Typed Tagless Final Interpreters

Oleg Kiselyov

oleg@okmij.org

Abstract. The so-called 'typed tagless final' approach of Carette et al. [6] has collected and polished a number of techniques for representing typed higher-order languages in a typed metalanguage, along with type-preserving interpretation, compilation and partial evaluation. The approach is an alternative to the traditional, or 'initial' encoding of an object language as a (generalized) algebraic data type. Both approaches permit multiple interpretations of an expression, to evaluate it, pretty-print, etc. The final encoding represents all and only typed object terms without resorting to generalized algebraic data types, dependent or other fancy types. The final encoding lets us add new language forms and interpretations without breaking the existing terms and interpreters.
These lecture notes introduce the final approach slowly and in detail, highlighting extensibility, the solution to the expression problem, and the seemingly impossible pattern-matching. We develop the approach further, to type-safe cast, run-time-type representation, Dynamics, and type reconstruction. We finish with telling examples of type-directed partial evaluation and encodings of type-and-effect systems and linear lambda-calculus.

1 Introduction

One reinvents generic programming when writing accumulation, pretty-printing, equality comparison functions for data types – and writing these functions again and again for extended or slightly different data types. Generic programming aims to relieve the tedium by making programs more applicable, abstracting over values, shapes, processing strategies and so on. One may dually view a data type as an encoding of a domain-specific language, and data type processing as an interpretation of that language. That view comes to the fore if the data type indeed represents an abstract syntax tree (AST). Generic programming then is writing extensible interpreters. The embedded-language point-of-view highlights that oftentimes not all sentences generated by a context-free grammar – not all values fitting the datatype declaration – are regarded as meaningful. A type system is a common way of stating the additional validity constraints. Typed extensible interpreters of typed languages, fitting the theme of the school, express both generic programming (parametrization over interpretations) and indexed programming (expressing processing invariants and validity constraints).

There are two basic approaches to embedding languages and writing their interpreters, which we shall call, somewhat informally, 'initial' and 'final'. The initial approach represents a term of an object language as a value of an algebraic

data type in the metalanguage; interpreters recursively traverse the values deconstructing them by pattern-matching. In the final approach, object language terms are represented as expressions built from a small set of combinators, which are ordinary functions rather than data constructors. The values of these expressions give denotations of the corresponding object terms. An object term is hence represented not by its abstract syntax but by its denotation in a semantic domain. Abstracting over the domain gives us a family of interpretations.

The most noticeable advantages of the final approach are seen in the encoding of typed object languages. The final approach expresses object language types as metalanguage types, without any type tagging and its accompanying overhead. The metalanguage type checker not only checks object types but can also infer them. We can therefore statically ascertain that an object language interpreter preserves object types and does not get stuck: The soundness of the metalanguage's type system entails the soundness of the object language's type system. As a consequence, the final approach easily solves otherwise arduous problems of writing assuredly type-preserving partial evaluators and continuation-passing style transformers. The final approach also turns out extensible, helping solve the so-called expression problem, – and hence useful also for unityped object languages.

The final approach [6] collects and polishes many old ideas, starting from Reynolds [35] and its further development as final algebra specifications in [23, 41]. The approach relies on Yang's [49] encoding of type-indexed values and its generalization as the TypeCase pattern [30], and Thiemann's [38] deforestation of syntax constructors. These techniques require just a Hindley-Milner type system with higher-order polymorphism, as realized in all variants of ML (as functors) and Haskell (as constructor classes).

In this course we explain and develop the final approach, detailing and extending the original presentation [6]. We discuss the duality with the initial approach, and the similarity and differences of the final and Church encodings. We concentrate on operations that at first blush seem impossible in the tagless final approach, such as pattern-matching, binary and other operations that do not seem to be expressible as folds over a term.

The roadmap We start slowly, by introducing the final approach in §2 on a overly simplistic example of a first-order unityped language, corresponding to the ordinary algebraic data type. We introduce the approach together with the more common initial embedding in §2.1 and discuss extensibility and the expression problem in §2.2. The simplicity of the running example, albeit excessive, does help to demonstrate the subtle aspects such as pattern-matching §2.4 and similar non-compositional interpretations, and to explain the solution to the open deserialization problem in §2.3.

§3 makes the proper introduction, for typed higher-order languages, or data types with binders. We dive into the complexities of interpreting typed languages ensuring type preservation, and emerge with surprisingly lucid solutions. The

attributes 'typed' and 'tagless' in the title of the course will finally be explained in §3.1.

Having done all the introductions, we will have real fun in §4, sweeping through type-safe cast, type checking, parametrization over the evaluation order, typed CPS transformation, typed formatting and type-directed partial evaluation. We touch upon the languages with fancy type systems, with effect, §4.2, and linear, §4.3, types.

Throughout the course we use Haskell as our implementation language (meta-language); familiarity with Haskell is assumed. The complete code for the course is available online: `http://okmij.org/ftp/tagless-final/course/`

Main ideas Throughout the course we shall hear the refrain of several main ideas:

- Multiple interpretations: writing a term once and interpreting it many times, in standard and non-standard ways;
- Extensibility: adding more interpreters and enriching the language with more syntactic forms – solving the expression problem;
- Types: to specify interpreters and their logic, to delineate valid object terms;
- Finality: preferring lower-case, functions over constructors, elimination over introduction, denotational over operational.

2 Interpreters for first-order languages

This warm-up section deals with first-order, unityped languages. We will have to wait until §3 for the appearance of the 'tagless typed'. The simplification in this section, however drastic, does help introduce pattern-matching on final terms, extensibility, and the solution to the expression problem.

2.1 Initial and Final embeddings

We start with a very simple language, to be extended later. The language has integer literals, negation and addition. Here are sample expressions of the language: $-(1 + 2)$ and $8 + (- (1+ 2))$. The latter is our running example.

The *initial* embedding of the language in Haskell encodes the expressions of the language as the values of an algebraic data type:

```
data Exp = Lit Int
         | Neg Exp
         | Add Exp Exp
```

Our running example is written as follows:

```
ti1  = Add (Lit 8) (Neg (Add (Lit 1) (Lit  2)))
```

The first interpreter of the language is an evaluator, which proceeds by case analysis, that is, pattern-matching:

```
eval :: Exp → Int
eval (Lit n)     = n
eval (Neg e)     = − eval e
eval (Add e1 e2) = eval e1 + eval e2
```

Evaluating our sample expression, eval ti1, gives the result 5.

We can embed our language in Haskell differently. If all we ever need is the value of an expression, we can represent the term in our arithmetic language by its value, or by a Haskell expression that computes that value. We introduce a representation type for the meaning of an expression, Int, and the functions computing the meaning of the three expression forms of the language (literals, negation and addition).

```
type Repr = Int

lit :: Int → Repr
lit n = n

neg :: Repr → Repr
neg e = − e

add :: Repr → Repr → Repr
add e1 e2 = e1 + e2
```

The computation is *compositional:* the meaning of, for example, addition is computed from the meaning, the value of, the summands. We see the first intimation of the denotational semantics, with further to come. Our running example has the form

```
tf1 = add (lit 8) (neg (add (lit 1) (lit 2)))
```

which is a Haskell expression with the value 5. We will call this second, meta-circular embedding a *final* embedding. It does appear to be dual to the initial embedding: tf1 is strikingly similar to ti1, differing only in the case of the identifiers.

The initial embedding seems more general however, permitting other interpretations, for example, pretty-printing:

```
view:: Exp → String
view (Lit n) = show n
view (Neg e) = "(−" ++ view e ++ ")"
view (Add e1 e2) = "(" ++ view e1 ++ "␣+␣" ++ view e2 ++ ")"
```

The view interpreter 'evaluates' the very same term ti1 to the string "(8 + (-(1 + 2)))" rather than to an integer. In the final encoding, the evaluator is hard-wired into the representation tf1, making it impossible to interpret the object term as something other than integer. We want the final embedding to permit multiple interpretations too; we must therefore find a way to parameterize the 'constructor functions' such as lit and neg by the result type.

Haskell has just the right tool for such a parametrization: a type class.

```
class ExpSYM repr where
  lit :: Int  → repr
  neg :: repr → repr
  add :: repr → repr → repr
```

The constructor functions lit, neg and add have essentially the same signatures as before; the repr type however is in lower-case since it is now variable. The declaration of ExpSYM should remind us even more of the denotational semantics [46], over the semantic domain repr. As befits denotational semantics, the meaning of an expression, whatever repr happens to be, is computed from the meanings of the components. The running example has the same form

```
tf1 = add (lit 8) (neg (add (lit 1) (lit 2)))
```

but the inferred type (provided we disable Haskell's monomorphism restriction) is different. It no longer Repr (that is, Int). Rather, it is ExpSYM repr ⇒ repr, polymorphic over the semantic domain. An object term is represented not by its abstract syntax but by its meaning, denotation in a semantic domain.

To interpret finally-encoded expressions, we write an instance for ExpSYM, specifying the semantic domain. For example, we may interpret expressions as integers

```
instance ExpSYM Int where
  lit n      = n
  neg e      = - e
  add e1 e2 = e1 + e2
```

mapping object language integers to Haskell integers, and object language negation to Haskell negation. We write the evaluator as

```
eval :: Int → Int
eval = id
```

so that eval tf1 has the value 5. The function eval has a strange type for an evaluator, and an even stranger definition – the identity function. It is more proper to call eval a selector of an interpretation as an integer. A finally-encoded expression has an indefinite number of interpretations; eval selects one of them.

The instance ExpSYM Int looks meta-circular, interpreting each expression in the object language as the corresponding expression in the metalanguage: object addition as Haskell addition. Unlike the initial interpreter eval :: Exp → Int, the final interpreter has no pattern-matching on (abstract) syntax and so has no syntax-dispatch overhead. In that respect ExpSYM resembles threaded code [11].

SYM in ExpSYM stands for *Symantics* [6]: the class declaration defines the syntax of the embedded language (its expression forms); class instances define interpretations, or the semantics. Multiple interpretations are now possible: we may interpret the very same term tf1 as a string, to pretty-print it:[1]

[1] Generally there are many ways to pretty-print a term – e.g., using prefix or infix notation for addition – all interpreting the term as a String. To distinguish among them, we could wrap String in various **newtype**s, see §3.4.

```
instance ExpSYM String where
  lit   n = show n
  neg e = "(-" ++ e ++ ")"
  add e1 e2 = "(" ++ e1 ++ " + " ++ e2 ++ ")"

view :: String -> String
view = id
```

with view tf1 giving the string "(8 + (-(1 + 2)))". The pretty-printing interpreter is again the identity; indeed it does not do anything. Only its type matters, which selects from the multitude of, one may imagine, already computed interpretations.

In the initial embedding, encoded object terms and their interpreters are ordinary, monomophic Haskell values, and hence are first-class. We may collect terms into a list, of the type [Exp]

```
til1  = [Lit 1, Add (Lit 1) (Lit 3)]
```

and interpret uniformly by mapping an evaluator, such as eval from the module I: **map** I.eval til1 gives the result [1,4] . The final encoding represents object terms as polymorphic Haskell values, which are not fully first-class: storing them in data structures or passing as arguments generally loses polymorphism. In some cases, such as the present one, it does not matter. We may still collect terms into a list

```
tfl1  = [lit 1, add (lit 1) (lit 3)]
```

and then **map** F.eval tfl1 obtaining the same [1,4] . We shall talk in detail about lost and regained polymorphism in §2.3.

We have defined the final embedding of an object language in Haskell and have seen its many similarities with the initial, data-type representation. Both embeddings permit multiple interpretations, and appear dual. The object language expressions in both encodings look the same modulo the case of the identifiers. The similarities prompt many further questions:

1. How to pattern-match on finally-encoded terms? How to process them in the ways that do not seem to resemble evaluation, that is, seemingly inexpressible as fold?
2. How to compare finally-encoded terms for equality?
3. How to use the final encoding to embed languages that are typed and higher-order?

We will be talking about these questions throughout the course. Before we get to them, we show the final approach answer to one of the common problems of generic programming, writing extensible code.

2.2 Extensibility and the expression problem

We have seen one sort of extensibility already, when we defined a view interpreter to complement the existing evaluator, and pretty-printed the existing object

language terms. We now want to extend the object language itself, by adding a new syntactic form, multiplication.

The initial encoding represents the object language as an algebraic data type; to add new forms to the language we have to add new variants to the data type:

```
data Exp = Lit Int
         | ...
         | Mul Exp Exp
```

We have to change the data type declaration, and hence adjust or at least recompile all the code that directly or indirectly refers to that declaration. We thus affirm the conventional wisdom [24, 40] that in basic functional programming it is easy to add new operations on data but hard to add new data variants. That is the problem – the *expression problem*.

The expression problem, like Sudoku, has gushed a fountain of various solutions and language features; see [24] for the history, the main approaches to the solution and references. We now demonstrate the final approach.

Suppose the final encoding of the original language, the type class ExpSYM and its two instances, §2.1, are in a (separately compiled) module that we import as F (its file name in the accompanying code is Intro2.hs). To add multiplication, we define a new type class, just for the new language form:

```
class MulSYM repr where
    mul :: repr → repr → repr
```

which can be used right away to write extended language terms:

```
tfm1 = add (lit 7) (neg (mul (lit 1) (lit 2)))
tfm2 = mul (lit 7) F.tf1
```

In tfm1, the new mul appears alongside the old forms lit, neg and add imported from F; the sample term tfm2 incorporates, with no changes, the imported term F.tf1 of the unextended language. The inferred type of the extended sample terms, (ExpSYM repr, MulSYM repr) ⇒ repr, patently shows the terms' using the mix of old ExpSYM and new MulSYM features.

We are yet to extend the two existing interpreters, eval and view. We do not touch the definitions of eval and view, however, or any other code in the module F. We merely define the meaning of mul in the semantic domains of Int and String:

```
instance MulSYM Int where
    mul e1 e2 = e1 * e2
instance MulSYM String where
    mul e1 e2 = "(" ++ e1 ++ " * " ++ e2 ++ ")"
```

That is all. We evaluate a sample term, eval tfm1, and pretty-print it, view tfm1, with the unmodified eval and view. Recall that these 'evaluators', both the identity functions, merely select an interpretation in the desired semantic domain (the result type). If we forget the MulSYM String instance, for example, attempting to view tfm1 will raise a type error.

Thus the final encoding makes it easy to add not only new interpretations but also new language forms, making the interpreters extensible by default. All the old code is reused, even in its compiled form. The extension mismatches are statically caught by the type checker.

A simple initial encoding like `Exp` can also be made extensible, with the folklore trick of defining a data type as a fixpoint of a constructor signature, a functor (see Swierstra [36] for explanation and history). The paper [36] describes combining constructor signatures by taking a co-product and automating the injections. Alas, the automation requires the controversial overlapping instances extension of GHC and the explicit enumeration of all constructor signatures in interpreter types. In contrast, the final approach works in Haskell 2010 and enjoys type inference. We will see in §3.4 that the final approach encodes terms that cannot be represented as fixpoints of a functor.

The dictionary-passing implementation of type classes gives an insight into the extensibility of the final approach. The implicit type class dictionaries are extensible. The OCaml code `final_dic.ml` explicates this point, by implementing the final encoding with explicit dictionary passing, using OCaml's extensible records as dictionaries. (OCaml also has extensible data types – so-called polymorphic variants – which permit writing simple classes of extensible interpreters as well [13].)

2.3 The de-serialization problem

The de-serialization problem [25] is a new problem, spun off the expression problem. Recall that the expression problem, cast in terms of embedded languages, is about defining new interpreters for the existing terms and extending interpreters to handle an enriched language. In both cases we are given embedded language terms as input. We have obtained the terms so far by entering Haskell code that, when compiled and run, will produce the values representing the desired terms. That method works well if we know the terms to process beforehand, or if we may use a Haskell interpreter such as GHCi, which lets us enter and evaluate code on-the-fly. The method of writing new Haskell code for each new embedded language term does not work well for communicating terms between computers or storing terms in files and processing them later.

One direction – storing and sending of the terms, or converting them into a sequence of bytes – is unproblematic, being a variant of pretty-printing, which we have already implemented. More difficult is the converse: reading a sequence of bytes representing an embedded language term and producing a value that can be interpreted with any existing interpreter. Reading, as a projection, is necessarily partial, since the input sequence of bytes, having potentially come from a network, could be corrupted. We wish to see the parsing error only once, upon de-serialization, rather than every time we interpret the term. Furthermore, extending our parser to accommodate the enriched language should reuse as much of the old parser code as possible, without breaking it. The de-serialization problem, of writing an extensible de-serializer [25, slide 18], is very hard. This section presents one of the first solutions.

We begin with the wire format for communicating and storing encoded embedded language terms. We chose a JSON-like format, represented in Haskell as

```
data Tree = Leaf String
          | Node String [Tree]
            deriving (Eq, Read, Show)
```

We rely on standard Haskell **read** and **show** to read and write Tree values from files.

The serializer, toTree, is just another interpreter of embedded language terms, quite similar to the view interpreter in §2.1:

```
instance ExpSYM Tree where
    lit  n      = Node "Lit" [Leaf $ show n]
    neg e       = Node "Neg" [e]
    add e1 e2 = Node "Add" [e1,e2]

toTree :: Tree → Tree
toTree = id
tf1_tree  = toTree tf1    -- sample tree
```

The result tf1_tree of serializing our running sample expression is

```
Node "Add" [Node "Lit" [Leaf "8"],
            Node "Neg" [Node "Add" [Node "Lit" [Leaf "1"],
                                    Node "Lit" [Leaf "2"]]]]
```

which does look like JSON data, or an S-expression.

Our task is to write the function fromTree, converting a Tree to a term that can be interpreted with any existing or future interpreter. Maintaining interpretation extensibility is challenging, as we shall see soon. To start, we should decide on fromTree's type. The type of a sample finally-encoded term tf1 :: ExpSYM repr ⇒ repr suggests ExpSYM repr ⇒ Tree → repr for the type of fromTree. Recall that the de-serializer may receive invalid input, for example, Node "x" [] . To model partiality and to report parsing errors we turn to the Error monad. We introduce a function to safely read an Int or other readable value, reporting the parsing error

```
type ErrMsg = String

safeRead :: Read a ⇒ String → Either ErrMsg a
safeRead s = case reads s of
               [(x,"")] → Right x
               _        → Left $ "Read␣error:␣" ++ s
```

and use it to de-serialize integer literals; we de-serialize composite expressions of our language inductively. The inferred type, shown as a comment, is as desired.

```
-- fromTree :: (ExpSYM repr) ⇒ Tree → Either ErrMsg repr
```

```
fromTree (Node "Lit" [Leaf n]) = liftM lit  $ safeRead n
fromTree (Node "Neg" [e])      = liftM neg $ fromTree e
fromTree (Node "Add" [e1,e2])  = liftM2 add (fromTree e1) (fromTree e2)
fromTree e = Left $ "Invalid tree: " ++ show e
```

As an example, we de-serialize tf1_tree serialized earlier

```
tf1'_eval =
    let tf1' = fromTree tf1_tree
    in case tf1' of
         Left  e -> putStrLn $ "Error: " ++ e
         Right x -> print $ eval x
```

and evaluate it. Since the de-serializer is partial, we have to check for error first, pattern-matching on **Either** ErrMsg repr value, before we get the term to interpret. The code works – but the problem is far from being solved.

We want to interpret a de-serialized term many times with many interpreters. If we try two, eval and view,

```
case fromTree tf1_tree of
  Left  e -> putStrLn $ "Error: " ++ e
  Right x -> do print $ eval x
                print $ view x
```

we get a type error, reporting that x cannot have both types Int and String. We have lost polymorphism. The problem is subtle: the function fromTree is indeed polymorphic over repr, as its inferred type shows. However, to extract the de-serialized term, we have to do pattern-matching; the variable x is bound in the case pattern and hence, like a lambda-pattern–bound variable, gets a monomorphic, non-generalizable type. Therefore, we cannot interpret x with several arbitrary interpreters; the extensibility is lost.

We may try changing fromTree to have the following signature

```
newtype Wrapped = Wrapped (∀ repr. ExpSYM repr ⇒ repr)
fromTree :: String -> Either ErrMsg Wrapped
```

resorting to fake first-class polymorphism.[2] The successful case analysis of the de-serialization result will give us a Wrapped value, which can be interpreted in many ways, as its type indicates. Alas we lost a different sort of extensibility. To wrap a term of an extended language with multiplication, we need the MulSYM repr constraint. There is no way to put that constraint into Wrapped except by changing the type declaration, which will break fromTree and all dependent code, requiring re-compilation.

The problem is indeed very hard. Yet there is a solution, involving a new, puzzling interpreter:

```
instance (ExpSYM repr, ExpSYM repr') ⇒ ExpSYM (repr,repr') where
```

[2] With the impredicative polymorphism GHC extension, we do not have to fake first-class polymorphism and do not need Wrapped.

```
lit  x          = (lit  x,  lit  x)
neg (e1,e2) = (neg e1,  neg e2)
add (e11,e12) (e21,  e22) = (add e11 e21, add e12 e22)

duplicate :: (ExpSYM repr, ExpSYM repr') ⇒ (repr,repr')  → (repr , repr ')
duplicate  = id
```

interpreting an embedded language term as two new terms. We observe in passing that the three occurrences of lit on the second line of the code belong to three different terms; the lit on the left-hand-side is of the term being interpreted; the two lits on the right-hand side are the constructors of two fresh terms. Suspending our bewilderment at the duplicating interpreter, we use the duplicator for multiple interpretations:

```
check_consume f (Left  e) = putStrLn $ "Error:␣" ++ e
check_consume f (Right x) = f x

dup_consume ev x = print (ev x1) ≫ return x2
 where (x1,x2) = duplicate x

thrice  x = dup_consume eval x ≫=dup_consume view ≫=print ∘ toTree

tf1 ' _int3  = check_consume thrice ∘ fromTree $ tf1_tree
```

and finally get working code, which prints the results of evaluating a successfully de-serialized term with three different interpreters. The trick becomes obvious in hindsight: a term has a polymorphic type if the term can be put – shared or copied – into differently-typed contexts. The duplicator copies, converting a monomorphic term into two monomorphic terms, with different, repr and repr', types. The two resulting terms hence can be passed to different interpreters. Thus, every time we wish to interpret a term, we have to duplicate it first, leaving a copy for the next interpreter (the function dup_consume illustrates that idiom).[3] It is an open question if the copying and its accompanying run-time cost can be avoided.

To be able to extend our de-serializer, we have to write it in the open-recursion style [27]. It is a bit unfortunate that we have to anticipate extensibility; alas, open recursion seems unavoidable for any extensible inductive de-serializer. After all, we will be extending not only the language but also the wire format.

```
fromTreeExt :: (ExpSYM repr) ⇒
   (Tree → Either ErrMsg repr) → (Tree → Either ErrMsg repr)
fromTreeExt self  (Node "Lit" [Leaf n]) = liftM lit  $ safeRead n
fromTreeExt self  (Node "Neg" [e])      = liftM neg $ self  e
```

[3] If one does not care at this point about enriching the language, one may avoid repeated duplications by defining a duplicate -like interpreter that yields Wrapped values. The unwrapping gives a term polymorphic over the interpretations of a non-extensible language – informally, performing a generalization. See the interpreter CL in `TypeCheck.hs` for an example.

```
fromTreeExt self (Node "Add" [e1,e2]) = liftM2 add (self e1) (self e2)
fromTreeExt self e = Left $ "Invalid tree: " ++ show e
```

We tie the knot with the fix-point combinator

```
fix f = f (fix f)
fromTree' = fix fromTreeExt
```

and run our examples:

```
tf1E_int3 = check_consume thrice ∘ fromTree' $ tf1_tree
tfxE_int3 = check_consume thrice ∘ fromTree' $
            Node "Lit" [Leaf "1", Leaf "2"]
```

The test tfxE_int3 tries to de-serialize an invalid input. Running the test prints an error that the input tree is bad. Since check_consume cannot help but check for errors before attempting interpretations, the printed error message confirms that the parsing of the input completes before any interpretation starts. We have indeed implemented a genuine de-serializer.

Our de-serializer is genuinely extensible as well. Not only can we interpret the successful de-serialization result in many ways; we can also enrich our language and re-use the existing, already compiled code. The file SerializeExt.hs in the accompanying code demonstrates assembling of the extended de-serializer from several, separately compiled pieces. We import the declaration of the base language (class ExpSYM) and its interpreters; we then import, from a different file, declarations of the new language form mul and of the extended interpreters. We finally import the basic de-serializer from the third module, called S. What remains is to add the instances for the serializer and the duplicator

```
instance MulSYM Tree where
    mul e1 e2 = Node "Mul" [e1,e2]

instance (MulSYM repr, MulSYM repr') ⇒ MulSYM (repr,repr') where
    mul (e11,e12) (e21, e22) = (mul e11 e21, mul e12 e22)
```

We define a new, for the module SerializeExt, function fromTreeExt with only two clauses. The first clause deals with the Mul node of the tree whereas the second clause has the old de-serializer S.fromTreeExt handle the other tree nodes.

```
fromTreeExt self (Node "Mul" [e1,e2]) = liftM2 mul (self e1) (self e2)
fromTreeExt self e = S.fromTreeExt self e -- use the old one
```

Finally, we tie the knot

```
fromTree = S.fix fromTreeExt
```

We test processing of the old serialized terms (tf1_tree from the module S) and the serialized extended terms

```
tf1'_int3  = S.check_consume S.thrice ∘ fromTree $ S.tf1_tree
tfm1'_int3 = S.check_consume S.thrice ∘ fromTree $ S.toTree tfm1
```

The last expression is striking: it uses the *old* interpreter code S.check_consume and S.thrice from a separately compiled module S to interpret the newly extended de-serialized expression tree tfm1.

We have solved the de-serialization problem, of writing an extensible de-serializer. We will re-visit this solution when we move to higher-order, typed languages, in §4.1.

2.4 Pattern-matching in the final approach

Evaluators, pretty-printers, serializers and other processors of embedded language terms have been interpreters, folding over a term. This section describes operations that do not look like folds. The initial approach lets us write such operations easily, with pattern-matching and general recursion. The final approach does not seem to permit these operations or pattern-matching. This section details why the impossibility is illusory, demonstrating at the end the tight correspondence between the initial and final approaches, letting us translate operations on terms back and forth. Although the general idea behind the final pattern-matching – making the context-dependence explicit – is clear, its realization at present is not as mechanical as one may wish. Explicating the idioms of the final approach is the subject of current research. We start by recalling the principle that underlies interpreters.

Compositionality The principle of compositionality [37]:

> (C) The meaning of a complex expression is determined by its structure and the meanings of its constituents.

is exemplified by the following clause of the evaluator for our language of arithmetic expressions, §2.1.

```
eval (Add e1 e2) = eval e1 + eval e2
```

To determine the meaning, the value, of the addition expression, we need only the meaning of its components e1 and e2; we do not need to know the syntactic form of the summands, their nesting depth, etc. Furthermore, we determine the meaning of e1 and e2 in isolation from each other and from the expression they are part of – that is, regardless of their context. Compositionality thus is context insensitivity. We defined the meaning of the addition expression without needing to know other expressions in the language. Compositionality thus is modularity, the all-important engineering principle, letting us assemble meanings from separately developed components.

Our embedded language interpreters have been compositional; they define the language's denotational semantics, which is required to be compositional. The compositional interpretation of a term is epitomized in a fold. Our interpreters are all folds. In the final approach, the fold is 'wired in' in the definition of the interpreters. Compositionality, or context-insensitivity, lets us build the meaning of a larger expression bottom-up, from leaves to the root. Again, in the final approach, that mode of computation is hard-wired in.

There are however many operations – for example, program transformations and optimizations – that do not seem compositional because the handling of a sub-expression does depend on where it appears in a larger expression (i.e., depends on the context). An apt example is transforming a logical formula to disjunctive normal form (DNF), by applying the distributivity laws and eliminating double-negation. To eliminate double-negation, we have to know if a negated expression appears as part of a bigger negated expression. We take this example to illustrate seemingly non-compositional processing, in initial and final approaches. Although our sample language is of arithmetic expressions rather than of logic formulas, the DNF transformation easily maps to our language, as the multiplying-out of factors.

Pushing negation down: the initial view Pushing the negation down to the literals, eliminating double-negation along the way is the first phase of DNF and of the multiplying-out transformations. The following BNF grammar defines the general form of expressions in our language:

```
e ::= int | neg e | add e e
```

We wish to transform the expressions to match a more restrictive grammar:

```
e      ::= factor | add e e
factor ::= int | neg int
```

which permits only integer literals be negated, and only once.

We write the negation pusher first in the initial approach, file PushNegI.hs, where object language expressions are represented as values of the algebraic data type Exp, §2.1. We rely on the law of negating a sum to push the negation towards the literals, eliminating double-negation along the way:

```
push_neg :: Exp → Exp
push_neg e@Lit{}              = e
push_neg e@(Neg (Lit _))      = e
push_neg (Neg (Neg e))        = push_neg e
push_neg (Neg (Add e1 e2)) = Add (push_neg (Neg e1)) (push_neg (Neg e2))
push_neg (Add e1 e2)          = Add (push_neg e1) (push_neg e2)
```

The type of push_neg emphasizes that we are transforming one expression to another; the result is an embedded expression in its own right and can be processed with any existing interpreter. The transformed expression should be equivalent to the source with respect to a set of laws. Recall our sample term ti1, whose printed view ti1 form is "(8 + (-(1 + 2)))". Pushing the negation down gives a new term, ti1_norm, which can be interpreted in many ways, for example, pretty-printed ti1_norm_view and evaluated ti1_norm_eval:

```
ti1_norm = push_neg ti1
ti1_norm_view = view ti1_norm
-- "(8 + ((-1) + (-2)))"
ti1_norm_eval = eval ti1_norm
-- 5
```

The result of pretty-printing, in the comments after ti1_norm_view, shows the negation having indeed been pushed down. The result of ti1_norm_eval is the same as that of eval ti1, confirming that the value of the term is preserved upon the transformation. As an additional example, negating ti1 and pushing the negation down, push_neg (Neg ti1), gives a term that pretty-prints as "((-8) + (1 + 2))".

The code for push_neg exhibits nested pattern-matching, betraying context-sensitivity. The processing of a negated expression depends on its context. The function push_neg is recursive but not structurally inductive: see, for example, the clause push_neg (Neg (Add e1 e2)). Therefore, the termination of the transformation is rather hard to see.

Pushing negation down: the final view We now write the negation pushing transformation in the seemingly impossible final approach, the file PushNegF.hs. The terms to transform are represented as polymorphic values of the type ExpSYM repr ⇒ repr. We cannot pattern-match on them, we can only write an interpreter of them. Writing an interpreter is what we shall do – paradoxically implementing a seeming non-compositional transformation as a compositional interpreter.

The operation of pushing negation down is indeed non-compositional, because the processing of a negated expression depends on its context. To be precise, it depends on whether the negated expression appears as part of a negated expression. We make that context-dependence explicit:

```
data Ctx = Pos | Neg

instance ExpSYM repr ⇒ ExpSYM (Ctx → repr) where
    lit  n Pos = lit  n
    lit  n Neg = neg (lit  n)
    neg e  Pos = e Neg
    neg e  Neg = e Pos
    add e1 e2 ctx  = add (e1 ctx) (e2 ctx)
```

This interpretation of a term yields another finally-encoded term ExpSYM repr ⇒ repr, depending on the context, Neg (within a negation) or Pos. The neg form interprets its sub-expression in the opposite context. The transformation interpreter supplies the initial, Pos, context:

```
push_neg e = e Pos
```

Several examples of pushing down the negation in the final style can be found in the file PushNegF.hs. The result is a tagless-final term, which can be interpreted in many ways; for example, pretty-printing view (push_neg tf1) gives "(8 + ((-1) + (-2)))".

One may informally argue that the negation-pushing transformation is more perspicuous in the final style; for example, it is clearly seen as a homomorphism with respect to addition. The transformation is now structurally inductive – it

is a fold; the termination is apparent. Elucidating the idioms of programming in the final style and of proving termination and other properties will hopefully give a firm basis to argue about clarity.

The final approach is extensible with respect to enriching the language. This advantage is preserved: pushing the negation is extensible as we add new forms to the language, for example, multiplication. The file PushNegFExt.hs shows assembling the extended transformer from the previously compiled components. We merely add the negation-pushing interpretation of multiplication. (Unlike addition, the negation of the product is equivalent to the negation of only one factor; we chose to negate the second factor.)

```
instance MulSYM repr ⇒ MulSYM (Ctx → repr) where
    mul e1 e2 Pos = mul (e1 Pos) (e2 Pos)
    mul e1 e2 Neg = mul (e1 Pos) (e2 Neg)
```

The previously defined PushNegF.push_neg can be used as it is to process extended language terms.

Flattening: the initial view It is instructive to try another example of seemingly non-compositional transformation. We continue the topic of the DNF-like normalization: after negations are pushed down, additions should be flattened and 'straightened out.' The embedded expressions should satisfy even more restricted grammar:

```
e ::= factor | add factor e
factor ::= int | neg int
```

The first summand must be a factor. The transformation amounts to repeatedly performing the conversion of (Add (Add e1 e2) e3) to (Add e1 (Add e2 e3)), that is, applying the associativity law to associate the factors to the right.

Again we start by writing the transformation in the initial approach, file FlatI.hs, on terms represented by the data type Exp:

```
flata :: Exp → Exp
flata e@Lit{} = e
flata e@Neg{} = e
flata (Add (Add e1 e2) e3) = flata (Add e1 (Add e2 e3))
flata (Add e1 e2)          = Add e1 (flata e2)
```

The code literally implements the algorithm of the repeated reassociation-to-the-right, assuming that only literals are negated. Unlike the pushing of negations, we repeatedly process the transformed expression in the last-but-one clause, which is patently not structurally inductive. The termination, and hence, correctness, is even harder to see. To show the termination, we have to introduce lexicographic ordering on the left- and the overall depths of a term. The nested pattern-match again betrays the context-sensitivity of the transformation.

To convert the terms of our language in the DNF-like form, we compose the two transformations.

```
norm :: Exp → Exp
norm = flata ∘ push_neg
```

Applying norm to a term ti3

```
ti3  = (Add ti1 (Neg (Neg ti1)))
ti3_view  = view ti3
-- "((8 + (−(1 + 2))) + (−(−(8 + (−(1 + 2))))))"

ti3_norm = norm ti3
ti3_norm_view = view ti3_norm
-- "(8 + ((−1) + ((−2) + (8 + ((−1) + (−2))))))"
```

produces ti3_norm that pretty-prints as shown in the comment line.

Flattening: the final view To write the flattening-of-additions transformation in the final approach, we again apply the principle of making the context explicit. Explicating the context-dependency turns the transformation into a compositional interpretation. In the initial flata code, the context-dependency manifested in the nested pattern-match (Add (Add e1 e2) e3): the processing of an addition expression depended on whether the expression is the left immediate child of an addition, or not. That is precisely the context information we need to make explicit.

In the file FlatF.hs we introduce

```
data Ctx e = LCA e | NonLCA
```

to discriminate the two contexts we care about. The variant LCA e3 represents the context Add [] e3, of being the left immediate child of the addition – or, of being added to e3 on the left. The following interpreter yields a flattened term, depending on the context:

```
instance ExpSYM repr ⇒ ExpSYM (Ctx repr → repr) where
    lit  n NonLCA = lit  n
    lit  n (LCA e)  = add (lit  n) e
    neg e NonLCA = neg (e NonLCA)
    neg e (LCA e3) = add (neg (e NonLCA)) e3
    add e1 e2 ctx  = e1 (LCA (e2 ctx))
```

As in the initial approach, we have assumed, in the second neg clause, that the push_neg transformation has been applied and so only literals are negated. The file FlatF.hs shows several examples of flattening and normalizing sample terms.

Recall that the data type Ctx with its two variants was meant to represent the context of an expression. In BNF that context can be defined as follows, where [] stands for the hole:

```
C ::= Add C e |[]
```

The last clause of the flattening interpreter, add e1 e2 ctx, specifies the meaning of C[Add e1 e2], that is, of the addition expression in the context ctx. That

meaning is e1 (LCA (e2 ctx)): the meaning of e1 in the context Add [] C[e2]. Overall, the last clause of the interpreter implements the transformation

$$\mathsf{C[Add\ e1\ e2]} \rightsquigarrow \mathsf{Add\ e1\ C[e2]}$$

which is precisely the rule of reassociating to the right. We argue again that the transformation is more perspicuous in the final approach, being structurally inductive. The termination and the correctness are much easier to see.

The reader is encouraged to add multiplication to the embedded language and implement the complete transformation of multiplying-out the factors.

Relating initial and final approaches in the first order The examples have demonstrated that it is possible after all to express seemingly non-compositional operations in the final approach: non-compositionality disappears when the context is explicated. The examples hinted at a connection between the two approaches, at the conversion of an initial-style transformation to the final style. The conversion so far has been creative. One may wonder about a systematic, mechanical process.

We describe two ways to systematically convert an operation on initially-encoded terms to the corresponding operation on the corresponding finally-encoded terms. We outline the first method, describing the second in detail.

The final approach represents a term in the embedded language as the value of the type ExpSYM repr ⇒ repr, where the type class ExpSYM is defined for example as in §2.1. The Haskell compiler GHC uses a so-called dictionary-translation [17] to represent polymorphic values with a type-class constraint. In the case of ExpSYM, the dictionary is defined as

```
data ExpSYMDict repr =
    ExpSYMDict{ lit_dict :: Int   → repr,
                neg_dict :: repr  → repr,
                add_dict :: repr  → repr  → repr }
type FinTerm repr = ExpSYMDict repr → repr
```

so that an embedded term is represented by GHC as the value of the type FinTerm repr. The latter is the Böhm-Berarducci encoding of algebraic data type Exp in System F [5], which may be regarded as the typed version of Church encoding. In the first-order unityped case, the *non-extensible* final encoding is thus equivalent to the Church/Böhm/Berarducci encoding of the data type representing the embedded language. (The type class dictionary in the explicit form ExpSYMDict is clearly not extensible.) The case of the typed object language, with more than one type, is discussed in §3.5.

The initial and the closed-to-language-extensions final approaches can also be related most straightforwardly (see the file PushNegFI.hs), by transforming a finally-encoded term to the corresponding data-type-encoded term, and vice versa. The relation thus is a bijection, witnessed by two total interpreters: interpreting a finally-encoded term as a data type

```
instance ExpSYM Exp where
```

```
    lit = Lit
    neg = Neg
    add = Add

initialize :: Exp → Exp
initialize = id
```

and conversely

```
finalize :: ExpSYM repr ⇒ Exp → repr
finalize (Lit n)     = lit n
finalize (Neg e)     = neg (finalize e)
finalize (Add e1 e2) = add (finalize e1) (finalize e2)
```

The interpreters look like glorified identity functions; the left- and right-hand-sides of each clause mention the same 'constructors', but in different cases. The finalize interpreter is explicitly a fold.

The push_neg transformation then in the final style is obtained from the corresponding transformation on the data type values by composing with the conversion interpreters:

```
push_neg = finalize ∘ I. push_neg ∘ initialize
```

Thus if we forget about extensibility, *any* processing on data type values, however non-compositional, can be performed on the corresponding finally-encoded terms. Using the intermediate data type to implement a transformation on finally-encoded terms is inefficient, and destroys the extensibility. It is an open question whether the intermediate data type values can be deforested or fused in.

3 Interpreting typed higher-order languages

Having warmed up, we turn to data types with binders and well-formedness constraints – in other words, to the embedding and interpretations of typed, higher-order object languages. This section introduces the typed tagless final approach in full, demonstrating not only the extensibility but also object types, expressing them in the metalanguage and manifestly ensuring their preservation during interpretations. Surprisingly we get by without dependent types, using only Haskell 2010. Our sample object language in this section will be simply typed lambda-calculus with constants, with binding represented either as de Bruijn indices, §3.3, or higher-order abstract syntax (HOAS), §3.4. As before we will be contrasting, §3.2, the final approach with the initial one. The initial approach will now require generalized algebraic data types (GADTs); we relate GADTs and the tagless final approach in §3.5. We call our approach in full 'typed tagless final'. We begin by explaining type tags, which may seem inevitable when interpreting a typed language in a typed metalanguage.

3.1 The problem of tags

We introduce type tags on the example of lambda-calculus with booleans, which was the introductory example of [31], extensively discussed in [6]. The object language, untyped for now, can be embedded in Haskell as an algebraic data type, similarly to the first-order case of §2.1.

```
data Exp = V Var
         | B Bool
         | L Exp
         | A Exp Exp
data Var = VZ |VS Var
```

The language has variables (represented as de Bruijn indices), boolean literals, abstractions L e and applications A e1 e2. A sample term applying the identity function to the boolean true is represented as

```
ti1  = A (L (V VZ)) (B True)
```

We naively try to write the textbook evaluator

```
-- Does not type check
eval0  env (V v) = lookp v env
eval0  env (B b) = b
eval0  env (L e) = \x → eval0  (x: env) e
eval0  env (A e1 e2) = (eval0 env e1) (eval0  env e2)
```

The first argument env to eval0 is the environment, a finite map from variables to values; the function lookp looks up the value associated with the given variable. The code is correct, and would have worked had our metalanguage been untyped. Expressed in a typed language, eval0 is ill-typed, which we can tell even without seeing the implementation of lookp. The second clause returns a boolean b whereas the next one returns a Haskell function, which cannot be of the type Bool. All branches of a pattern-match on ordinary algebraic data types must yield values of the same type. We have little choice but introduce the union type for booleans and functions, the universal type:

```
data U = UB Bool |UA (U → U)
```

The evaluator will return a value of the type U. The evaluator environment will likewise associate variables with the U values, letting us use the ordinary, homogeneous Haskell list to represent the environment, with lookp extracting an element of the list by its index.

```
lookp VZ (x:_) = x
lookp (VS v) (_: env) = lookp v env

-- eval :: [U] → Exp → U
eval  env (V v) = lookp v env
eval  env (B b) = UB b
eval  env (L e) = UA (\x → eval (x: env) e)
eval  env (A e1 e2) = case eval env e1 of UA f → f  (eval  env e2)
```

This code type-checks and works. Evaluating the sample term as eval [] ti1 gives the result UB **True**. The result is of the union type U, with UB being the discriminator, or the *tag*, of the value. The discriminators UB and UA of the union type U tell the type of the injected value; the discriminators are thus *type tags*. Had we written eval0 in a untyped metalanguage, the type tags would be present, too, but hidden in the run-time representation of values. The typed metalanguage gives more insight, forcing the hidden assumptions out.

Unlike the interpreters for the first-order languages in §2.1, eval is partial: first, there is a inexhaustive pattern-match when evaluating the A e1 e2 form. The inexhaustive pattern-match error its triggered when evaluating the term

```
ti2a = A (B True) (B False)
```

that tries to apply a boolean. There is also an inexhaustive pattern-match in the lookp function, triggering an error when looking up an unbound variable. In other words, we get stuck evaluating the open term ti2o

```
ti2o = A (L (V (VS VZ))) (B True)
```

in the empty initial environment: eval [] ti2o.

The object language being untyped, both sorts of errors may indeed occur during evaluation. To prevent such errors, we impose a type system, turning our language into simply typed lambda-calculus. One may imagine writing a function

```
typecheck :: Exp → Either ErrMsg Exp
type ErrMsg = String
```

implementing the standard type reconstruction and checking algorithm, taking a term and returning a type-checked term or a type error message. The type system will act then as an additional well-formedness constraint (per Curry); the function typecheck checks that constraint. If a term passes the check, the evaluation of the term should not encounter any errors: well-typed terms "don't go wrong." An attentive reader must have noticed the similarity with de-serialization in §2.3. As we did in the latter section, we type check a term once and evaluate it (potentially) many times:

```
\term → case typecheck term of
          Left e  → putStrLn $ "Type error: " ++ e
          Right x → do
                    print $ eval [] x
                    -- interpret again
```

Now eval [] x *should* always yield a value, without ever raising a run-time error. When evaluating a type-checked term, the pattern-matches in lookp and eval are effectively exhaustive. Yet they remain written as inexhaustive pattern-matches and are treated as such by the Haskell system, with all the attendant, now unnecessary, run-time type tag checking.

Pattern-matching on type tags in eval performs the dynamic checks that should not be necessary if the input term is well-typed. However, eval has no

way of knowing that its argument has been type checked. The function typecheck takes an Exp value and produces, if successful, another value of the same type Exp; the fact of the successful type checking is not reflected in types.

Thus the presence of the type tags like UB and UA and run-time tag checking are symptoms of the problem of embedding typed object languages. Informally, our embedding is not 'tight': the algebraic data type Exp contains more values than there are well-typed terms in the simply typed lambda-calculus with booleans. The embedding failed to represent the well-formedness constraints imposed by the object language's type system.

3.2 Tagless, initial and final embeddings

The problem thus is how to embed a typed object language without junk and take advantage of the well-typedness when writing interpreters, avoiding unnecessary checks and run-time errors. Hopefully we would also avoid the universal type U and hence the type tags and their run-time overhead.

The evaluator in the previous section could get stuck because of two inexhaustive pattern-matches; the one in the lookp function may raise a run-time error when looking up a variable absent in the environment. To eliminate such, essentially array bound errors, we may need dependent types [48]. A metalanguage with dependent types such as Agda [29] indeed embeds simply typed lambda calculus enforcing well-typedness constraints on embedded terms and implements evaluator that does not get stuck.

In this section, we show that Haskell is sufficient to solve the problem of embedding of typed languages and their tag-free interpretations. There are again the initial and final approaches. Whereas the initial approach requires generalized algebraic data types, the final approach is implementable in Haskell2010.

We start with the initial approach to embedding simply typed lambda calculus with booleans, the language of the previous section. As before, §2.1, the initial approach represents terms of the embedded language as values of an algebraic data type. The previous section showed that an ordinary algebraic data type is unsuitable: it is too 'large'. We need a tight embedding that represents all and *only* typed object terms. We hence move from Curry's view of types to Church's view: ill-typed terms 'do not exist' and should not be representable. To express the well-typedness constraint we have to keep track of types when constructing representations. To avoid getting stuck when encountering an unbound variable, we have to parameterize the representation data type not only with the type of the object term but also with the free variables in the term. Thus we need a generalized algebraic data type (GADT) with two type parameters:

```
data Exp env t where
  B :: Bool              → Exp env Bool
  V :: Var env t         → Exp env t
  L :: Exp (a,env) b     → Exp env (a→ b)
  A :: Exp env (a→ b) → Exp env a → Exp env b
```

```
data Var env t where
    VZ :: Var (t, env) t
    VS :: Var env t → Var (a, env) t
```

The data types Var and Exp are quite like those in §3.1, modulo two parameters, env and t, and the well-formedness constraint expressed in the types of the data constructors. The first type parameter, env, is the type environment, modeled as a nested tuple, assigning types to free variables in a term. The type parameter t is the object type of the embedded expression, which could be boolean (represented by Haskell's Bool) or a function type, represented by Haskell's arrow type. Object terms that do not have types cannot be embedded.

The constructor declarations express the type system of the calculus: for example, the type signature of B says that boolean literals have the type Bool in any environment. The signature of A states that applying a function of type a→ b to a term of type a gives a term of type b, all in the same environment env. We may also read the Exp and Var declarations as the statements of the axioms and inference rules of the implication fragment of minimal intuitionistic logic: B is the axiom of booleans; V is the reference to a hypothesis denoted by a variable; L is implication introduction and A is implication elimination. VZ is the assumption axiom and VS is weakening.

The sample term, an application of the identity function to the boolean true, looks exactly as in §3.1

```
ti1  = A (L (V VZ)) (B True)
```

To evaluate it, we write the standard evaluator in the most straightforward way:

```
eval :: env → Exp env t → t
eval env (V v) = lookp v env
eval env (B b) = b
eval env (L e) = \x → eval (x, env) e
eval env (A e1 e2) = (eval env e1) (eval env e2)
```

This is exactly the evaluator eval0 that we wanted to write in §3.1! It did not type check then because all branches of a pattern-match on ordinary algebraic data type must return the values of the same type. GADTs lift that restriction. The type of eval states that the type parameter t of the Exp GADT is the type of the evaluation result. The type of B says that B constructs Exp env Bool from a boolean b. When checking the second clause of eval, the type checker assumes t to be Bool and expects the clause to produce a boolean, which it does. Likewise, the L data constructor builds Exp env (a→ b) values. The type checker expects then the third clause of eval to yield a value of the arrow type; a Haskell lambda-term does indeed have the arrow type. The need for the union type U has disappeared, and with it, the type tags.

We are yet to implement lookp to look up a variable in the environment env, which we have decided to model as a nested tuple.

```
lookp :: Var env t → env → t
lookp VZ (x,_) = x
```

```
lookp (VS v) (_, env) = lookp v env
```

The code is similar to lookp of §3.1, with the heterogeneous list (nested tuple) in place of a homogeneous, ordinary Haskell list. More interesting is the type of lookp, claiming that if we have a variable that has type t in the environment env, the environment certainly has the corresponding value of the type t and we can retrieve it. The code proves the claim. The function lookp is now total: it cannot receive a VS VZ value and the empty environment because VS VZ has the type Var (a, env) t for some a, env and t. (Although the pattern-match in lookp is exhaustive as we have just shown, GHC currently cannot do such reasoning and flags the pattern-match as inexhaustive. Therefore, GHC has to compile in a test and the code to raise the pattern-match exception.)

We evaluate the sample term, eval () ti1, and obtain **True**, a genuine Haskell boolean with no type tags. The problematic term ti2a from §3.1 applying a boolean cannot be built: GHC rejects the constructor expression A (B **True**) (B **False**) as ill-typed. Indeed, only well-typed terms can be represented. Open terms like ti2o from §3.1 are representable. The inferred type Exp (b, env) b tells that the term is open; therefore, an attempt to evaluate it in the empty environment, eval () ti2o, is rejected by the type checker.

We have thus solved the problem of representing a typed object language, simply typed lambda-calculus, in a typed metalanguage in the tight encoding. We wrote a tagless interpreter, which does no dynamic tag checking and does not get stuck. The well-formedness constraints imposed by the type system of the object language are expressed through the types of the metalanguage. As a bonus, the Haskell compiler checks the object types and even infers them, relieving us from writing our own type checker.

The initial approach solution has relied on GADTs. One may hear claims that dependent types or at least their lightweight version, GADTs, are essential for tagless embeddings of typed languages. Let us look however at the final tagless encoding.

Recall that the final approach represents a term of an embedded language by its value (in the appropriate semantic domain), or by a Haskell expression that computes that value. That expression is build compositionally from functions that compute the meaning of primitive expressions. Our sample language has five primitive expressions: boolean literals, abstraction, application, zero-index variable reference and the de Bruijn index increment. Assuming the functions b, l, a, vz and vs to compute the meaning of these primitive expressions, we write a complex expression as follows:

```
tf1 = a (l vz) (b True)
```

This Haskell expression represents an embedded language term applying the identity function to the boolean true. The expression looks exactly like the initial encoding ti1 of the same term, with all data 'constructors' in lower case.

We have not yet defined the functions a, b, etc. If we are interested in the value of an expression, we choose the semantic domain to be a map from the environment to the Haskell value representing the result of the expression – the

textbook semantic domain for the standard denotational semantics of simply typed lambda calculus. We define the value of each primitive expression in that domain:

```
vz (vc, _)       = vc
vs vp (_, envr)  = vp envr

b bv env     = bv
l e env      = \x -> e (x, env)
a e1 e2 env = (e1 env) (e2 env)
```

Booleans are interpreted as Haskell booleans, variable references are interpreted by whatever the environment associates with them. Supplying the initial empty environment, e.g., `tf1 ()`, gives us the value of the represented term, **True**.

These five lines are the entire final interpreter. It clearly has no type tags. It is typed; the inferred types are

```
b :: t -> env -> t
l :: ((t1, env) -> t) -> env -> t1 -> t
a :: (env -> t1 -> t) -> (env -> t1) -> env -> t
```

The interpreter is expressed in the Hindley-Milner subset of Haskell 2010. No fancy types are thus needed for a tagless embedding of a typed object language. The final interpreter we have shown is the evaluator; the evaluator, one may say, was wired into the representation. To permit multiple interpretations of an embedded language term we have to abstract over the interpretation, as we did in §2.1. That is, we abstract from the term such as `tf1` the primitive expression denotations, the functions `a`, `b`, etc. ML modules or Haskell type classes provide exactly the right abstraction mechanism.

3.3 Tagless final embedding with de Bruijn indices

This section describes the abstraction over the primitive interpreters, introducing the typed tagless final approach in full, along the lines of [6]. Our example is still simply typed lambda calculus with constants; we replace boolean literals with integer ones and include addition, to write more interesting examples. We also rename our primitive form interpreters to match [6].

We introduce, in the file `TTFdB.hs`, the type class with methods to interpret the primitive forms of the embedded language.

```
class Symantics repr where
  int :: Int -> repr h Int
  add :: repr h Int -> repr h Int -> repr h Int

  z   :: repr (a,h) a
  s   :: repr h a -> repr (any,h) a
  lam :: repr (a,h) b -> repr h (a-> b)
  app :: repr h (a-> b) -> repr h a -> repr h b
```

We may read this declaration as the BNF grammar for the language: the integer literals int and the zero-index variable z are the terminals; if e1 and e2 are expressions of the language (that is, have the type repr · ·), so is app e1 e2. The Symantics declaration thus defines the syntax of the object language (the type class instances will define semantics, hence the type class name).

The type class Symantics is quite like GADT Exp from §3.2. Symantics is also similar to the type class ExpSYM seen in the first-order unityped case, §2.1. The type class parameter repr is now a type constructor (with higher kind, $* \to * \to *$). The declaration of the class Symantics defines not just the syntax of the object language, but also its type system. A (non-bottom) Haskell value of the type repr h t represents an embedded language expression – or, witnesses its grammar derivation. The same value also witnesses the type judgment that the represented expression has the type t in the type environment h. The types of Symantics methods read as the specification of the axioms and inference rules of the type system: the type of z says that a zero-index variable z in the type environment modeled as the nested pair (a,h) has the type a. The type of lam specifies that if e has the type b in the environment (a,h) then lam e has the type a→ b in the environment h. We may also read these types as stating the axioms and inference rules of the minimal logic: z is the assumption axiom (assuming A we may derive A) and lam is the implication introduction (if we may derive B assuming A, we may derive A→ B). We have thus demonstrated specifying the type system of the simply typed lambda calculus in Haskell 2010, with no need for dependent types. (We point to the specification of more complex type systems in §§4.2, 4.3)

Here are a few sample embedded terms; the first one represents the addition of 1 and 2:

```
td1 = add (int 1) (int 2)
-- td1 :: (Symantics repr) => repr h Int

td2o = lam (add z (s z))
-- td2o :: (Symantics repr) => repr (Int, h) (Int -> Int)

td3 = lam (add (app z (int 1)) (int 2))
-- td3 :: (Symantics repr) => repr h ((Int -> Int) -> Int)
```

GHC infers the types for us, shown in the comments underneath the term. GHC also infers the most general environment in which the term is typed: td1 is typed in any environment, whereas td2o is typed only in the environment whose first assumption is Int: td2o is an open term. Only well-typed terms are representable. Furthermore, GHC gives a good error message when rejecting an ill-typed term, such as the self-application:

```
* TTFdB> lam (app z z)
    Occurs check: cannot construct the infinite type: a = a -> b
      Expected type: repr (a -> b, h) a
      Inferred type: repr (a, h) a
```

```
In the second argument of 'app', namely 'z'
In the first  argument of 'lam', namely '(app z z)'
```

As in the unityped case §2.1, interpreters of the embedded language are the instances of Symantics. Our first interpreter is the evaluator. Since terms may be open, we interpret them as functions from the environment (a nested tuple carrying the values associated with free variables) to Haskell values, implementing the standard denotational (or, natural) semantics of simply typed lambda-calculus.

```
newtype R h a = R{unR :: h → a}

instance Symantics R where
  int x     = R $ const x
  add e1 e2 = R $ \h → (unR e1 h) + (unR e2 h)

  z     = R $ \(x,_) → x
  s v   = R $ \(_,h) → unR v h
  lam e = R $ \h → \x → unR e (x,h)
  app e1 e2 = R $ \h → (unR e1 h) (unR e2 h)
```

The type constructor R is the interpreter's name, or the selector from many possible interpretations of an embedded term. In §2.1 we used the type of the desired result, the semantic type, as the selector. There may be several interpreters with the same semantic type, however; we turn to user-defined names like R to disambiguate. We stress that R is not a type tag: pattern-matching on R is always exhaustive, the function unR is total. Furthermore, since R is declared as a **newtype**, it has no run-time representation; the function unR is operationally the identity. R interprets the object-language addition as the Haskell addition; the object-level application as the Haskell one. Since these Haskell operations do not raise run-time errors and R has no inexhaustive pattern-matching, R never gets stuck (and is in fact total). Well-typed (object) terms indeed "don't go wrong." One may view R as a constructive proof of the type soundness for the object language: R does not get stuck, and interpreting an object expression of the type t in the environment h indeed gives a value of the type h→ t. We are sure of the latter claim because the instance Symantics R has been accepted by the Haskell type checker, which verified that the right-hand side for, say, add e1 e2 indeed has the claimed type R h Int , isomorphic to h→ Int. Our subset of Haskell 2010 is sound. We thus reduced the type soundness of the object language to the type soundness of the metalanguage.

To evaluate a closed object term, we R-interpret it in the empty environment.

```
eval e = unR e ()
```

For example, our sample term td1 evaluates to the Haskell integer 3; eval td3 gives the Haskell value of the type (Int → Int) → Int , which is a regular Haskell function, which we can apply to an Int → Int argument (e.g., (eval td3) (+ 2) evaluates to 5). The term td2o is open, therefore, eval td2o is ill-typed.

The typed tagless final approach lets the programmer define new interpreters for existing terms. As an example, we show a pretty-printing interpreter:

```
newtype S h a = S{unS :: Int → String}

instance Symantics S where
    int  x     = S $ const $ show x
    add e1 e2 = S $ \h →
      "(" ++ unS e1 h ++ "+" ++ unS e2 h ++ ")"

    z     = S $ \h → "x" ++ show (h−1)
    s v   = S $ \h → unS v (h−1)
    lam e = S $ \h →
       let x = "x" ++ show h
       in "(\\" ++ x ++ "␣→␣" ++ unS e (h+ 1) ++ ")"
    app e1 e2 = S $ \h →
      "(" ++ unS e1 h ++ "␣" ++ unS e2 h ++ ")"

view :: S () a → String
view e = unS e 0
```

For the sample term td3, view-ing it gives the string "(\\x0␣→␣((x0␣1)+ 2))". The semantic domain now is functions from the nesting depth of lambda-abstractions to text strings. The most notable difference between the R and the S interpreters is the interpretation for lam. In either case however the interpreters express a typed fold over a typed term – which is the essence of the typed tagless final approach.

We are yet to demonstrate a different sort of extensibility, enriching the language with more expression forms. As we make the language more interesting and examples more complex, we quickly realize that variable names are much better for humans than indices. Therefore, we first solve the problem of naming the variables, introducing an alternative typed tagless final embedding.

3.4 Tagless final embedding with higher-order abstract syntax

We now let the programmers write embedded terms using names for the variables rather than obscure indices. This alternative typed tagless final embedding shares most of the properties with the de Bruijn-index–based approach of §3.3. Only typed terms are representable; GHC checks and infers the types and prints descriptive messages for ill-formed or ill-typed terms. New is the guarantee that all terms are closed, since open terms become inexpressible (that is, open object terms cannot be represented as repr values). The improved convenience of writing embedded terms gives us a better opportunity to demonstrate enriching the language. The result of the enrichments is PCF [33], the simply typed lambda-calculus with integer and boolean operations, the conditional, and the fixpoint. The language is an extension of the one used in [47] to introduce GADTs. We will be using Haskell 2010, with no GADTs.

We start with the small embedded language of the previous section, simply typed lambda calculus with integer literals and addition. We will now model

bindings in the object language with Haskell bindings, similar to the way Church [7] used metalanguage bindings (lambda-abstractions) to model quantification. This, so-called higher-order abstract syntax [28, 32], represents object language abstractions as Haskell abstractions and object variables as Haskell, named variables. The object-level binder lam becomes a higher-order Haskell constant, similar to the quantifiers ∀ and ∃ in Church's Simple Theory of Types [7]. The type class Symantics from the previous section becomes as follows

```
class Symantics repr where
  int :: Int -> repr Int
  add :: repr Int -> repr Int -> repr Int

  lam :: (repr a -> repr b) -> repr (a-> b)
  app :: repr (a-> b) -> repr a -> repr b
```

Embedded expressions of the type t are represented as Haskell values of the type Symantics repr ⇒ repr t. We no longer keep track of bindings and the environment since Haskell does that for us. The types of Symantics methods do read like the axioms and inference rules of the minimal logic in Gentzen-style natural deduction. The duality of implication introduction, lam, and implication elimination, app, has become clear.

Here are the sample embedded terms and their inferred types

```
th1 = add (int 1) (int 2)
-- th1 :: (Symantics repr) => repr Int

th2 = lam (\x -> add x x)
-- th2 :: (Symantics repr) => repr (Int -> Int)

th3 = lam (\x -> add (app x (int 1)) (int 2))
-- th3 :: (Symantics repr) => repr ((Int -> Int) -> Int)
```

The terms use variable names like x rather than indices: compare th3 with td3 from §3.3. Open terms like td2o from the previous section cannot be expressed at all at the top-level: since object variables are now Haskell variables, open object terms are open Haskell terms, not allowed at the top level.

Haskell's taking over of the tracking of binders and the maintenance of the binding environment simplifies the interpreters. The evaluator R from §3.3 now reads

```
newtype R a = R{unR :: a}

instance Symantics R where
  int x    = R x
  add e1 e2 = R $ unR e1 + unR e2

  lam f    = R $ unR ∘ f ∘ R
  app e1 e2 = R $ (unR e1) (unR e2)
```

Since R (which is the name for the interpreter, not a type tag) is a **newtype**, at run-time, R x is indistinguishable from x. It becomes obvious that the interpreter R is meta-circular: object-language integers are the Haskell integers themselves; object-language addition *is* Haskell addition and object-language application *is* Haskell application. It is even more obvious that R never gets stuck. The eval function, as that in §2.1, is operationally the identity

```
-- eval :: R a → a
eval  e = unR e
```

Only its type matters, selecting the interpretation named R from other interpretations of a term. Evaluating th1 gives 3; eval th3 is a Haskell function, which we cannot show but can apply: eval th3 (+ 2) is 5.

The pretty-printing interpreter S, which does let us see object terms including abstractions, has scarcely changed compared to the previous section; in both cases we have to convert the variable names or indices into character strings and so maintain a counter to generate fresh names.

```
type VarCounter = Int
newtype S a = S{unS:: VarCounter → String}

instance Symantics S where
    int  x      = S $ const $ show x
    add e1 e2 = S $ \h →
      "(" ++ unS e1 h ++ "+ " ++ unS e2 h ++ ")"

    lam e = S $ \h →
       let  x = "x" ++ show h
       in  "(\\" ++ x ++ "␣→ ␣" ++
             unS (e (S $ const x)) (succ h) ++ ")"
    app e1 e2 = S $ \h →
      "(" ++ unS e1 h ++ "␣" ++ unS e2 h ++ ")"

view e = unS e 0
```

Although embedded language abstractions are represented as Haskell abstractions, we can show them: view th3 gives "(\\x0␣→ ␣((x0␣1)+ 2))".

An object language term of the type Int is represented as a Haskell value of the type Symantics repr ⇒ repr Int, which can be specialized either as R Int or S Int. The former is essentially an Int; the latter is VarCounter → String regardless of the object type. Different interpretations of a term may indeed vary quite a lot. This variety is hidden behind the opaque repr in Symantics repr ⇒ repr t. The typed tagless final encoding may be called translucent: it hides concrete representations yet exposes enough of the type information to type check the encoding of an object term without knowing its concrete representation. The checked term is then well-typed in any interpreter, for any instantiation of repr. The higher-order polymorphism, quantifying over type variables like repr of higher kind, is essential.

We finally demonstrate the promised extensibility, enriching the language with multiplication, boolean literals, integer comparison, the conditional, and the fixpoint. The language extensions can be introduced independently and separately. We group them in three classes

```
class MulSYM repr where
  mul :: repr Int → repr Int → repr Int

class BoolSYM repr where
  bool :: Bool → repr Bool
  leq  :: repr Int → repr Int → repr Bool
  if_  :: repr Bool → repr a → repr a → repr a

class FixSYM repr where
  fix  :: (repr a → repr a) → repr a
```

The extension method is quite like the one in §2.2; new is the parametrizing of repr by the object type. We write more interesting sample terms, such as the power function tpow, its partial application tpow7 and the saturated application tpow72, representing the object term that computes 2^7.

```
tpow = lam (\x → fix (\self → lam (\n →
                        if_ (leq n (int 0)) (int 1)
                            (mul x (app self (add n (int (−1))))))))
-- tpow :: (Symantics repr, BoolSYM repr, MulSYM repr, FixSYM repr)
--         ⇒ repr (Int → Int → Int)

tpow7 = lam (\x → (tpow `app` x) `app` int 7)
tpow72 = app tpow7 (int 2)
-- tpow72 :: (Symantics repr, BoolSYM repr, MulSYM repr, FixSYM repr)
--           ⇒ repr Int
```

The inferred types, shown in the comments underneath the corresponding terms, enumerate the language features used by the term. The expression tpow looks like Scheme code; we could have defined infix operators however for arithmetic expressions, or used the infix notation as in tpow7. The convenience of variable names becomes apparent: writing tpow with de Bruijn indices is frightening.

Having extended the language, we extend its interpreters, re-using rather than breaking the interpreters for the base language. The extensions are independent and separate:

```
instance MulSYM R where
  mul e1 e2 = R $ unR e1 ∗ unR e2

instance BoolSYM R where
  bool b    = R b
  leq e1 e2 = R $ unR e1 ≤unR e2
  if_ be et ee = R $ if unR be then unR et else unR ee
```

```
instance FixSYM R where
  fix  f = R $ fx (unR ∘ f ∘ R) where fx f = f (fx f)
```

Evaluating tpow72 as eval tpow72 gives 128. The interpreter R inherits the evaluation strategy from the metalanguage; R is thus non-strict. We did not notice it before because our object language was strongly normalizing. The introduction of the fixpoint combinator lets us distinguish call-by-value and call-by-name. For example, eval (lam (\x → int 5) `app` (fix **id**)) returns 5; the call-by-value evaluation would diverge. We can write call-by-value evaluators too, and even call-by-need (see §4.4).

We likewise extend the S interpreter. We show the most interesting case, for fix:

```
instance FixSYM S where
  fix  e = S $ \h →
    let self = "self" ++ show h
    in "(fix " ++ self ++ "." ++
         unS (e (S $ const self)) (succ h) ++ ")"
```

Pretty-printing a term with fix, unlike evaluating it, requires no recursion. The extended S interpreter clearly remains total. Pretty-printing tpow gives

```
"(\\x0 → (fix self1 .
   (\\x2 → (if (x2 ≤ 0) then 1 else (x0 * (self1 (x2 + -1))))))"
```

One may write more interpreters, to compute the size of a term by counting the number of constructors, to partially evaluate a term or transform it into continuation-passing style. We may also add more language features, for example, state or reference cells; see [6] for details. No type tags, no GADTs, no dependent types, no intensional type analysis are involved. The type system of the metalanguage assures that all interpretations preserve object types. Since the code of the interpreters use no partial operations, the interpreters *manifestly* do not get stuck. The interpreters thus provide a constructive proof of type soundness of the object language.

We have thus demonstrated a family of interpreters for a *typed* higher-order object language in a *typed* metalanguage, solving the problem of tagless type-preserving interpretation, without fancy.

3.5 Relating initial and final typed tagless encodings

We have described two tagless embeddings of a typed higher-order language into the typed metalanguage, Haskell. Both embeddings are tight, in that only well-typed object terms are representable. The tightness of embedding lets us write interpreters without resorting to type tags. The final encoding uses no fancy types (in particular, no GADTs) whereas the initial encoding does. The similarities of the two approaches raise the question of their deeper relationship, which may give insight into the fancy types. We have compared the initial and the

final approaches for first-order, unityped object languages in §2.4 and found them related by bijection; furthermore, the non-extensible final encoding is equivalent to the Church/Böhm/Berarducci encoding of the data type representing the embedded language. This section shows that the bijection between the initial and final embeddings holds also for higher-order typed languages, for which Church/Böhm/Berarducci encoding does not apply.

We have described the initial typed tagless encoding in §3.2, with the de Bruijn-index representation of variables. We briefly revisit the initial encoding, using this time a richer language and higher-order abstract syntax, for the ease of comparison with the tagless final encoding of §3.4. Our revisited initial encoding is a generalization of the one presented in [47] as the motivation for GADTs (for brevity we elide the fixpoint combinator below; see TTIF.hs for the complete example, of the full PCF.)

The embedded language is simply typed lambda calculus with integer literals and addition. The tagless initial encoding represents expressions of the language as values of the following GADT:

```
data IR h t where
  INT :: Int   → IR h Int
  Add :: IR h Int → IR h Int → IR h Int

  Var :: h t → IR h t
  Lam :: (IR h t1 → IR h t2) → IR h (t1→ t2)
  App :: IR h (t1→ t2) → IR h t1 → IR h t2
```

The Var form, like HOASLift of [47], 'lifts' any value from the metalanguage into the object language. Unlike HOASLift, Var is parametrized by the representation of the lifted values, h. One may view h as modeling the binding environment: h t is the type of an environment cell holding a value of the type t. The type of the constructor Lam contains the contra-variant occurrence of IR h. Therefore, IR h t is not an inductive data type and is not representable as a fix-point of a functor. Such generally recursive data types are not in the domain of Böhm/Berarducci encoding.

The sample embedded terms th1, th2, th3 of §3.4 have the following form and types in the initial encoding. The only difference between the initial and final encodings is the capitalization of the constructors.

```
ti1 = Add (INT 1) (INT 2)
-- ti1 :: IR h Int

ti2 = Lam (\x → Add x x)
-- ti2 :: IR h (Int → Int)

ti3 = Lam (\x → Add (App x (INT 1)) (INT 2))
-- ti3 :: IR h ((Int → Int) → Int)
```

The evaluator of the embedded language looks almost the same as the one in §3.2 and is standard

```
evalI :: IR R t → t
evalI (INT n)     = n
evalI (Add e1 e2) = evalI e1 + evalI e2
evalI (Var v) = unR v
evalI (Lam b) = \x → evalI (b ∘ Var ∘ R $ x)
evalI (App e1 e2) = (evalI e1) (evalI e2)
```

The evaluator does no environment look-up to obtain the value associated with a bound object variable; the value is available directly (see the Var x clause, keeping in mind that operationally R x is the same as x since R is a **newtype**, §3.4). The reliance on Haskell's environment for variable bindings relieves us from maintaining our own (compare with eval in §3.2). Like the tagless final evaluator R, this initial evaluator is also tagless and free from pattern-match errors. As in the final approaches, we may add more interpreters, see see TTIF.hs for the initial pretty-printer. Unlike the final approach however, enriching the language breaks existing interpreters.

The initial and final tagless typed approaches are related by bijection, as they were in the first-order unityped case, §2.4. The bijection is witnessed by the total interpreter of finally-encoded terms producing the initial IR representation of the same terms

```
instance Symantics (IR h) where
    int  = INT
    add  = Add

    lam  = Lam
    app  = App

f2i :: IR h t → IR h t
f2i = id
```

and by the inverse, total interpreter, returning finally-encoded terms

```
i2f :: Symantics repr ⇒ IR repr t → repr t
i2f (INT x)     = int x
i2f (Add e1 e2) = add (i2f e1) (i2f e2)
i2f (Var v) = v
i2f (Lam e) = lam(\x → i2f (e (Var x)))
i2f (App e1 e2) = app (i2f e1) (i2f e2)
```

The interpreters follow the pattern seen in §2.4: f2i is a glorified identity and i2f is a fold. The file TTIF.hs shows on many examples that the composition of f2i and i2f is the identity.

The tagless final encoding for typed object languages may be regarded as the generalization of Church/Böhm/Berarducci encoding to generalized, generally recursive algebraic data types. If it were not for the Lam constructor (and the corresponding constructor function lam), the tagless final encoding is closely connected to a generalized Church encoding [22] for the GADT IR h t (The

category-theoretical treatment in the paper [22] assumes a number of implicit type isomorphisms, which have to be worked out to connect tagless final and generalized Church encodings.) However, with the Lam constructor, the GADT IR h t is not covariant and is out of scope of Johann and Ghani [22]. Weirich [43, 45] has described early encodings, of some GADTs, in System F and System F_ω.

4 Real fun

The attraction of the typed tagless final approach is in its applications beyond the embedding of the simply typed lambda calculus. This section gives a taste of these fun applications: extensibility as in adding more evaluation strategies including call-by-need, §4.4; embedding of languages with more interesting type systems such as effect typing, §4.2, and of the linear lambda calculus, §4.3; embedded language transformations like the optimal continuation-passing–style (CPS) transform, §4.5, and the type-directed partial evaluation, §4.6. Save for typed formatting §4.2, we will give only a brief overview pointing to the well-commented online code for further details.

4.1 Typed compilation

We start by revisiting the de-serialization problem described in §2.3: the problem becomes much more frustrating, exhilarating, time consuming and addictive in the general case of higher-order typed embedded languages. The problem is to read an embedded language expression from a file, parse it and 'compile' it; the result should be the same as if we entered the expression as its representing Haskell code, compiled and ran the code. In either case, the result fits for any existing and future interpreter of the embedded language. The aim is to parse an expression only once and interpret it many times; parsing errors should be reported once, before any interpretation is attempted. If the embedded language is typed, we not only have to parse embedded terms but also type check them. We no longer can rely on the Haskell compiler for type checking, type inference, and type error reporting. Our goal is still to type check an expression once, during de-serialization, and interpret the result many times.

Since our type checker has to represent types and reason about type equality, we first develop type representations, comparison and the type safe cast, see the file Typ.hs. We regard the language of types, too, as a typed, first-order object language, which we embed in Haskell in the typed tagless final style and for which we solve the de-serialization problem. The file Typ.hs is the tagless final version of the standard Data.Typeable, implemented however above-the-board, with no internal GHC operations, no questionable extensions, or even a hint of unsafe operations.

The type checker itself is in the file TypeCheck.hs. The code is quite similar to Baars and Swierstra's "Typing Dynamic Typing" [4] . The main difference is open interpretation: the result of our type checking is interpretable with any

existing or future interpreter of the embedded language. Furthermore, our code is written to expose more properties of the type checker for verification by the Haskell type checker; for example, if we successfully de-serialized a term in the empty initial environment, the result is the assuredly closed final term. In the initial approach, Weirich wrote a similar type checker [44] that produces initial encodings of embedded terms and extensively relies on GADTs.

4.2 Typed formatting

We turn to the embedding of languages with interesting type systems. This section describes the language of formatting patterns and its two interpreters, sprintf for formatted printing and sscanf for parsing. To ensure that the formatted IO is type-safe, that is, the types and the number of arguments to sprintf and sscanf functions match the formatting pattern, the language of patterns has in essence a type-and-effect system.

The typed formatting problem is to write type-safe versions of the familiar C functions printf and scanf. (Formatted IO existed already in FORTRAN.) The polyvariadic function sprintf should take the formatting specification (the formatting pattern) and the values to format, and return the formatted string. The types and the number of sprintf's arguments have to match the formatting specification. The typed sscanf takes the input string, the format specification and the consumer function. It parses data from the string according to the formatting specification, passing them to the consumer. The number and the types of the arguments to the consumer function have to match the formatting specification. Since parsing is necessarily partial, sscanf should return the result of the consumer function in the **Maybe** monad. Here are a few examples of formatting and parsing; the comment underneath an expression shows its result:

```
tp1 = sprintf $ lit "Hello world"
-- "Hello world"
ts1 = sscanf "Hello world" (lit "Hello world") ()
-- Just ()

tp2 = sprintf (lit "Hello " ^ lit "world" ^ char) '!'
-- "Hello world!"
ts2 = sscanf "Hello world!" (lit "Hello " ^ lit "world" ^ char) id
-- Just '!'
```

A formatting specification is built by connecting the primitive specifications lit "string", int, and char with (^). Here is a more elaborate example demonstrating that sprintf and sscanf may use exactly the same formatting specification, which is a first-class value, fmt3

```
fmt3 () = lit "The value of " ^ char ^ lit " is " ^ int
tp3 = sprintf (fmt3 ()) 'x' 3
-- "The value of x is 3"
```

```
ts3 = sscanf "The value of x is 3" (fmt3 ()) (\c i → (c, i))
-- Just ('x',3)
```

(The () in the fmt3 definition keeps the type of fmt3 polymorphic and avoids the monomorphism restriction.) The formatting specification is typed: whereas

```
sprintf $ lit "Hello world"
```

has the type String,

```
sprintf $ lit "The value of " ^ char ^ lit " is " ^ int
```

has the type Char → Int → String .

The typed sprintf problem has been investigated extensively: the first solution was shown by Danvy [8], with more proposed by Hinze [18] and Asai [1]. The typed sscanf problem received significantly less attention, if any. The implementation of the typed sprintf and sscanf sharing the same formatting pattern specification is new.

We solve the problem by regarding the format specification as a domain-specific language and embedding it in Haskell 2010, see the file PrintScanF.hs. The language has to be typed. We get the idea of the type system from the types of sample sprintf and sscanf expressions:

```
sprintf (lit "xxx")      :: String
sprintf int              :: Int → String
sprintf (char ^ int )    :: Char → Int → String

sscanf inp (lit "xxx")   :: x → Maybe x
sscanf inp int           :: (Int → x) → Maybe x
sscanf inp (char ^ int ) :: (Char → Int → x) → Maybe x
```

The occurrence of int in the formatting pattern matches Int → in the types of the sprintf and sscanf expressions. A formatting specification hence corresponds to a type function, or a functor [18]. The composition of the specifications corresponds to the composition of the functors. One may view the specification int as having an "effect" of formatting or parsing an integer; the effect is reflected in expression's type. The connection between typed formatting and effects (specifically, delimited control effects) has been well explained in [1].

Until recently Haskell has not directly supported type functions. Therefore Hinze [18] represented functors indirectly, by associating them with ordinary types, their 'codes':

$\lambda\tau.\ \tau$	**data** Id
$\lambda\tau.\ x \to \tau$	**data** From x
$\lambda\tau.\ f_1(f_2\tau)$	**data** C f1 f2

The application of a functor is likewise indirect, requiring the interpretation of functor's code; the interpreter was written as a multi-parameter type class with functional dependencies. We show a more direct representation of functors, expressible already in Haskell 2010, taking inspiration from the well-known encoding of linear lambda terms in Prolog. Uninstantiated logic variables are used

for lambda-bound variables; a lambda-abstraction is represented as a pair of the logic variable and abstraction's body. Substitution for a lambda-bound variable is done by unifying the corresponding logic variable with the replacement term. The same trick applies to Haskell bearing in mind that type variables act like logic variables, with the type checker doing unifications. Assuming a binary type constructor F, we represent the identity functor by a Haskell term of type `F t t` and the functor $\lambda\tau.\ \texttt{Int} \to \tau$ by a term of type `F t (Int → t)`. The composition operation then is represented by the term of type `F b c → F a b → F a c`. We come to the following typed tagless final embedding of the language of formatting patterns:

```
class FormattingSpec repr where
    lit  :: String → repr a a
    int  :: repr a (Int → a)
    char :: repr a (Char → a)
    (^)  :: repr b c → repr a b → repr a c
infixl 5 ^
```

The typed sprintf interprets the formatting specification as a function that receives the values to format and produces a string. The existing solutions to the typed sprintf problem can be written, it turns out, as such interpreters, or the instances of FormattingSpec. We pick Asai's [1] accumulator-less variant of Danvy's functional unparsing, which corresponds to interpreting repr as FPr:

```
newtype FPr a b = FPr ((String → a) → b)
```

The type on the right-hand-side should look familiar, reminding us of the right Kan extension explained in Hinze's course at the school. .

```
instance FormattingSpec FPr where
    lit  str = FPr $ \k → k str
    int      = FPr $ \k → \x → k (show x)
    char     = FPr $ \k → \x → k [x]
    (FPr a) ^ (FPr b) = FPr $ \k → a (\sa → b (\sb → k (sa ++ sb)))

sprintf :: FPr String b → b
sprintf (FPr fmt) = fmt id
```

The signature of sprintf demonstrates the application of the functor associated with fmt to String. The resulting type becomes the type of sprintf fmt.

The interpreter for sscanf is somewhat dual:

```
newtype FSc a b = FSc (String → b → Maybe (a,String))

instance FormattingSpec FSc where
    lit  str = ...
    char     = FSc $ \inp f → case inp of
                                (c:inp) → Just (f c, inp)
                                ""      → Nothing
```

```
int       = ...
(FSc a) ^ (FSc b) = FSc $ \inp f →
    maybe Nothing (\(vb,inp') → b inp' vb) $ a inp f

sscanf :: String → FSc a b → b → Maybe a
sscanf inp (FSc fmt) f = fmap fst $ fmt inp f
```

We have seen sample applications of these sprintf and sscanf at the beginning of the section. See the complete code PrintScanF.hs for detail and more examples.

We may write other interpreters of FormattingSpec, for example, to store the formatting result to or take the input string from a file or a communication channel, or to convert the formatting pattern to a C-style formatting string. We may also enrich our language with primitive specifications for field width, precision, padding, etc. – without breaking the existing interpreters.

4.3 Linear and affine lambda-calculi

The second example of a typed tagless final embedding of a language with a non-Hindley-Milner type system deals with the typed linear lambda calculus, which requires each bound variable be referenced exactly once in an abstraction's body. Haskell will statically reject as ill-typed the attempts to represent abstractions whose bound variable is referenced several times – or, as in the K combinator, never.

We build on the embedding of simply typed lambda calculus with de Bruijn indices, §3.3. Recall that an object term of the type t was represented as a value of the type Symantics repr ⇒ repr h t where h stands for the type environment assigning types to free variables ('hypotheses') of a term. Linear lambda calculus regards bound variables as representing resources; referencing a variable consumes the resource. We use the type environment for tracking the state of resources: available or consumed. The type environment becomes 'type state'. The file LinearLC.hs defines the embedded calculus and its two interpreters, to evaluate and to show linear lambda terms. The code demonstrates extensions relaxing linearity.

4.4 Call-by-name, call-by-value, call-by-need

Among the different interpreters of the embedded language may also be different evaluators. Each evaluator maps a term to its Haskell value, each evaluator is type-preserving, neither gets stuck. The evaluators may differ in evaluation strategies. Evaluating the same term with different strategies helps us compare them and assess their efficiency. (If the object language has effects, different strategies differ not only in efficiency but also in results.) Our evaluators so far have been call-by-name, inheriting the evaluation strategy from the meta-language. We now show call-by-value and call-by-need (or, lazy) evaluators. The latter does fit within the typed tagless final framework. The three evaluators are quite alike, sharing most of the code. The only difference among them is the

interpretation for lam, the denotation assigned to object language abstractions. For that reason, lam is moved to a separate class

```
class SymLam repr where
  lam :: (repr a → repr b) → repr (repr a → repr b)
```

Evaluators instantiate repr to be S l m:

```
newtype S l m a = S { unS :: m a } deriving (Monad, MonadIO)
```

where m is a MonadIO and l is the label for the evaluation strategy: Name, Value, or Lazy. We use **IO** solely to print the evaluation trace, to clearly see the differences in the evaluation order. The three instances of SymLam, for the three evaluation strategies, are as follows:

```
instance Monad m ⇒ SymLam (S Name m) where
  lam body = return (\x → body x)

instance Monad m ⇒ SymLam (S Value m) where
  lam body = return (\x → (body ∘ return) =≪ x)

instance MonadIO m ⇒ SymLam (S Lazy m) where
  lam body = return (\x → body =≪ share x)
```

The call-by-name lam substitutes its argument expression, unevaluated, into the body. The call-by-value lam evaluates the argument expression first and substitutes its result. The call-by-need lam substitutes the memoized argument expression: the expression will be evaluated on first reference and the result remembered for further references. The memoization is performed by the function share :: MonadIO m ⇒ m a → m (m a), similar to the one described by Fischer et al. [12]. The complete code with several examples demonstrating the difference in efficiency among the strategies is in the file CBAny.hs. (There is also a file with the Haskell 2010 version of the code.)

4.5 Typed ordinary and one-pass CPS transforms

We turn from evaluators and pretty-printers to transformers. Transformers take an embedded language term and return another term, that is, the Haskell value representing the term in the typed tagless final approach. We must be able to interpret the result multiple times, with any existing and future interpreter of the language. In particular, the result can be transformed again.

The result of transforming a well-typed term ought to be well-typed. The typed tagless final approach clearly fulfills that requirement: after all, only well-typed terms are expressible. We impose therefore a more stringent requirement that the transformation be total. In particular, the fact that the transformation handles all cases of source terms must be patently, syntactically clear. The complete coverage must be so clear that the metalanguage compiler should be able to see that, without the aid of extra tools. The new requirement is also easy to fulfill in the typed tagless final approach: term transformers are expressed as

interpreters (after all, interpretation is all we can do with embedded terms in the final approach) and typed tagless final interpreters do not get stuck.

We picked as an example typed call-by-value Continuation Passing Style (CPS) transform, which is profound, fascinating, and complex – especially for a typed higher-order language since the source and the result of the transform have different types and the transformation on types does not commute with the arrow type constructor. Our transformer is typing- (rather than type-) preserving and is patently total.

The ordinary (Fischer or Plotkin) CPS transform introduces many administrative redices, which make the result too hard to read. Danvy and Filinski [10] proposed a one-pass CPS transform, which relies on the metalanguage to get rid of the administrative redices. (See Washburn and Weirich [42] for the Haskell implementation. Like the original one-pass transform, it deals with untyped lambda-calculus.) The one-pass CPS transform can be regarded as an example of normalization-by-evaluation.

The complete, commented source code for the ordinary and one-pass typed transforms is in the file CPS.hs. The code has many examples, in particular, demonstrating that a CPS-transformed term can be interpreted by the CPS transformer again, yielding 2-CPS terms, etc. CPS transformers are composable, as expected. The initial approach to typed CPS is described in [15].

4.6 Type-directed partial evaluation

We started the fun section with a producer of embedded language terms, the de-serializer/type-checker in §4.1. We end on a similar note, with a producer of object terms of type t from Haskell values of type t. The type of the producer makes it look like an inverse of an object language evaluator, which is bewildering. The magic is explained in detail in Danvy's lecture notes on type-directed partial evaluation [9] and, in brief, in the comments in the source code. It helps that the type t of the values to un-evaluate must be polymorphic, and that the un-evaluator, called reify, is a family of functions, indexed by t. For example, for the type a→ a where a is any base type, the reify function has the type Symantics repr ⇒ (a → a) → repr (a → a). It converts a Haskell a→ a function (which is an opaque value that we cannot print) to a Haskell value repr (a→ a) representing an embedded language term, which we *can* print. Curtly, reify converts from a denotation to notation, hence the name.

Danvy's original 1996 presentation of the technique, expounded in [9], used an untyped target language represented as an algebraic data type. Type preservation was not apparent and had to be proved. The typed tagless final presentation makes type preservation patent, verified by the Haskell type checker. In the tagless-final presentation, reification and its dual reflection appear particularly symmetric, elegant and insightful.

The file TDPE.hs contains the source code and the comments explaining the derivation of reify and reflect. The imported ToTDPE.hs defines a few sample functions to reify. Compiling this module makes for a nicer example, demon-

strating that we can reify and hence pretty-print Haskell functions that have been compiled into machine code and for which we have no source.

5 Conclusions and Lessons

It has been argued and repeatedly agreed that domain-specific languages (DSLs) are well-worth dedicating several conference series to. Embedding DSLs in a host language is a good way of implementing them [20]. Typed tagless final approach is a good way of embedding typed DSLs and writing their interpreters. The interpreters are patently type preserving, efficient, and do not get stuck. The final approach can express pattern-matching and seemingly non-compositional processing. The strength of the final approach is in its extensibility, letting the programmer add new interpreters for the language and new forms to the language, without breaking, or even re-compiling, the existing code. The final approach lets us define a DSL and make use of it incrementally.

We have learned to appreciate type-constructor polymorphism as a mechanism for translucent abstractions. The parameterization over the type constructor repr lets just enough information out, to type-check an embedded term, but hides the representation details, thus permitting many and varied interpretations. We have confirmed the old insight from denotational semantics that making context explicit turns seemingly non-compositional operations compositional. We have also learned that the typed tagless final approach often leads to simpler-typed DSL embeddings, requiring a less sophisticated type system of the metalanguage, compared to the initial approach.

The typed tagless final approach, compiled and polished in [6], has been further illustrated, developed, exposed and refined in a number of recent publications [2, 3, 19, 26, 39]. The rich history of the approach has been reviewed in detail in [6]. The very similar insight has been independently developed in linguistics under the name of Abstract Categorial Grammars (ACG) [34].

Acknowledgements

I thank the organizers and the participants of the Spring School. I am thankful to Jeremy Gibbons and Stephanie Weirich for great many helpful comments. Many fruitful discussions with Chung-chieh Shan are gratefully appreciated.

Bibliography

[1] Asai, Kenichi. 2009. On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22(3):275–291.

[2] Atkey, Robert. 2009. Syntax for free: Representing syntax with binding using parametricity. In *TLCA 2009: Proceedings of the 9th international conference on typed lambda calculi and applications*, ed. Pierre-Louis Curien, 35–49. Lecture Notes in Computer Science 5608, Berlin: Springer.

[3] Atkey, Robert, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, ed. Stephanie Weirich, 37–48. New York: ACM Press.

[4] Baars, Arthur I., and S. Doaitse Swierstra. 2002. Typing dynamic typing. In [21], 157–166.

[5] Böhm, Corrado, and Alessandro Berarducci. 1985. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science* 39: 135–154.

[6] Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19(5):509–543.

[7] Church, Alonzo. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5(2):56–68.

[8] Danvy, Olivier. 1998. Functional unparsing. *Journal of Functional Programming* 8(6):621–625.

[9] ———. 1999. Lecture notes on type-directed partial evaluation. `http://www.brics.dk/~danvy/tdpe-ln.pdf`.

[10] Danvy, Olivier, and Andrzej Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

[11] Ertl, Anton. 2008. Threaded code. `http://www.complang.tuwien.ac.at/forth/threaded-code.html`.

[12] Fischer, Sebastian, Oleg Kiselyov, and Chung-chieh Shan. 2009. Purely functional lazy non-deterministic programming. In *ICFP '09: Proceedings of the ACM international conference on functional programming*, ed. Graham Hutton and Andrew P. Tolmach, 11–22. New York: ACM Press.

[13] Garrigue, Jacques. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*.

[14] Gries, David, ed. 1978. *Programming methodology: A collection of articles by members of IFIP WG 2.3*. Berlin: Springer.

[15] Guillemette, Louis-Julien, and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In *ICFP '08: Proceedings of the ACM international conference on functional programming*, ed. James Hook and Peter Thiemann, vol. 43(9) of *ACM SIGPLAN Notices*, 75–86. New York: ACM Press.

[16] Gunter, Carl A., and John C. Mitchell, eds. 1994. *Theoretical aspects of object-oriented programming: Types, semantics, and language design*. Cambridge: MIT Press.

[17] Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* 18(2):109–138.

[18] Hinze, Ralf. 2003. Formatting: A class act. *Journal of Functional Programming* 13(5):935–944.

[19] Hofer, Christian, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of GPCE 2008: 7th international conference on generative programming and component engi-*

neering, ed. Yannis Smaragdakis and Jeremy G. Siek, 137–147. New York: ACM Press.

[20] Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es):196.

[21] ICFP. 2002. *ICFP '02: Proceedings of the ACM international conference on functional programming*. New York: ACM Press.

[22] Johann, Patricia, and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *POPL '08: Conference record of the annual ACM symposium on principles of programming languages*, ed. George C. Necula and Philip Wadler, 297–308. New York: ACM Press.

[23] Kamin, Samuel. 1983. Final data types and their specification. *ACM Transactions on Programming Languages and Systems* 5(1):97–121.

[24] Lämmel, Ralf, and Oleg Kiselyov. 2010. Exploring typed language design in Haskell: Lecture 1. Haskell's take on the Expression Problem. `http://userpages.uni-koblenz.de/~laemmel/TheEagle/`.

[25] ———. 2010. Lecture 2: Spin-offs from the Expression Problem. `http://userpages.uni-koblenz.de/~laemmel/TheEagle/resources/xproblem2.html`.

[26] Magi, Sandro. 2009. Mobile code in C# via finally tagless interpreters. `http://higherlogics.blogspot.com/2009/06/mobile-code-in-c-via-finally-tagless.html`.

[27] McAdam, Bruce J. 2001. Y in practical programs. Workshop on fixed points in computer science; `http://www.dsi.uniroma1.it/~labella/absMcAdam.ps`.

[28] Miller, Dale, and Gopalan Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE symposium on logic programming*, ed. Seif Haridi, 379–388. Washington, DC: IEEE Computer Society Press.

[29] Mu, Shin-Cheng. 2008. Typed λ-Calculus Interpreter in Agda. `http://www.iis.sinica.edu.tw/~scm/2008/typed-lambda-calculus-interprete/`.

[30] Oliveira, Bruno César dos Santos, and Jeremy Gibbons. 2005. TypeCase: A design pattern for type-indexed functions. In *Proceedings of the 2005 Haskell workshop*, 98–109. New York: ACM Press.

[31] Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In [21], 157–166.

[32] Pfenning, Frank, and Conal Elliott. 1988. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM conference on programming language design and implementation*, vol. 23(7) of *ACM SIGPLAN Notices*, 199–208. New York: ACM Press.

[33] Plotkin, Gordon D. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5:223–255.

[34] Pogodalla, Sylvain. Abstract Categorial Grammar homepage. `http://www.loria.fr/equipes/calligramme/acg`.

[35] Reynolds, John C. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New directions in al-*

gorithmic languages 1975, ed. Stephen A. Schuman, 157–168. IFIP Working Group 2.1 on Algol, Rocquencourt, France: INRIA. Also in [14, 309–317], [16, 13–23].

[36] Swierstra, Wouter. 2008. Data types á la carte. *Journal of Functional Programming* 18(4):423–436.

[37] Szabó, Zoltán Gendler. 2008. Compositionality. In *The Stanford Encyclopedia of Philosophy*, ed. Edward N. Zalta, Winter 2008 ed.

[38] Thiemann, Peter. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5):483–525.

[39] Thiemann, Peter, and Martin Sulzmann. 2010. Tag-free combinators for binding-time polymorphic program generation. In *Proceedings of FLOPS 2010: 10th international symposium on functional and logic programming*, ed. Matthias Blume, Naoki Kobayashi, and Germán Vidal, 87–102. Lecture Notes in Computer Science 6009, Berlin: Springer.

[40] Wadler, Philip. 1998. The expression problem. Message to java-genericity electronic mailing list. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`.

[41] Wand, Mitchell. 1979. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences* 19(1):27–44.

[42] Washburn, Geoffrey Alan, and Stephanie Weirich. 2008. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming* 18(1):87–140.

[43] Weirich, Stephanie. 2001. Encoding intensional type analysis. In *Esop*, ed. David Sands, vol. 2028 of *LNCS*, 92–106. Springer.

[44] ———. 2004. Typechecking into GADT. `http://www.comlab.ox.ac.uk/projects/gip/school/tc.hs`.

[45] ———. 2006. Type-safe run-time polytypic programming. *Journal of Functional Programming* 16(6):751–791.

[46] Winskel, Glynn. 1993. *Formal semantics of programming languages*. Cambridge: MIT Press.

[47] Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 224–235. New York: ACM Press.

[48] Xi, Hongwei, and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM conference on programming language design and implementation*, vol. 33(5) of *ACM SIGPLAN Notices*, 249–257. New York: ACM Press.

[49] Yang, Zhe. 1998. Encoding types in ML-like languages. In *ICFP '98: Proceedings of the ACM international conference on functional programming*, vol. 34(1) of *ACM SIGPLAN Notices*, 289–300. New York: ACM Press.