mathit{loop} = \\
\quad \mathit{inject}\ (\mathit{For}\ \mathit{init}\ \mathit{cond}\ \mathit{incr}\ \mathit{loop}\ (\mathit{return}\ \mathit{Void})) \\
\mathit{while} :: \mathit{FoFCode}\ \mathit{PureExpr} \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{while}\ \mathit{cond}\ \mathit{loop} = \\
\quad \mathit{inject}\ (\mathit{While}\ \mathit{cond}\ \mathit{loop}\ (\mathit{return}\ \mathit{Void})) \\
\mathit{doWhile} :: \mathit{FoFCode}\ \mathit{PureExpr} \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{doWhile}\ \mathit{loop}\ \mathit{cond} = \\
\quad \mathit{inject}\ (\mathit{DoWhile}\ \mathit{loop}\ \mathit{cond}\ (\mathit{return}\ \mathit{Void})) \\
\mathit{break} :: \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{break} = \mathit{inject}\ \mathit{Break} \\
\mathit{continue} :: \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{continue} = \mathit{inject}\ \mathit{Continue}
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{switch} :: \mathit{PureExpr} \to \\
\quad [(\mathit{PureExpr}, \mathit{FoFCode}\ \mathit{PureExpr})] \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \to \\
\quad \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{switch}\ \mathit{cond}\ \mathit{cases}\ \mathit{defaultCase} = \\
\quad \mathit{inject}\ (\mathit{Switch}\ \mathit{cond}\ \mathit{cases}\ \mathit{defaultCase}\ (\mathit{return}\ \mathit{Void}))
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{compileConditionals}\ (\mathit{If}\ \mathit{condi}\ \mathit{ifTrue}\ \mathit{ifFalse}\ r)\ \mathit{binding} = \\
\quad (\mathit{FIf}\ \mathit{compCond}\ \mathit{compIfTrue}\ \mathit{compIfFalse}\ \mathit{cont}, \\
\qquad \mathit{binding2}) \\
\qquad\quad \textbf{where}\ (\mathit{compCond}, \mathit{binding1}) = \mathit{compileSemtoFoF'}\ \mathit{condi}\ \mathit{binding} \\
\qquad\qquad (\mathit{compIfTrue}, \mathit{binding1'}) = \mathit{compileSemtoFoF'}\ \mathit{ifTrue}\ \mathit{binding1} \\
\qquad\qquad (\mathit{compIfFalse}, \mathit{binding1''}) = \mathit{compileSemtoFoF'}\ \mathit{ifFalse} \\
\qquad\qquad\quad (\mathit{binding1'} \mid - > \mathit{binding1}) \\
\qquad\qquad (\mathit{cont}, \mathit{binding2}) = r\ (\mathit{binding1''} \mid - > \mathit{binding}) \\
\mathit{compileConditionals}\ (\mathit{While}\ \mathit{condW}\ \mathit{loop}\ r)\ \mathit{binding} = \\
\quad (\mathit{FWhile}\ \mathit{compCond}\ \mathit{compLoop}\ \mathit{cont}, \\
\qquad \mathit{binding3}) \\
\qquad\quad \textbf{where}\ (\mathit{compCond}, \mathit{binding1}) = \mathit{compileSemtoFoF'}\ \mathit{condW}\ \mathit{binding} \\
\qquad\qquad (\mathit{compLoop}, \mathit{binding2}) = \mathit{compileSemtoFoF'}\ \mathit{loop}\ \mathit{binding1} \\
\qquad\qquad (\mathit{cont}, \mathit{binding3}) = r\ (\mathit{binding2} \mid - > \mathit{binding}) \\
\mathit{compileConditionals}\ (\mathit{DoWhile}\ \mathit{loop}\ \mathit{condD}\ r)\ \mathit{binding} = \\
\quad (\mathit{FDoWhile}\ \mathit{compLoop}\ \mathit{compCond}\ \mathit{cont}, \\
\qquad \mathit{binding3}) \\
\qquad\quad \textbf{where}\ (\mathit{compLoop}, \mathit{binding1}) = \mathit{compileSemtoFoF'}\ \mathit{loop}\ \mathit{binding} \\
\qquad\qquad (\mathit{compCond}, \mathit{binding2}) = \mathit{compileSemtoFoF'}\ \mathit{condD} \\
\qquad\qquad\quad (\mathit{binding1} \mid - > \mathit{binding})
\end{array}
$$

```
            (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 heap1 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*.

```
newEnum :: String →
  Enumeration →
  String →
  FoFCode PureExpr
newEnum nameEnum fields value =
  inject (NewEnum Nothing nameEnum fields value return)
```

Similarly, *newEnumN* creates a named member of an enumeration.

```
newEnumN :: String →
  String →
  Enumeration →
  String →
  FoFCode PureExpr
newEnumN name nameEnum fields value =
  inject (NewEnum (Just name) name fields value return)
```

3.3.2 Compile Instantiation

A *NewEnum* is compiled as follow.

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

$$
\begin{aligned}
&\mathit{runEnumerations}\ (\mathit{NewEnum}\ \_\ \_\ \mathit{enum}\ \mathit{name}\ r)\ \mathit{heap} = \\
&\quad \mathbf{let}\ \mathit{ref} = \mathit{uint64}\ \$\ \mathit{toInteger}\ \$\ \mathit{fromJust}\ \$\ \mathit{name}\ `\mathit{lookup}`\ \mathit{enum}\ \mathbf{in} \\
&\quad r\ \mathit{ref}\ \mathit{heap}
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{def} :: [\mathit{FunAttr}] \rightarrow \\
&\quad \mathit{String} \rightarrow \\
&\quad ([\mathit{PureExpr}] \rightarrow \mathit{FoFCode}\ \mathit{PureExpr}) \rightarrow \\
&\quad \mathit{TypeExpr} \rightarrow \\
&\quad [(\mathit{TypeExpr}, \mathit{Maybe}\ \mathit{String})] \rightarrow \\
&\quad \mathit{FoFCode}\ \mathit{PureExpr} \\
&\mathit{def}\ \mathit{attr}\ \mathit{name}\ \mathit{fun}\ \mathit{returnT}\ \mathit{argsT} = \\
&\quad \mathit{inject}\ (\mathit{NewDef}\ \mathit{attr}\ \mathit{name}\ (\mathit{Fun}\ \mathit{fun})\ \mathit{returnT}\ \mathit{argsT}\ \mathit{return})
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{call} :: \mathit{PureExpr} \rightarrow [\mathit{PureExpr}] \rightarrow \mathit{FoFCode}\ \mathit{PureExpr} \\
&\mathit{call}\ \mathit{funRef}\ \mathit{params} = \\
&\quad \mathit{inject}\ (\mathit{CallDef}\ \mathit{Nothing}\ \mathit{funRef}\ \mathit{params}\ \mathit{return}) \\
&\mathit{callN} :: \mathit{String} \rightarrow \mathit{PureExpr} \rightarrow [\mathit{PureExpr}] \rightarrow \mathit{FoFCode}\ \mathit{PureExpr} \\
&\mathit{callN}\ \mathit{varName}\ \mathit{funRef}\ \mathit{params} = \\
&\quad \mathit{inject}\ (\mathit{CallDef}\ (\mathit{Just}\ \mathit{varName})\ \mathit{funRef}\ \mathit{params}\ \mathit{return})
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{returnc} :: \mathit{PureExpr} \rightarrow \mathit{FoFCode}\ \mathit{PureExpr} \\
&\mathit{returnc}\ \mathit{value} = \mathit{inject}\ (\mathit{Return}\ \mathit{value})
\end{aligned}
$$

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.

```
compileFunctions (NewDef attr nameF (Fun func) return args r)
  binding =
  (FNewDef attr nameF compBody return instanceArgs cont,
    binding2)
      where instanceArgs = instanciateArgs args
        (compBody, binding1) = compileSemtoFoF' (func instanceArgs) binding
        ref = CLRef 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)) = (CLRef Param typ name) : l
            where name = case mName of
              Just x → Provided x
              Nothing → makeVarName Param idx


compileFunctions (CallDef mName f@(CLRef _ (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 (CLRef Local
                returnT
                (fromJust name))
                binding1
```

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

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

3.4.3 Run Instantiation

As usual, we dispatch here:

$runFunctions\ (NewDef\ \_\ \_\ f\ \_\ \_\ r)\ heap =$
$\quad uncurry\ r\ \$\ runNewDef\ f\ heap$
$runFunctions\ (CallDef\ \_\ a\ b\ r)\ heap =$
$\quad uncurry\ r\ \$\ runCallDef\ a\ b\ heap$
$runFunctions\ (Return\ a)\ heap =$
$\quad runReturn\ a\ heap$ -- OK??

And compute there:

$runReturn :: PureExpr \to Heap \to (PureExpr, Heap)$
$runReturn\ e\ heap = (e, heap)$

$runNewDef :: Function \to Heap \to (PureExpr, Heap)$
$runNewDef\ function\ heap =$
$\quad (CLRef\ Global\ (TFun\ \bot\ function\ \bot\ \bot)\ \bot, heap)$

$runCallDef :: PureExpr \to [PureExpr] \to Heap \to$
$\quad (PureExpr, Heap)$
$runCallDef\ (CLRef\ \_\ (TFun\ \_\ (Fun\ function)\ \_\ \_)\ \_)\ args\ heap =$
$\quad \mathbf{let}\ (result, heap1) = run\ (function\ args)\ heap\ \mathbf{in}$
$\quad (result, heap1)$

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.

$newRef :: Data \to FoFCode\ Loc$
$newRef\ d = inject\ (NewRef\ Nothing\ d\ return)$

$newRefN :: String \to Data \to FoFCode\ Loc$
$newRefN\ name\ d = inject\ (NewRef\ (Just\ name)\ d\ return)$

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

$readRef :: Loc \to FoFCode\ Data$
$readRef\ l = inject\ (ReadRef\ l\ return)$

$writeRef :: Loc \to Data \to FoFCode\ PureExpr$
$writeRef\ l\ d = inject\ (WriteRef\ l\ d\ (return\ Void))$

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.

```
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 _ _ xloc) r) binding =
  (FStatement (FReadRef name ref) cont,
    binding2)
      where (loc, name, binding1) = heritVarName binding xloc
         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:

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

And the per-construct interpreters follow:

```
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 :: String → FoFCode Loc
newString value = inject (NewString Nothing value return)

newStringN :: String → String → FoFCode Loc
newStringN name value = inject (NewString (Just name) value return)
```

3.6.2 Compile Instantiation

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

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

3.6.3 Run Instantiation

Similarly, the interpreter is simple.

```
runString (NewString a b r) heap = uncurry r $ runNewString b heap
```

```
runNewString :: String → Heap → (Loc, Heap)
runNewString string heap =
```

$$
\begin{array}{l}
\textbf{let } \mathit{loc} = \mathit{freshALoc\ heap} \textbf{ in} \\
\textbf{let } \mathit{size} = \mathit{length\ string} \textbf{ in} \\
\textbf{let } \mathit{name} = \mathit{makeVarName\ Dynamic\ loc} \textbf{ in} \\
\textbf{let } \mathit{ref} = \mathit{CLRef\ Dynamic\ (TArray\ (StaticArray\ size)\ TChar)\ name} \textbf{ in} \\
\textbf{let } \mathit{heap1} = \mathit{heap}\ \{\mathit{freshALoc} = \mathit{loc} + 1, \\
\quad \mathit{arrayMap} = (\mathit{name}, \mathit{map\ cchar\ string}) : (\mathit{arrayMap\ heap})\} \textbf{ in} \\
(\mathit{ref}, \mathit{heap1})
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{newStaticStruct} :: \mathit{String} \to \\
\quad [(\mathit{TypeExpr}, \mathit{String}, \mathit{Data})] \to \\
\quad \mathit{FoFCode\ Loc} \\
\mathit{newStaticStruct\ name\ stt} = \\
\quad \mathit{inject\ (NewStruct\ Nothing\ StaticStruct\ name} \\
\qquad (\mathit{map}\ (\lambda(t, n, v) \to (n, (t, v)))\ \mathit{stt}) \\
\qquad \mathit{return}) \\
\mathit{newStaticStructN} :: \mathit{String} \to \\
\quad \mathit{String} \to \\
\quad [(\mathit{TypeExpr}, \mathit{String}, \mathit{Data})] \to \\
\quad \mathit{FoFCode\ Loc} \\
\mathit{newStaticStructN\ nameStr\ name\ stt} = \\
\quad \mathit{inject\ (NewStruct\ (Just\ nameStr)\ StaticStruct\ name} \\
\qquad (\mathit{map}\ (\lambda(t, n, v) \to (n, (t, v)))\ \mathit{stt}) \\
\qquad \mathit{return}) \\
\mathit{newStruct} :: \mathit{String} \to \\
\quad [(\mathit{TypeExpr}, \mathit{String}, \mathit{Data})] \to \\
\quad \mathit{FoFCode\ Loc} \\
\mathit{newStruct\ name\ stt} = \\
\quad \mathit{inject\ (NewStruct\ Nothing\ DynamicStruct\ name} \\
\qquad (\mathit{map}\ (\lambda(t, n, v) \to (n, (t, v)))\ \mathit{stt}) \\
\qquad \mathit{return}) \\
\mathit{newStructN} :: \mathit{String} \to \\
\quad \mathit{String} \to \\
\quad [(\mathit{TypeExpr}, \mathit{String}, \mathit{Data})] \to \\
\quad \mathit{FoFCode\ Loc} \\
\mathit{newStructN\ nameStr\ name\ stt} = \\
\quad \mathit{inject\ (NewStruct\ (Just\ nameStr)\ DynamicStruct\ name} \\
\qquad (\mathit{map}\ (\lambda(t, n, v) \to (n, (t, v)))\ \mathit{stt}) \\
\qquad \mathit{return})
\end{array}
$$

Follow the read and write combinators:

```
readStruct :: Loc → String → FoFCode Data
readStruct l f = inject (ReadStruct l f return)

writeStruct :: Loc → String → Data → FoFCode ()
writeStruct l f d = inject (WriteStruct l f d (return ()))
```

3.7.2 Compile Instantiation

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

```
compileStructures (NewStruct refName allocStruct name fields r) binding =
  (FStatement newS cont,
    binding2)
      where (loc, binding1) = getFreshVar binding
        structName = case refName of
          Just x → Provided x
          Nothing → makeVarName Dynamic loc
        fieldsTypeStr = [(field, typ)
          | (field, (typ, _)) ← fields]
        typeStr = TStruct DynamicStruct name fieldsTypeStr
        ret = CLRef Dynamic typeStr structName
        (cont, binding2) = r ret binding1
        newS = FNewStruct structName allocStruct name fields
compileStructures (ReadStruct ref@(CLRef origin
  typ@(TStruct alloc name fields)
  xloc)
  field r) binding =
  (FStatement readS cont,
    binding2)
      where (loc, varName, binding1) = heritVarName binding xloc
        typeField = fromJust $ field `lookup` fields
        ret = CLRef (allocToOrigin alloc) (readOf typeField) varName
        (cont, binding2) = r ret binding1
        readS = FReadStruct varName ref field
        allocToOrigin StaticStruct = Local
        allocToOrigin DynamicStruct = Dynamic
compileStructures (WriteStruct ref@(CLRef origin
  typ@(TStruct alloc name fields)
  xloc)
  field
  value r) binding =
  (FStatement writeS cont,
    binding1)
      where (cont, binding1) = r binding
        writeS = FWriteStruct ref field value
```

3.7.3 Run Instantiation

The interpreter follows with a dispatcher:

$$
\begin{array}{l}
\mathit{runStructures}\ (\mathit{NewStruct}\ \_\ a\ b\ c\ r)\ \mathit{heap} = \\
\quad \mathit{uncurry}\ r\ \$\ \mathit{runNewStruct}\ a\ b\ c\ \mathit{heap} \\
\mathit{runStructures}\ (\mathit{ReadStruct}\ a\ b\ r)\ \mathit{heap} = \\
\quad \mathit{uncurry}\ r\ \$\ \mathit{runReadStruct}\ a\ b\ \mathit{heap} \\
\mathit{runStructures}\ (\mathit{WriteStruct}\ a\ b\ c\ r)\ \mathit{heap} = \\
\quad r\ \$\ \mathit{runWriteStruct}\ a\ b\ c\ \mathit{heap}
\end{array}
$$

And the per-construct implementation:

$$
\begin{array}{l}
\mathit{runNewStruct} :: \mathit{AllocStruct} \rightarrow \\
\quad \mathit{String} \rightarrow \\
\quad [(\mathit{String}, (\mathit{TypeExpr}, \mathit{Data}))] \rightarrow \\
\quad \mathit{Heap} \rightarrow (\mathit{Loc}, \mathit{Heap}) \\
\mathit{runNewStruct}\ \mathit{alloc}\ \mathit{name}\ \mathit{struct}\ \mathit{heap} = \\
\quad \mathbf{let}\ \mathit{structT} = \mathit{map}\ (\lambda(\mathit{x1}, (\mathit{x2}, \_)) \rightarrow (\mathit{x1}, \mathit{x2}))\ \mathit{struct}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{structD} = \mathit{map}\ (\lambda(\mathit{x1}, (\_, \mathit{x2})) \rightarrow (\mathit{x1}, \mathit{x2}))\ \mathit{struct}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{loc} = \mathit{freshLoc}\ \mathit{heap}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{structs} = \mathit{strMap}\ \mathit{heap}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{varName} = \mathit{makeVarName}\ \mathit{Local}\ \mathit{loc}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{heap1} = \mathit{heap}\ \{\mathit{freshLoc} = \mathit{loc} + 1\}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{heap2} = \mathit{heap1}\ \{\mathit{strMap} = (\mathit{varName}, \mathit{structD}) : \mathit{structs}\}\ \mathbf{in} \\
\quad (\mathit{CLRef}\ \mathit{Local}\ (\mathit{TStruct}\ \mathit{alloc}\ \mathit{name}\ \mathit{structT})\ \mathit{varName}, \mathit{heap2}) \\
\mathit{runReadStruct} :: \mathit{Loc} \rightarrow \mathit{String} \rightarrow \mathit{Heap} \rightarrow (\mathit{Data}, \mathit{Heap}) \\
\mathit{runReadStruct}\ (\mathit{CLRef}\ \_\ \_\ \mathit{location})\ \mathit{field}\ \mathit{heap} = \\
\quad \mathbf{let}\ \mathit{structs} = \mathit{strMap}\ \mathit{heap}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{struct} = \mathit{fromJust}\ \$\ \mathit{location}\ \text{`}\mathit{lookup}\text{`}\ \mathit{structs}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{val} = \mathit{fromJust}\ \$\ \mathit{field}\ \text{`}\mathit{lookup}\text{`}\ \mathit{struct}\ \mathbf{in} \\
\quad (\mathit{val}, \mathit{heap}) \\
\mathit{runWriteStruct} :: \mathit{Loc} \rightarrow \mathit{String} \rightarrow \mathit{Data} \rightarrow \mathit{Heap} \rightarrow \mathit{Heap} \\
\mathit{runWriteStruct}\ (\mathit{CLRef}\ \_\ \_\ \mathit{location})\ \mathit{field}\ \mathit{value}\ \mathit{heap} = \\
\quad \mathbf{let}\ \mathit{structs} = \mathit{strMap}\ \mathit{heap}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{struct} = \mathit{fromJust}\ \$\ \mathit{location}\ \text{`}\mathit{lookup}\text{`}\ \mathit{structs}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{struct1} = (\mathit{field}, \mathit{value}) : \mathit{struct}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{structs1} = (\mathit{location}, \mathit{struct1}) : \mathit{structs}\ \mathbf{in} \\
\quad \mathit{heap}\ \{\mathit{strMap} = \mathit{structs1}\}
\end{array}
$$

3.8 Type Definition

The *Typedef* construct provides a similar service than the C `typedef`.

3.8.1 Smart Constructors

In particular, *Typedef* offers two combinators. The first one, *alias* allows you to locally define a type alias.

$$
\begin{array}{l}
\mathit{alias} :: \mathit{TypeExpr} \rightarrow \mathit{FoFCode}\ \mathit{PureExpr} \\
\mathit{alias}\ \mathit{typedef} = \mathit{inject}\ (\mathit{Typedef}\ \mathit{typedef}\ (\mathit{return}\ \mathit{void}))
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{aliasE} :: \mathit{String} \rightarrow \\
\quad \mathit{TypeExpr} \rightarrow
\end{array}
$$

$$FoFCode\ PureExpr$$
$$aliasE\ incl\ typedef = inject\ (TypedefE\ incl\ typedef\ (return\ void))$$

3.8.2 Compile Instantiation

The compilation to FoF is straightforward:

$$\begin{array}{l}
compileTypedef\ (Typedef\ (TTypedef\ typ\ aliasName)\ r)\ binding = \\
\quad \textbf{let}\ (cont, binding1) = r\ binding\ \textbf{in} \\
\quad (FStatement\ (FTypedef\ typ\ aliasName)\ cont, \\
\quad\quad binding1) \\
compileTypedef\ (TypedefE\ inclDirective\ typeDef@(TTypedef\ typ\ aliasName)\ r)\ binding = \\
\quad \textbf{let}\ (cont, binding1) = r\ binding\ \textbf{in} \\
\quad (FStatement\ (FTypedefE\ inclDirective\ typeDef)\ cont, \\
\quad\quad binding1)
\end{array}$$

3.8.3 Run Instantiation

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

$$\begin{array}{l}
runTypedef\ (Typedef\ _\ r)\ heap = r\ heap \\
runTypedef\ (TypedefE\ _\ _\ r)\ heap = r\ heap
\end{array}$$

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.

$$\begin{array}{l}
newStaticUnion :: String \to \\
\quad [(TypeExpr, String)] \to \\
\quad String \to \\
\quad Data \to \\
\quad FoFCode\ Loc \\
newStaticUnion\ name\ fields\ field\ dat = \\
\quad inject\ (NewUnion\ Nothing\ StaticUnion\ name \\
\quad\quad (map\ (\lambda(s1, s2) \to (s2, s1))\ fields) \\
\quad\quad (field, dat) \\
\quad\quad return) \\
newStaticUnionN :: String \to \\
\quad String \to \\
\quad [(TypeExpr, String)] \to \\
\quad String \to \\
\quad Data \to \\
\quad FoFCode\ Loc
\end{array}$$

```
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.

```
compileUnions (NewUnion refName allocUnion nameU fields (initField, initData) r) binding =
  (FStatement newU cont,
    binding2)
      where typeUnion = TUnion DynamicUnion nameU fields
        (loc, binding1) = getFreshVar binding
        name = case refName of
          Nothing → makeVarName Dynamic loc
          Just x → Provided x
        ret = CLRef Dynamic typeUnion name
        (cont, binding2) = r ret binding1
        newU = FNewUnion name allocUnion nameU fields (initField, initData)
compileUnions (ReadUnion ref@(CLRef _ typeU@(TUnion alloc
```

```
    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

Mortician: Bring out your dead! [clang] ...
Customer: Here's one – nine pence.
Dead person: I'm not dead!
Mortician: What?
Customer: Nothing – here's your nine pence.

Monty Python

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`.

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

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.

```
compilePrintf (Printf format params r) binding =
  let (cont, binding1) = r binding in
  (FStatement (FFFICall "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.

$$runPrintf\ (Printf\ a\ b\ r)\ heap = r\ heap$$

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.

$$assert :: PureExpr \to FoFCode\ PureExpr$$
$$assert\ test = inject\ (Assert\ test\ (return\ Void))$$

4.2.2 Compile Instantiation

The compilation is a direct translation into a foreign function:

$$\begin{array}{l} compileAssert\ (Assert\ e\ r)\ binding = \\ \quad \mathbf{let}\ (cont, binding1) = r\ binding\ \mathbf{in} \\ \quad (FStatement\ (FFFICall\ \texttt{"assert"}\ [e])\ cont, \\ \quad\quad binding1) \end{array}$$

4.2.3 Run Instantiation

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

$$runAssert\ (Assert\ a\ r)\ heap = r\ heap$$

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 libarrelfish 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.

$$has\_descendants :: PureExpr \rightarrow FoFCode\ PureExpr$$
$$has\_descendants\ cte = inject\ (HasDescendants\ Nothing\ cte\ return)$$

$$has\_descendantsN :: String \rightarrow PureExpr \rightarrow FoFCode\ PureExpr$$
$$has\_descendantsN\ name\ cte = inject\ (HasDescendants\ (Just\ name)\ cte\ return)$$

5.1.2 Compile Instanciation

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

```
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.

$runHasDescendants\ (HasDescendants\ \_\ a\ r)\ heap = error$ `"HasDescendants: eval not implemented"`

5.2 Mem To Phys

This construct embeds the libarrelfish 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.

```
mem_to_phys :: PureExpr → FoFCode PureExpr
mem_to_phys cte = inject (MemToPhys Nothing cte return)
mem_to_physN :: String → PureExpr → FoFCode PureExpr
mem_to_physN name cte = inject (MemToPhys (Just name) cte return)
```

5.2.2 Compile Instantiation

Compiling is straightforward: just declare a foreign function.

```
compileMemToPhys (MemToPhys mName arg r) binding =
  let (loc, binding1) = getFreshVar binding in
  let name = case mName of
     Just x → Provided x
     Nothing → makeVarName Local loc in
  let ref = CLRef Local uint64T name in
  let (cont, binding2) = r ref binding1 in
  (FStatement (FFFICall "mem_to_phys" [ref, arg]) cont,
     binding2)
```

5.2.3 Run Instantiation

However, the semantics remains to be defined.

$runMemToPhys\ (MemToPhys\ \_\ a\ r)\ heap = error$ `"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.

$$
\begin{aligned}
&\mathit{compile} :: \mathit{Semantics}\ \mathit{FoFConst}\ \mathit{PureExpr} \rightarrow \mathit{PakaCode}\\
&\mathit{compile}\ \mathit{sem} =\\
&\quad \mathit{optimizePaka}\ \$!\\
&\quad \mathit{compileFoFtoPaka}\ \$!\\
&\quad \mathit{compileSemtoFoF}\ \mathit{sem}
\end{aligned}
$$

First of all, The compiler is provided a value of type $\mathit{Semantics}\ \mathit{FoFConst}\ \mathit{PureExpr}$, 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

- [...] 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

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:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{ILFoF}\\
&\quad = \mathit{FConstant}\ \mathit{PureExpr}\\
&\quad \mid \mathit{FStatement}\ \mathit{FStatement}\ \mathit{ILFoF}\\
&\quad \mid \mathit{FClosing}\ \mathit{FClosing}\\
&\quad \mid \mathit{FNewDef}\ [\mathit{FunAttr}]\ \mathit{String}\ \mathit{ILFoF}\ \mathit{TypeExpr}\ [\mathit{PureExpr}]\ \mathit{ILFoF}\\
&\quad \mid \mathit{FIf}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\\
&\quad \mid \mathit{FFor}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\\
&\quad \mid \mathit{FWhile}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\\
&\quad \mid \mathit{FDoWhile}\ \mathit{ILFoF}\ \mathit{ILFoF}\ \mathit{ILFoF}\\
&\quad \mid \mathit{FSwitch}\ \mathit{PureExpr}\ [(\mathit{PureExpr}, \mathit{ILFoF})]\ \mathit{ILFoF}\ \mathit{ILFoF}
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{FStatement}\\
&\quad = \mathit{FNewUnion}\ \mathit{VarName}\ \mathit{AllocUnion}\ \mathit{String}\ [(\mathit{String}, \mathit{TypeExpr})]\ (\mathit{String}, \mathit{Data})\\
&\quad \mid \mathit{FReadUnion}\ \mathit{VarName}\ \mathit{Loc}\ \mathit{String}\\
&\quad \mid \mathit{FWriteUnion}\ \mathit{Loc}\ \mathit{String}\ \mathit{Data}\\
&\quad \mid \mathit{FTypedef}\ \mathit{TypeExpr}\ \mathit{String}\\
&\quad \mid \mathit{FTypedefE}\ \mathit{String}\ \mathit{TypeExpr}\\
&\quad \mid \mathit{FNewStruct}\ \mathit{VarName}\ \mathit{AllocStruct}\ \mathit{String}\ [(\mathit{String}, (\mathit{TypeExpr}, \mathit{Data}))]\\
&\quad \mid \mathit{FReadStruct}\ \mathit{VarName}\ \mathit{Loc}\ \mathit{String}\\
&\quad \mid \mathit{FWriteStruct}\ \mathit{Loc}\ \mathit{String}\ \mathit{Data}\\
&\quad \mid \mathit{FNewString}\ \mathit{VarName}\ \mathit{String}\\
&\quad \mid \mathit{FNewRef}\ \mathit{VarName}\ \mathit{Data}\\
&\quad \mid \mathit{FReadRef}\ \mathit{VarName}\ \mathit{Loc}\\
&\quad \mid \mathit{FWriteRef}\ \mathit{Loc}\ \mathit{Data}\\
&\quad \mid \mathit{FNewEnum}\ \mathit{VarName}\ \mathit{String}\ \mathit{Enumeration}\ \mathit{String}\\
&\quad \mid \mathit{FNewArray}\ \mathit{VarName}\ \mathit{AllocArray}\ [\mathit{Data}]\\
&\quad \mid \mathit{FReadArray}\ \mathit{VarName}\ \mathit{Loc}\ \mathit{Index}\\
&\quad \mid \mathit{FWriteArray}\ \mathit{Loc}\ \mathit{Index}\ \mathit{Data}\\
&\quad \mid \mathit{FCallDef}\ (\mathit{Maybe}\ \mathit{VarName})\ \mathit{PureExpr}\ [\mathit{PureExpr}]\\
&\quad \mid \mathit{FFFICall}\ \mathit{String}\ [\mathit{PureExpr}]
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{FClosing}\\
&\quad = \mathit{FReturn}\ \mathit{PureExpr}\\
&\quad \mid \mathit{FBreak}\\
&\quad \mid \mathit{FContinue}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{compileSemtoFoF'} :: \mathit{FoFCode}\ \mathit{PureExpr} \to \mathit{Binding} \to (\mathit{ILFoF}, \mathit{Binding})\\
&\mathit{compileSemtoFoF'} = \mathit{foldSemantics}\ \mathit{compilePure}\ \mathit{compileAlgebra}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{compilePure} :: \mathit{PureExpr} \to \mathit{Binding} \to (\mathit{ILFoF}, \mathit{Binding})\\
&\mathit{compilePure}\ x\ \mathit{binding} = (\mathit{FConstant}\ x, \mathit{binding})
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{compileSemtoFoF} :: \mathit{FoFCode}\ \mathit{PureExpr} \to \mathit{ILFoF}\\
&\mathit{compileSemtoFoF}\ \mathit{term} = \mathit{fst}\ \$\ \mathit{compileSemtoFoF'}\ \mathit{term}\ \mathit{emptyBinding}\\
&\quad \textbf{where}\ \mathit{emptyBinding} = \mathit{Binding}\ \{\mathit{freshVar} = 1,\\
&\qquad \mathit{defStructs} = [\,],\\
&\qquad \mathit{defUnions} = [\,],\\
&\qquad \mathit{defEnums} = [\,]\}
\end{aligned}
$$

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:

```
getFreshVar :: Binding → (Int, Binding)
getFreshVar binding = (loc, binding1)
  where loc = freshVar binding
    binding1 = binding {freshVar = loc + 1}
```

Note that a clever implementation would be something of type:

```
better_getFreshVar :: Binding → (Int → Binding → t) → t
better_getFreshVar binding f = ⊥
```

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 *passFreshVar* function that builds a *stableBinding* whose fresh variables are ensured not to clash with the one generated using *upBinding*. Similarly, it carries the structures defined in *upBinding*.

```
passFreshVar :: Binding → Binding → Binding
passFreshVar upBinding stableBinding =
  stableBinding {freshVar = freshVar upBinding,
    defStructs = defStructs upBinding,
    defUnions = defUnions upBinding,
    defEnums = defEnums upBinding}
(|− >) = passFreshVar
```

From variable identifier and an origin, we can later make *VarName*. In a craze of Hungarian naming, the origin dictactes the name of variables.

```
makeVarName :: Origin → Int → VarName
makeVarName orig loc = Generated $ makeVarName' orig loc
  where makeVarName' Local x = "fof_x" ++ show x
    makeVarName' Param x = "fof_y" ++ show x
    makeVarName' Dynamic x = "fof_d" ++ show x
    makeVarName' Global x = "fof_g" ++ show x
```

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

```
heritVarName :: Binding → VarName → (Int, VarName, Binding)
heritVarName binding name = (loc, Inherited loc name, binding1)
  where (loc, binding1) = getFreshVar 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 *readOf* and *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:

$typeOf :: PureExpr \rightarrow TypeExpr$
$typeOf\ (Void) = TVoid$
$typeOf\ (CLInteger\ sign\ size\ \_) = TInt\ sign\ size$
$typeOf\ (CLFloat\ \_) = TFloat$
$typeOf\ (CLRef\ \_\ typ\ \_) = typ$
$typeOf\ (Unary\ \_\ x) = typeOf\ x$

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 $(typeOfx \equiv typeOfy)$ **then**
    $typeOfx$
  else *error* `"typeOf: Binop on distinct types."`
  where $typeOfx = typeOf\ x$
    $typeOfy = typeOf\ y$
$typeOf\ (Test\ \_\ t1\ t2) =$
  if $(typeOft1 \equiv typeOft2)$ **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.

$$readOf :: TypeExpr \to TypeExpr$$
$$readOf\ (TPointer\ typ\ \_) = TPointer\ typ\ Read$$
$$readOf\ x = x$$
$$unfoldPtrType :: PureExpr \to TypeExpr$$
$$unfoldPtrType\ (CLRef\ \_\ (TPointer\ typ\ \_)\ \_) = readOf\ typ$$

liftType: Although our Reference Cell representation abstracts away the distinction between variables and pointers, it has one drawback. A variable is assigned a *TPointer* type, whereas, in C, we will be working one *TPointer*-level below: our reference cell types corresponds to the same C type but one pointer dereference. Hence, we introduce the following lifting function:

$$liftType :: TypeExpr \to TypeExpr$$
$$liftType\ (TPointer\ x\ \_) = x$$
$$liftType\ x = x$$

deref: The *deref* is another operator dealing with the specify of reference cells. In the compiler, we translate the high-level reference cell operators by pointer manipulations and assignment. Therefore, when manipulating a reference cell, we will not interested in its actual content but its address. Hence the following function. Values will manipulated just as usual, by value.

$$deref :: PureExpr \to String$$
$$deref\ (CLRef\ \_\ (TPointer\ \_\ \_)\ \_) = \texttt{"\&"}$$
$$deref\ \_ = \texttt{""}$$

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:

$$run :: Semantics\ FoFConst\ PureExpr \to Heap \to (PureExpr, Heap)$$
$$run = foldSemantics\ (,)\ runAlgebra$$

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).

```
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,
```

$$
\begin{aligned}
&types = Map.empty,\\
&declarations = [\,],\\
&prototypes = Map.empty,\\
&globalVars = [\,],\\
&functions = Map.empty\}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\textbf{data}\ PakaIntra\\
&\quad = PakaIntra\ \{localVars :: Map.Map\ String\ Doc,\\
&\qquad expr :: (Maybe\ PureExpr)\}\\
&\qquad \textbf{deriving}\ Show\\
&emptyIntra = PakaIntra\ \{localVars = Map.empty,\\
&\quad expr = Nothing\}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\textbf{data}\ ILPaka\\
&\quad = PVoid\\
&\quad \mid PClosing\ PakaClosing\\
&\quad \mid PStatement\ PakaStatement\ ILPaka\\
&\quad \mid PIf\ ILPaka\ PureExpr\ ILPaka\ ILPaka\ ILPaka\\
&\quad \mid PWhile\ ILPaka\ PureExpr\ ILPaka\ ILPaka\\
&\quad \mid PDoWhile\ ILPaka\ ILPaka\ PureExpr\ ILPaka\\
&\quad \mid PSwitch\ PureExpr\ [(PureExpr, ILPaka)]\ ILPaka\ ILPaka
\end{aligned}
$$

However, the major specificity of Paka is its definition of a statement: a statement is either an assigment 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.

$$
\begin{aligned}
&\textbf{data}\ PakaStatement\\
&\quad = PAssign\ PakaVarName\ Term\ [PakaVarName]\\
&\quad \mid PInstruction\ Term\ [PakaVarName]
\end{aligned}
$$

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 $t\ ys$.

$$
\textbf{type}\ Term = [Doc] \rightarrow Doc
$$

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".

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.

$$
\begin{array}{l}
\textbf{data}\ \mathit{PakaVarName} \\
\quad = \mathit{Var}\ \mathit{String} \\
\quad \mid \mathit{Ptr}\ \mathit{PakaVarName} \\
\quad \mid \mathit{Deref}\ \mathit{PakaVarName} \\
\quad \mid \mathit{Complex}\ \mathit{PakaVarName} \\
\quad \mid K\ \mathit{PureExpr} \\
\qquad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{PakaClosing} \\
\quad = \mathit{PReturn}\ \mathit{PureExpr} \\
\quad \mid \mathit{PBreak} \\
\quad \mid \mathit{PContinue} \\
\qquad \textbf{deriving}\ \mathit{Show}
\end{array}
$$

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:

$$\textbf{type}\ \mathit{PakaBuilding} = (\mathit{ILPaka} \rightarrow \mathit{ILPaka}, \mathit{PakaCode}, \mathit{PakaIntra})$$

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:

$$f \mathbin{\#} g = \lambda x \rightarrow g\ (f\ x)$$

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:

$$
\begin{array}{l}
\mathit{first} :: (a \rightarrow b) \rightarrow (a, c, d) \rightarrow (b, c, d) \\
\mathit{first}\ f\ (a, b, c) = (f\ a, b, c) \\
\mathit{second} :: (a \rightarrow b) \rightarrow (c, a, d) \rightarrow (c, b, d) \\
\mathit{second}\ f\ (a, b, c) = (a, f\ b, c) \\
\mathit{third} :: (a \rightarrow b) \rightarrow (c, d, a) \rightarrow (c, d, b) \\
\mathit{third}\ f\ (a, b, c) = (a, b, f\ c)
\end{array}
$$

7.2.2 Building $PakaCode$

We can add new C includes:

$$
\begin{array}{l}
include :: String \to PakaBuilding \to PakaBuilding \\
include\ id = second\ \$\ include'\ id \\
\quad \textbf{where}\ include'\ id\ globalEnv \\
\qquad = \textbf{case}\ id\ `Map.lookup`\ incls\ \textbf{of} \\
\qquad\quad Nothing \to globalEnv\ \{includes = Map.insert\ id\ decl\ incls\} \\
\qquad\quad Just\ \_ \to globalEnv \\
\qquad \textbf{where}\ incls = includes\ globalEnv \\
\qquad\quad decl = text\ \texttt{"\#include"} <+> text\ id
\end{array}
$$

We can declare new C types:

$$
\begin{array}{l}
declare :: String \to Doc \to Doc \to PakaBuilding \to PakaBuilding \\
declare\ id\ typ\ decl = second\ \$\ declare'\ id\ typ\ decl \\
\quad \textbf{where}\ declare'\ id\ typ\ decl\ globalEnv = \\
\qquad \textbf{case}\ id\ `Map.lookup`\ typs\ \textbf{of} \\
\qquad\quad Nothing \to globalEnv\ \{declarations = (id, decl) : decls, \\
\qquad\qquad types = Map.insert\ id\ typ\ typs\} \\
\qquad\quad Just\ \_ \to globalEnv \\
\qquad \textbf{where}\ decls = declarations\ globalEnv \\
\qquad\quad typs = types\ globalEnv
\end{array}
$$

We can declare global variables:

$$
\begin{array}{l}
globalVar :: String \to Doc \to PakaBuilding \to PakaBuilding \\
globalVar\ id\ def = second\ \$\ globalVar'\ id\ def \\
\quad \textbf{where}\ globalVar'\ id\ def\ globalEnv = \\
\qquad \textbf{case}\ id\ `lookup`\ vars\ \textbf{of} \\
\qquad\quad Nothing \to globalEnv\ \{globalVars = (id, def) : vars\} \\
\qquad\quad Just\ \_ \to globalEnv \\
\qquad \textbf{where}\ vars = globalVars\ globalEnv
\end{array}
$$

We can add function prototypes:

$$
\begin{array}{l}
prototype :: String \to Doc \to PakaBuilding \to PakaBuilding \\
prototype\ id\ proto = second\ \$\ prototype'\ id\ proto \\
\quad \textbf{where}\ prototype'\ id\ proto\ globalEnv = \\
\qquad \textbf{case}\ id\ `Map.lookup`\ protos\ \textbf{of} \\
\qquad\quad Nothing \to globalEnv\ \{prototypes = Map.insert\ id\ proto\ protos\} \\
\qquad\quad Just\ \_ \to globalEnv \\
\qquad \textbf{where}\ protos = prototypes\ globalEnv
\end{array}
$$

And we can define new functions:

$$
\begin{array}{l}
function :: Doc \to Doc \to String \to Doc \to PakaIntra \to ILPaka \to PakaBuilding \to PakaBuilding \\
function\ returnT\ attrs\ funName\ funArgs\ lEnv\ body = \\
\quad second\ \$\ function'\ returnT\ attrs\ funName\ funArgs\ lEnv\ body \\
\quad \textbf{where}\ function'\ returnT\ attrs\ funName\ funArgs\ lEnv\ body\ gEnv = \\
\qquad \textbf{case}\ funName\ `Map.lookup`\ functions'\ \textbf{of} \\
\qquad\quad Nothing \to gEnv\ \{functions = Map.insert\ funName\ (returnT, attrs, funName, funArgs, lEnv, body)\ function \\
\qquad\quad Just\ \_ \to gEnv \\
\qquad \textbf{where}\ functions' = functions\ gEnv
\end{array}
$$

7.2.3 Building *PakaIntra*

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

```
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*:

```
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.

```
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:

```
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:

```
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
```

```
    where pwhile' cond test loop cont = λc →
      cont $ 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 = λc →
      cont $ 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 = λc →
      cont $ 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' t@(FIf _ _ _ _) = compileFoFtoPakaIf t
compileFoFtoPaka' (FClosing c) = compileFoFtoPakaClosing c
compileFoFtoPaka' t@(FNewDef _ _ _ _ _ _) = compileFoFtoPakaFunDef t
compileFoFtoPaka' t@(FWhile _ _ _) = compileFoFtoPakaWhile t
compileFoFtoPaka' t@(FDoWhile _ _ _) = compileFoFtoPakaDoWhile t
compileFoFtoPaka' t@(FFor _ _ _ _ _) = compileFoFtoPakaFor t
compileFoFtoPaka' t@(FSwitch _ _ _ _) = compileFoFtoPakaSwitch t
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 <+> returnType <+> text funName <> parens functionArgs <> semi)
```

$\# function\ attr\ returnType\ funName\ functionArgs\ lEnv1\ \ cbody$
$\# compileFoFtoPaka'\ k$
$\$ (cont, gEnv1, lEnv)$
$\textbf{where}\ returnType = toC\ returnT$
$attr = hsep\ (map\ (text \circ show)\ funAttrs)$
$functionArgs = buildFunctionArgs\ args$
$buildFunctionArgs\ params = hcat\ \$\ intersperse\ comma\ \$$
$map\ buildFunctionArg\ params$
$buildFunctionArg\ x = toC\ (liftType\ \$\ typeOf\ \ x) < + > toC\ x$
$(cbody_{-}, gEnv1, lEnv1) = compileFoFtoPaka'\ body\ (id, gEnv, emptyIntra)$
$cbody = cbody_{-}\ PVoid$

7.3.2 Compiling Constant

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

$compileFoFtoPakaCst :: PureExpr \to PakaBuilding \to PakaBuilding$
$compileFoFtoPakaCst = constant$

7.3.3 Compiling Closing statements

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

$compileFoFtoPakaClosing :: FClosing \to PakaBuilding \to PakaBuilding$
$compileFoFtoPakaClosing\ (FReturn\ expr) = close\ \$\ PReturn\ expr$
$compileFoFtoPakaClosing\ (FBreak) = close\ PBreak$
$compileFoFtoPakaClosing\ (FContinue) = close\ PContinue$

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 $compileFoFtoPaka'$, as one would expect. Second, to get a $ILPaka$ value out of an $ILPaka \to ILPaka$ continuation k, we call $k\ pVoid$: `void` is the ultimate closing statement, after all. Third, an expression computing a tested value *must* return a pure expression, which we can grab $fromJust\ \$\ expr\ \ intraEnv$. This is an invariant, if not respected $fromJust$ will blow up.

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

$compileFoFtoPakaIf :: ILFoF \to PakaBuilding \to PakaBuilding$
$compileFoFtoPakaIf\ \ (FIf\ \ cond$
$ifTrue$
$ifFalse$
$k)\ (cont, gEnv, lEnv) =$
$pif\ \ ccond\ \ test\ \ cifTrue\ \ cifFalse$
$\#\ second\ \ (const\ \ gEnv3)$

```
  # third (const lEnv3)
  # compileFoFtoPaka' k
  $ (cont, gEnv3, lEnv3)
    where (ccond_, 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) =
  pwhile ccond test cloop
  # second (const gEnv2)
  # third (const lEnv2)
  # compileFoFtoPaka' k
  $ (cont, gEnv2, lEnv2)
    where (ccond_, 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 (ccond_, 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 ccases 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 (fcase_, gEnv1, lEnv1) = compileFoFtoPaka' fcase (id, gEnv, lEnv)
      cfcase = fcase_ 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 *Term*s 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 $(\lambda[xs, \_, xss] \rightarrow ...)$, 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)
  # assgn pvarName (λ[_, e] → toC varName <+> char '=' <+> e <> semi)
    [pakaVarName dat]
  # compileFoFtoPaka' k
```

```
    where mvarName = mkPakaVarName varName
      pvarName = Var $ mkPakaVarName varName
compileFoFtoPakaStmt (FReadRef varName ref) k =
  localVar mvarName (toC (unfoldPtrType ref) <+> toC varName <> semi)
  # assgn pvarName (λ[_, e] →
    toC varName <+> char '=' <+> e <> semi)
    [pakaValName ref]
  # compileFoFtoPaka' k
    where mvarName = mkPakaVarName varName
      pvarName = Var $ mkPakaVarName varName
compileFoFtoPakaStmt (FWriteRef ref dat) k =
  assgn (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
        assgn pvarName (λ[_, _] →
          toC varName <+> char '='
          <+> toC xloc <> brackets (toC index) <> semi)
            [Complex $ Var $ mkPakaVarName xloc]
```

```
        else
          instr (\_ →
            text "assert" <> parens (text "! \"ReadArray: Out of bound\"") <> semi)
            []
      _ →
        assgn pvarName (λ[_, _, e] →
          text "if" < + > parens (e
            < + > char '<'
            < + > int size) <> lbrace
          $ + $
          nest 4 (toC varName < + > char '='
                  < + > toC xloc <> brackets e <> semi)
          $ + $
          rbrace < + > text "else" < + > lbrace $ + $
            nest 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 (λ[_, e, f] →
      text "if" < + > parens (f < + > char '<' < + > int size) <> lbrace
      $ + $ nest 4 (toC xloc <> brackets f
        < + > char '=' < + > e <> semi)
      $ + $ rbrace < + > text "else" < + > lbrace
      $ + $ nest 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:

```
compileFoFtoPakaStmt (FCallDef mVarName
  (CLRef _ (TFun nameF
    func
    returnT
    argsT) _)
  args) k =
  case mVarName of
    Nothing →
      instr (\_ →
        text nameF
         <> parens (hcat $ intersperse comma $ map toC args) <> semi)
        (map (Complex ∘ pakaVarName) args)
    Just varName →
      localVar (mkPakaVarName varName)
          (toC returnT <+> toC varName <> semi)
       # assgn (Var $ mkPakaVarName varName)
        (\_ → toC varName <+> char '='
            <+> text nameF
            <> parens (hcat $ intersperse comma $ map toC args) <> semi)
        (map (Complex ∘ pakaValName) args)
   # compileFoFtoPaka′ k
```

Compiling Enumerations

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 <> s
   # assgn varName (λ[_] →
    toC name <+> char '=' <+>
      parens (text "union" <+> text nameUnion <> char '*')
         <+> text "malloc" <> parens (
           text "sizeof" <> parens (
           text "union" <+> text nameUnion))
         <> semi) []
```

```
    # assgn varName (λ[_, b] →
      toC name <> text "->" <> text initField
         <+> char '=' <+> b <> semi)
      [pakaVarName initData]
    # compileFoFtoPaka' k
      where varName = Var $ mkPakaVarName name
compileFoFtoPakaStmt (FNewUnion name StaticUnion nameUnion fields (initField, initData)) k =
  declareRecursive (TUnion StaticUnion nameUnion fields)
    # localVar (mkPakaVarName name) (text "union" <+> text nameUnion <+> toC name <> semi)
    # assgn varName (λ[_, e] →
      toC name <> char '.' <> text initField
         <+> char '=' <+> e <> semi)
      [pakaVarName initData]
    # compileFoFtoPaka' k
      where varName = Var $ mkPakaVarName name
compileFoFtoPakaStmt (FReadUnion varName
  (CLRef _ typeU@(TUnion alloc
      nameU
      fields)
                    xloc)
  field) k =
  declareRecursive typeU
    # localVar mpVarName (toC typeField <+> toC varName <> semi)
    # assgn pVarName (λ[_, _] →
      toC varName
      <+> char '='
      <+> toC xloc <> ptrSigUnion alloc <> text field <> semi)
      [Complex $ Var $ mkPakaVarName xloc]
    # compileFoFtoPaka' k
      where typeField = fromJust $ field `lookup` fields
        mpVarName = mkPakaVarName varName
        pVarName = Var $ mkPakaVarName varName
compileFoFtoPakaStmt (FWriteUnion (CLRef origin
  typeU@(TUnion alloc
      nameU
      fields)
  xloc)
  field
  value) k =
  declareRecursive typeU
    # assgn pxloc (λ[_, e] →
      toC xloc <> ptrSigUnion alloc <> text field
      <+> char '=' <+> e <> semi)
      [pakaVarName value]
    # compileFoFtoPaka' k
      where pxloc = Var $ mkPakaVarName xloc
```

Compiling Structs

Quite the same goes for structure operations:

```
compileFoFtoPakaStmt (FNewStruct varName
  DynamicStruct
  nameStruct
```

```
  fields) k =
  declareRecursive (TStruct DynamicStruct nameStruct fieldsTypeStr)
   # localVar mVarName (text "struct" <+> text nameStruct <+> toC varName <> semi)
   # (assgn pVarName (λ[_] →
      toC varName <+> char '='
       <+> parens (text "struct" <+> text nameStruct <+> char '*')
       <+> text "malloc"
       <> parens (text "sizeof"
       <> parens (text "struct" <+> text nameStruct))
       <> semi) [])
     # foldl' (#) id [assgn pVarName (λ[_, e] →
             toC varName <> text "->" <> text field
              <+> char '='
              <+> e <> semi) [pakaVarName val]
       | (field, (typ, val)) ← fields]
      where mVarName = mkPakaVarName varName
        pVarName = Var $ mkPakaVarName varName
        fieldsTypeStr = [(field, typ)
            | (field, (typ, _)) ← fields]
compileFoFtoPakaStmt (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
        fieldsTypeStr = [(field, typ)
            | (field, (typ, _)) ← fields]
compileFoFtoPakaStmt (FReadStruct varName
  ref@(CLRef origin
    typeS@(TStruct alloc
      nameStruct
      fields)
    xloc)
  field) k =
  declareRecursive typeS
   # localVar mvarName (toC typeField <+> toC varName <> semi)
   # assgn 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
compileFoFtoPakaStmt (FWriteStruct ref@(CLRef origin
  typeS@(TStruct alloc
    nameStruct
```

```
      fields)
    xloc)
    field
    value) k =
    declareRecursive typeS
    # assgn pxloc (λ[_, e] →
      toC xloc <> ptrSigStruct alloc <> text field
        < + > char '=' < + > e <> semi)
      [pakaVarName value]
    # compileFoFtoPaka′ k
      where pxloc = Var $ mkPakaVarName xloc
```

Compiling Typedef

And we can even get `typedefs`:

```
compileFoFtoPakaStmt (FTypedef typ aliasName) k =
  declareRecursive typ
  # declare aliasName Pprinter.empty
    (text "typedef" < + > toC typ < + > text aliasName <> semi)
  # compileFoFtoPaka′ k
compileFoFtoPakaStmt (FTypedefE inclDirective
  (TTypedef typ aliasName)) k =
  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.

```
compileFoFtoPakaStmt (FFFICall nameCall args) k =
  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.

```
compileFFI nameCall params | nameCall ≡ "printf" =
  include "<stdio.h>"
  # instr (\_ → text "printf" <> parens (hcat (punctuate comma (map toC params))) <> semi)
    (map (Complex ∘ pakaVarName) params)
compileFFI nameCall [e] | nameCall ≡ "assert" =
  include "<assert.h>"
  # instr (λ[e] → text "assert" <> parens e <> semi) [pakaValName e]
compileFFI nameCall [varName, param] | nameCall ≡ "has_descendants" =
  include "<mdb/mdb.h>"
  # include "<capabilities.h>"
  # include "<stdbool.h>"
  # localVar (show $ toC varName)
    (text "bool" < + > toC varName <> semi)
```

```
    # assgn (pakaValName $ varName)
      (λ[_, e] →
        toC varName <+> char '='
         <+> text "has_descendants"
         <> parens e <> semi)
      [pakaValName $ param]
      -- XXX: mem_to_phys was renamed to mem_to_local_phys.
      -- This is a temporary hack till we get around to producing
      -- a whole list of translation functions here. -Akhi
      -- XXX: moved include to hamlet file compilation so that user version of
      -- cap_predicates can be built -Ross
compileFFI nameCall [varName, param] | nameCall ≡ "mem_to_phys" =
  localVar (show $ toC varName)
    (toC uint64T <+> toC varName <> semi)
    # assgn (pakaValName $ varName)
      (λ[_, e] →
        toC varName <+> char '=' <+>
        text "mem_to_local_phys" <> parens (toC param) <> semi)
      [pakaValName $ param]


compileFFI nameCall [varName, param] | nameCall ≡ "get_address" =
  localVar (show $ toC varName)
    (toC uint64T <+> toC varName <> semi)
    # assgn (pakaValName $ varName)
      (λ[_, e] →
        toC varName <+> char '=' <+>
        text "get_address" <> parens (toC param) <> semi)
      [pakaValName $ param]
```

Declaring types

Above, we have dealt with the compilation of operations on complex structures, such as `enum`s, `struct`s, and `union`s. 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`:

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

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

```
declareEnum nameEnum fields =
  declare nameEnum empty
    (text "enum" <+> text nameEnum <+> lbrace
      $+$ nest 4 (vcat' $ punctuate comma
        ([text name <+> char '=' <+> int val
          | (name, val) ← fields]))
      $+$ 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 *declareRecursive*:

```
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
```

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.

```
ptrSigUnion :: AllocUnion → Doc
ptrSigUnion DynamicUnion = text "->"
ptrSigUnion StaticUnion = char '.'

ptrSigStruct :: AllocStruct → Doc
ptrSigStruct DynamicStruct = text "->"
ptrSigStruct StaticStruct = char '.'
```

7.4 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.

7.4.1 Printing types and expressions

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

```
class Compileable a where
  toC :: a → Doc
```

Part of the *Compileable* class are FoF's types *TypeExpr* and FoF's pure expressions *PureExpr*.

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

```
instance Compileable PureExpr where
  toC (Quote s) = doubleQuotes $ text s
```

```haskell
    toC Void = empty
    toC (CLInteger _ _ x) = integer x
    toC (CLFloat x) = Pprinter.float x
    toC (CLRef origin (TPointer _ Avail) loc) = toC loc
    toC (CLRef origin (TPointer _ Read) loc) = char '*' <> toC loc
    toC (CLRef origin _ loc) = 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:

```haskell
  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 (TPointer 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
```

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.

```
vcat' :: [Doc] → Doc
vcat' [] = empty
vcat' (x : xs) = l ‘seq‘ r ‘seq‘ r
  where l = vcat' xs
    r = x $+$ 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:

```
pprintClosing :: PakaClosing → Doc
pprintClosing (PReturn e) = text "return" <+> parens (toC e) <> semi
pprintClosing PBreak = text "break"
pprintClosing PContinue = text "continue"
```

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

```
pprintStmt :: PakaStatement → Doc
pprintStmt (PAssign dst x srcs) = x (toC dst : map toC srcs)
pprintStmt (PInstruction x srcs) = x (map toC srcs)
```

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

```
pprintPaka :: ILPaka → Doc
pprintPaka PVoid = empty
pprintPaka (PClosing c) = pprintClosing c
pprintPaka (PStatement stmt k) =
  pprintStmt stmt $+$
  pprintPaka k
pprintPaka (PIf cond test ifTrue ifFalse k) =
  pprintPaka cond $+$
  text "if" <+> parens (toC test) <> lbrace $+$
    (nest 4 $! pprintPaka ifTrue) $+$
  rbrace <+> text "else" <+> lbrace $+$
    (nest 4 $! pprintPaka ifFalse) $+$
  rbrace $+$
  pprintPaka k
pprintPaka (PWhile cond test loop k) =
  pprintPaka cond $+$
  text "while" <> parens (toC test) <> lbrace $+$
    (nest 4 $! pprintPaka loop) $+$
  rbrace $+$
  pprintPaka k
pprintPaka (PDoWhile loop cond test k) =
  text "do" <+> lbrace $+$
    (nest 4 $! pprintPaka loop) $+$
  rbrace <+> text "while" <+> parens (toC test) <> semi $+$
  pprintPaka k
pprintPaka (PSwitch test cases defaultCase k) =
  text "switch" <+> parens (toC test) <+> lbrace $+$
```

```
        (nest 4 $ vcat' $ map compileCase cases) $+$
        (nest 4 (text "default:" <+> lbrace $+$
            (nest 4 $! pprintPaka defaultCase) $+$
            rbrace)) $+$
      rbrace $+$
      pprintPaka 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 other 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 (λy → 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 $ pprintPaka body) $+$
            rbrace $+$
            space)
            $ extractM
            $ functions code) $+$
```

$space)$
$\$+\$ space$

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.

$$
\begin{array}{l}
extractL :: Eq\ a \Rightarrow [(a, b)] \rightarrow [b] \\
extractL = (map\ snd) \circ \\
\quad reverse \\
extractM :: Map.Map\ a\ b \rightarrow [b] \\
extractM = Map.elems
\end{array}
$$

Because we have worked very hard, we are rewarded by the right to instantiate these $PakaCode$ in the $Show$.

$$
\begin{array}{l}
\textbf{instance}\ Show\ PakaCode\ \textbf{where} \\
\quad show = render \circ pprint
\end{array}
$$

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:

$$
\begin{array}{l}
optimizePaka :: PakaCode \rightarrow PakaCode \\
optimizePaka = optimizeAssgmtElim
\end{array}
$$

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

$$
\begin{array}{l}
optimizeAssgmtElim :: PakaCode \rightarrow PakaCode \\
optimizeAssgmtElim\ code = code\ \{functions = optFunc\} \\
\quad \textbf{where}\ funcs = functions\ code \\
\qquad optFunc = Map.mapMaybe\ (\lambda(b, c, d, e, f, fun) \rightarrow Just\ (b, c, d, e, f, assgmtElim\ fun))\ funcs
\end{array}
$$

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 $replace\ (Var\ x)\ (Var\ y)\ k$ –, that we are able to say if a variable y is either a constant or never used in a code k – using $isUsed\ flatten\ y\ k$ –, and that we are able to say if a variable x is used without side-effects in a code k – using $isUsed\ flattenS\ x\ k$.

The intra-procedural optimizer will turn an $ILPaka$ into a better $ILPaka$:

$$assgmtElim :: ILPaka \rightarrow ILPaka$$

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 $isUsed$ to allow the definition of more fine-grained predicates, such as "is overwritten".

$$
\begin{array}{l}
assgmtElim\ (PStatement\ a@(PAssign\ (Var\ x)\ \_\ [Var\ y])\ k) = \\
\quad \textbf{if}\ (\neg\ (isUsed\ flatten\ y\ k)) \\
\qquad \wedge\ (\neg\ (isUsed\ flattenS\ x\ k))\ \textbf{then} \\
\qquad\quad assgmtElim\ \$\ replace\ (Var\ x)\ (Var\ y)\ k \\
\quad \textbf{else} \\
\qquad\quad PStatement\ a\ \$ \\
\qquad\quad assgmtElim\ k
\end{array}
$$

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

$$
\begin{array}{l}
assgmtElim\ (PStatement\ a\ k) = \\
\quad PStatement\ a\ \$ \\
\quad assgmtElim\ k
\end{array}
$$

Finally, control-flow operators are simply iterated over:

$$
\begin{array}{l}
assgmtElim\ (PIf\ c\ t\ ifT\ ifF\ k) = \\
\quad PIf \\
\quad (assgmtElim\ c) \\
\quad t \\
\quad (assgmtElim\ ifT) \\
\quad (assgmtElim\ ifF) \\
\quad (assgmtElim\ k) \\
assgmtElim\ (PWhile\ c\ t\ l\ k) = \\
\quad PWhile \\
\quad (assgmtElim\ c) \\
\quad t \\
\quad (assgmtElim\ l) \\
\quad (assgmtElim\ k) \\
assgmtElim\ (PDoWhile\ l\ c\ t\ k) = \\
\quad PDoWhile \\
\quad (assgmtElim\ l) \\
\quad (assgmtElim\ c) \\
\quad t \\
\quad (assgmtElim\ k) \\
assgmtElim\ (PSwitch\ t\ cases\ d\ k) = \\
\quad PSwitch \\
\quad t \\
\quad (map\ (\lambda(a, b) \rightarrow (a, assgmtElim\ b))\ cases) \\
\quad (assgmtElim\ d) \\
\quad (assgmtElim\ k) \\
assgmtElim\ x = x
\end{array}
$$

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.

$isUsed\ f\ x\ k$ 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, $isUsed\ flatten$ will catch any kind of use or def. $isUsed\ flattenS$ 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 $PStatement$.

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

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

$$
\begin{array}{l}
flatten :: PakaVarName \rightarrow Maybe\ String \\
flatten\ (Var\ s) = Just\ \$\ s \\
flatten\ (Ptr\ x) = flatten\ x \\
flatten\ (Deref\ x) = flatten\ x \\
flatten\ (Complex\ x) = flatten\ x \\
flatten\ (K\ \_) = Nothing \\
flattenS :: PakaVarName \rightarrow Maybe\ String \\
flattenS\ (Var\ s) = Nothing \\
flattenS\ (Ptr\ x) = Nothing \\
flattenS\ (Deref\ x) = Nothing \\
flattenS\ (Complex\ x) = flatten\ x \\
flattenS\ (K\ \_) = Nothing
\end{array}
$$

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

[1] National Hurricane Center. Retired hurricane names since 1954.

[2] Edward Kmett. Monads for free, 2008.

[3] George C. Necula. Translation validation for an optimizing compiler. 35, 2000.

[4] Norman Ramsey, John Dias, and Simon Peyton Jones. Hoopl: Dataflow optimization made simple, 2009.

[5] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *1st Workshop on Managed Multi-Core Systems*, June 2008.

[6] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, Forthcoming(-1):1–14, 2008.