Lambda: the ultimate syntax-semantics interface[⋆]

Oleg Kiselyov[1] and Chung-chieh Shan[2]

[1] oleg@okmij.org
[2] Indiana University ccshan@indiana.edu

Abstract. Spreadsheets in accounting and Matlab in engineering are immensely popular because these glorified calculators let domain experts *play*: write down a problem in familiar terms and quickly try a number of solutions. Natural-language semanticists have a better tool. Not only does it compute grammar yields and truth values, it also infers types, normalizes terms, and displays truth conditions as formulas. Its modularity facilities make it easy to try fragments out, scale them up, and abstract encoding details out of semantic theories.
This tool is a combination of techniques created by functional programmers, who are as unaware of its application to semantics as most semanticists. This paper breaks the barrier. We express *extensible interpreters* of natural- and formal-language fragments as functional programs. Specifically, we work our way from the simply-typed lambda calculus and a context-free grammar to a dynamic treatment of quantification and anaphora. We strive to be comprehensible and informative to both linguists and programmers.

1 Introduction

A lecture on natural-language semantics started with this exchange:

> Lecturer: Did you find lambda-conversion difficult?
> Audience: YES.
> Lecturer: Sorry. This is the main technical tool. Without it, nothing else makes sense. Let's go back to the slide with long lambda-conversions, and go through them again. Slowly.

Our sympathy to the audience motivates this work. Although it is important to be able to do lambda-calculus conversions by hand, it is boring and error-prone to do them by hand all the time, as semanticists typically do. Hand calculations condemn the researcher to trying out a mere handful of simple examples, in danger of overlooking more complex counter-examples and the feature interactions they incur. The same danger that Karttunen [13] noted in phonology threatens semantics: computational support is badly needed to ensure that changing one's theory to fix one case doesn't break five other cases.

[⋆] This paper is based on a short course at the 2010 North American Summer School in Logic, Language, & Information. We thank the school organizers and class attendees.

It already helps to have a simple calculator that accepts lambda terms containing pre-defined domain constants and computes their normal forms. The Penn Lambda Calculator [3] answers this need. But more tasks should be automated and more patterns abstracted. For example, besides adding domain constants and lexical entries, a researcher may want to add intensionality to an existing grammar by uniformly lifting every rule's type. Moreover, it is common to lift the types several times, which usually turns an intuitive set of terms into an inscrutable nest of lambdas. To work with such grammars, the researcher may develop abbreviations to hide trivial encodings and administrative distractions.

The ideal calculator should automate uniform changes to the grammar and accept as well as display abbreviated terms. However, today's variety of semantic theories is too wide for any given calculator to anticipate and support even a significant fraction of them as built-in features. Instead, we need a calculator that is *programmable* by users. In fact, a programmable calculator that automates and hides irrelevant details can not only relieve semanticists of tedium, but also enable them to disavow that the details matter at all and thus formulate more general theories at higher levels of abstraction. In other words, a richer metalanguage, with better abstraction facilities, enables building richer theories while retaining the confidence and convenience that formalization affords.

Contributions This paper shows that such an ideal tool already exists—unbeknownst to its developers. The tool is based on a well-developed, well-documented, and well-maintained functional programming language with an interactive top-level. We use the Haskell language for illustration, but Standard ML and OCaml would work too. Applying some recently collected and polished techniques [2], we embed into this language several *domain-specific languages* (DSLs) of interest, such as English and the lambda calculus. Our contribution, besides explaining these techniques to an audience of natural-language semanticists, is to encode natural-language fragments and their interpretations as programming-language modules and thus to extend and relate them clearly and concisely. We demonstrate computing and printing not only truth values but truth conditions, on various levels of abstraction.

We explain the tool in tutorial style by a series of examples. In each example, we derive a set of well-formed expressions (as in a context-free grammar) and interpret the derivations in several ways to calculate their forms (such as the English strings they yield) and meanings (such as the first-order formulas they denote) automatically. Our theme is thus *calculemus*: let's turn manual reasoning into automatic calculation, to ease formulating theories and verifying hypotheses.

Organization Reflecting how we propose linguistic theories can and should be expressed, our series of examples develops many fragments of natural and formal languages. We start with a trivial fragment of English in §2, then express simply-typed lambda-terms in §3. In our code (online at `http://okmij.org/ftp/gengo/NASSLLI10/`), we add quantification, relative clauses, and pronouns. We use these additions to explain de Groote's dynamic analysis of donkey anaphora. The *metalanguage* in which we encode these analyses is Haskell, not English, so the analyses are precisely described and mechanically executable.

2 Calculating with grammars

In this section we introduce Haskell and show how to use it as a calculator to express linguistic derivations as programs. The *form* of the programs is intuitive in that it resembles familiar notation and thus makes our intentions clear.

2.1 Calculating yields and truth values

Haskell can calculate with numbers, booleans, and strings. The calculator is called GHCi and is a part of the Haskell Platform. The calculator starts by giving the prompt `Prelude>`, which means the standard Prelude functions are available. We can enter expressions (shown in italics) and see the result underneath:

```
Prelude> "John" ++ " came"
"John came"
```

We have calculated the concatenation of two strings, using the standard infix operator `++` for concatenation. The infix notation, albeit convenient, is not necessary: writing any operator in parentheses exposes it as a regular function:

```
Prelude> (++) "John" " came"
"John came"
```

Our expression is built of mere applications, written as juxtaposition. Actually `(++)` is a curried function: it is a function that maps `"John"` to a function that maps `" came"` to `"John came"`. Accordingly, juxtaposition for application associates to the left. There are also abstractions, with λ written as a backslash:

```
Prelude> (\np -> np ++ " " ++ "came") "John"
"John came"
```

The calculator has performed a β-reduction, substituting the string `"John"` for the variable `np`, followed by the same string concatenation as before.

Haskell is typed; the types are inferred and remain implicit. We become aware of types when we enter an ill-typed expression, such as `not "John"`, and see a type error message. We can also ask GHCi to show us the type it infers, using the `:t` command. (Some Haskell environments instead let us simply select an expression to ask for its type.)

```
Prelude> :t (\np -> np ++ " " ++ "came")
(\np -> np ++ " " ++ "came") :: [Char] -> [Char]
```

The abstraction has the type `[Char] -> [Char]`, indeed the type of a function from character lists (`[Char]`, which can also be written `String`) to strings.

One strength of the metalanguage is the ability to *name* frequently occurring expressions. One can think of such definitions as shortcuts or bookmarks. One way to name an expression for later reuse is to put its definition into into a file. For example, we can put the following definitions into a file `CFG1EN.hs`:

```
john     = "John"
mary     = "Mary"
like     = "likes"
r2 tv np = tv ++ " " ++ np
r1 np vp = np ++ " " ++ vp
```

```
sentence = r1 john (r2 like mary)
```

These definitions associate the name `john` with the string `"John"` and so on. The definition `r2` is parameterized by two arguments, `tv` and `np`. It is equivalent to defining a function explicitly: `r2 = \tv -> \np -> tv ++ " " ++ np`, or for short `r2 = \tv np -> tv ++ " " ++ np`. The definition `r1` is similar; both `r2` and `r1` abbreviate the concatenation of two arguments with a separating space. The final definition `sentence` abbreviates a familiar expression.

The *form* of these definitions is meant to express a context-free grammar:

NP → John	(john)	NP → Mary	(mary)	TV → likes	(like)
VP → TV NP	(r2)	S → NP VP	(r1)		

The definition `sentence` expresses a derivation in this CFG. We can calculate its *yield* by loading the file into GHCi and asking what `sentence` abbreviates:

```
Prelude> :load CFG1EN.hs
*CFG1EN> sentence
"John likes Mary"
```

Besides yields, we can associate our CFG notation with different computational interpretations. In particular, we can interpret a CFG derivation as a truth value in a particular model. In a different file `CFG1Sem.hs`, we enter

```
data Entity = John | Mary deriving (Eq, Show)

john     = John
mary     = Mary
like     = \o s -> elem (s,o) [(John,Mary), (Mary,John)]
r2 tv np = tv np
r1 np vp = vp np

sentence = r1 john (r2 like mary)
```

Our domain has two entities `John` and `Mary`, which can be compared for equality (`Eq`) and displayed (`Show`). A semanticist would write $D_e = \{\mathbf{john}, \mathbf{mary}\}$. We define the same names as before, but we let them denote not strings but semantic values such as entities (for `john` and `mary`), relations (for `like`, using a built-in function `elem`), and truth values (for `r1 np vp`, by applying a property `vp` to an entity `np`). The definition `sentence` looks exactly the same as before. However, if we load `CFG1Sem.hs` into GHCi and ask for `sentence`, we no longer see a string. We see a boolean `True`, telling us that `John` likes `Mary` in our model.

2.2 Expressing syntactic categories

Next, we teach our calculator to check our syntactic categories for us. This move begins our journey to expressing theories at a higher level of abstraction.

Ideally, the grammar we intend to express should correspond exactly to how the calculator executes our program. In `CFG1EN.hs` for example, every string abbreviation should denote the yield of a valid derivation. So far, the correspondence is shallow: we meant `r1` as a rule for combining an NP with a VP and chose the variable names `np` and `vp` accordingly, yet to GHCi these names are opaque and `r1` is just a function of type `String -> String -> String`. Thus, we can write nonsensical expressions such as `r1 (r2 like mary) john`, even though `r2 like mary` is not an NP and `john` is not a VP. The expression type-checks and evaluates to a string. It shouldn't. With the semantic interpretation in `CFG1Sem.hs`, the same expression is rejected by the type checker, but other nonsensical expressions such as `r2 (r2 like mary) john` still slip through.

Our calculator should reject invalid derivations when we try to build them, rather than when we try to interpret them. Following Russell and Church, we use types to delineate the set of *meaningful* expressions – those that represent valid derivations. The types will tell the calculator to check in derivations that `r2` is used as a CFG rule and `like` is used as a terminal as we intend them to be.

In the file `CFG3EN.hs`, we define new types `S`, `NP`, etc. for syntactic categories:

```
data S; data NP; data VP; data TV
```

We then use these types to annotate syntactic derivations:

```
data EN a = EN { unEN :: String }
```

To the left of the equal sign, `EN a` means to define a new type `EN a` for each type `a`, such as `EN S`, `EN NP`, etc. To the right of the equal sign, we define two functions that are inverses of each other: `EN` from `String` to `EN a` and `unEN` from `EN a` to `String`. One may thus view a type just defined, such as `EN NP`, as `String` annotated with `NP`, as if `NP` were a grammatical feature.

We now annotate the yield interpretation with syntactic categories. The definitions for `john`, `r2`, etc. remain essentially the same, with a sprinkling of the 'annotation glue' `EN`. After all, we did not intend to change the grammar.

```
john, mary :: EN NP
like       :: EN TV
r2         :: EN TV -> EN NP -> EN VP
r1         :: EN NP -> EN VP -> EN S

john               = EN "John"
mary               = EN "Mary"
like               = EN "likes"
r2 (EN tv) (EN np) = EN (tv ++ " " ++ np)
r1 (EN np) (EN vp) = EN (np ++ " " ++ vp)

sentence :: EN S
sentence = r1 john (r2 like mary)
```

New are the type annotations (written with `::`), assigning explicit types to the defined names. The types look like grammar rules! For example, the type of `r2` can be read as `VP` $\rightarrow$ `TV NP`. This type says explicitly that `r2` should only be used to combine a TV and an NP to form a VP, so GHCi rejects invalid derivations:

```
*CFG3EN> r2 (r2 like mary) john
... Couldn't match expected type 'TV' against inferred type 'VP' ...
    In the first argument of 'r2', namely '(r2 like mary)'
```

The message clearly describes the type error in terms of the grammar. We can also ask GHCi to show the syntactic category it infers for a valid derivation.

2.3 Type functions: from syntactic categories to semantic types

We just annotated the calculation of yields with syntactic categories. Likewise, in `CFG3Sem.hs` we annotate the calculation of truth values with syntactic categories:

```
data S; data NP; data VP; data TV

type family Tr (a :: *) :: *
type instance Tr S  = Bool
type instance Tr NP = Entity
type instance Tr VP = Entity -> Bool
type instance Tr TV = Entity -> Entity -> Bool

data Sem a = Sem { unSem :: Tr a }
```

Like `EN a` in §2.2, `Sem a` stands for an annotated type. Whereas `EN a` is always isomorphic to `String` because every derivation yields a string, the type that `Sem a` is isomorphic to depends on `a`: we interpret derivations of category `NP` as entities, derivations of category `S` as truth values (`Bool`), and so on. In Haskell, we specify this dependence as cases of a type function, `Tr`, mapping each category label to a semantic type. The definitions for `john`, `r2`, etc. are same as in `CFG1Sem.hs`, except with type annotations and the 'annotation glue' `Sem` added.

2.4 Unifying form with meaning

So far we have been keeping our definitions for yields and for truth values in separate files, even though the two sets of definitions are quite similar and it is bothersome to keep switching between loading two different files into GHCi. Below are the similar parts of `CFG3EN.hs` and `CFG3Sem.hs`, side by side:

```
data S; data NP; data VP; data TV    data S; data NP; data VP; data TV

john, mary :: EN NP                  john, mary :: Sem NP
like       :: EN TV                  like       :: Sem TV
r2   :: EN TV -> EN NP -> EN VP      r2   :: Sem TV -> Sem NP -> Sem VP
r1   :: EN NP -> EN VP -> EN S       r1   :: Sem NP -> Sem VP -> Sem S

sentence :: EN S                     sentence :: Sem S
sentence = r1 john (r2 like mary)    sentence = r1 john (r2 like mary)
```

Both sides define the same syntactic categories and use them to annotate the same terminals, rules, and sample derivation in the same way. As is rightful, the only difference is in how the terminals, rules, and sample derivation are interpreted: `EN` vs. `Sem`. The definition of `sentence` on the last line is also identical.

Repeated code is a tell-tale sign of an abstraction struggling to get out. We proceed to factor the common code out of the two interpretations. We first factor out the category annotations on terminals and rules by abstracting over the interpretations `EN` and `Sem`. Haskell provides just the right facility for such an abstraction: a type class. In a single new file `CFG4.hs`, we enter the code below.

```
data S; data NP; data VP; data TV
```

```
class Symantics repr where
  john, mary :: repr NP
  like       :: repr TV
  r2         :: repr TV -> repr NP -> repr VP
  r1         :: repr NP -> repr VP -> repr S
```

We replaced `EN` and `Sem` with a variable `repr`, which can be instantiated to `EN`, `Sem`, or something else. The type class `Symantics` lets us assign categories to terminals and rules once and for all, regardless of their possible interpretations.

Second, we write and type-check derivations once and for all as well, regardless of – in other words, abstracting over – their possible interpretations.

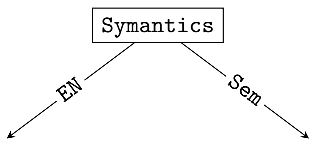
```
sentence = r1 john (r2 like mary)
```

The inferred type of `sentence` is `Symantics repr => repr S`, which says that, in *any* interpretation `repr`, the derivation has the type that `repr` assigns to `S`.

Finally, we define each way to interpret derivations as an instance of the type class `Symantics`. To the right is a map of `Symantics` and its two interpretations.

```
data EN a = EN { unEN :: String }
instance Symantics EN where
  john            = EN "John"
  ...             = ...
  r1 (EN x) (EN f) = EN (x ++ " " ++ f)

data Sem a = Sem { unSem :: Tr a }
instance Symantics Sem where
  john              = Sem John
  ...               = ...
  r1 (Sem x) (Sem f) = Sem (f x)
```



We can then calculate the yield and truth value of the *same* derivation `sentence` by instantiating the variable `repr` in the type of `sentence` to either `EN` or `Sem`:

```
*CFG4> unEN (sentence :: EN S)
"John likes Mary"
*CFG4> unSem (sentence :: Sem S)
True
```

Because type classes in Haskell are *open*, at any time we can add a new way to interpret derivations (as a new instance of `Symantics`) and re-interpret existing derivations without repeating them. In particular, at the end of the next section we add a new interpretation to see truth conditions as formulas, not just `True`.

Factoring out repetitive code in this way not only saves us typing but also adds a language of derivations to our calculator notation. This object language of derivations consists of what all interpretations have in common: the type system enforces the abstraction barrier between derivations and interpretations by rejecting derivations that depend on a particular interpretation. As many linguistic theories exhort [10, 17, 19], form and meaning must build in tandem. In particular, because for simplicity our `EN` interpretation only concatenates plain strings, our example grammar here can be understood as a *combinatory* *categorial grammar* (CCG) [20, 21] embedded in Haskell. In general, for form and meaning to build in tandem is well supported by how we embed an object language in a functional programming language: our notation for derivations looks like a typical linear notation and are executable. (Derivations can also be interpreted as trees familiar to linguists, as shown in Appendix A.)

3 The syntax and interpretations of semantics

We have embedded in Haskell a small fragment of English and interpreted its derivations in two ways. The same approach – representing valid derivations as well-typed programs and using a type class to abstract over interpretations – applies to formal languages too. We take as our example Church's Simple Theory of Types [4]. (The approach easily extends to multisorted logics such as Ty_2 [6, 7], popular in natural-language semantics.) As in §2, we represent well-formed STT formulas as well-typed Haskell programs that can be interpreted in several ways: evaluated in a model, printed, and simplified. The last two interpretations will let us see the truth conditions of our English derivations as simplified formulas.

STT is the simply-typed lambda calculus with two base types, `Entity` and `Bool`, and a few constants. The language is a higher-order predicate logic. Just as we defined a CFG using the type class `Symantics` in §2, we define the grammar of STT using a type class `Lambda` containing constants (`true`, `mary'`), connectives (`neg`, `conj`), abstraction, and application. The code below is in `Semantics.hs`.

```
class Lambda lrepr where
  john', mary' :: lrepr Entity
  like'        :: lrepr (Entity -> Entity -> Bool)
  true         :: lrepr Bool
  neg          :: lrepr Bool -> lrepr Bool
  conj         :: lrepr Bool -> lrepr Bool -> lrepr Bool
  exists       :: lrepr ((Entity -> Bool) -> Bool)
  app          :: lrepr (a -> b) -> lrepr a -> lrepr b
  lam          :: (lrepr a -> lrepr b) -> lrepr (a -> b)
```

We represent STT applications using `app`. For example, `app (app like' mary') john'` represents the formula (**like mary**) **john**. Following Church, `exists` is a higher-order constant, and `lam` encodes STT abstractions using Haskell ones. This *higher-order abstract syntax* (HOAS) [14, 18] represents STT variables as Haskell ones. Thus, the formula $\exists x.$ (**like mary**) x is represented as follows.

```
lsene :: Lambda lrepr => lrepr Bool
lsene = app exists (lam (\x -> app (app like' mary') x))
```

We have not defined `false` because it is `neg true`. Likewise, we define universal quantification with a restrictor predicate `r` as a 'macro' in terms of `exists`, then define the unrestricted `forall` in terms of the restricted `forall_`:

```
forall_ r = lam (\p -> neg (app exists
                        (lam (\x -> conj (app r x) (neg (app p x))))))
forall = forall_ (lam (\x -> true))
```

For example, the formula $\forall x. (\textbf{like mary})\, x$ is represented as follows.

```
lsenf :: Lambda lrepr => lrepr Bool
lsenf = app forall (lam (\x -> app (app like' mary') x))
```

We interpret these formulas in three ways, as mapped below to the right. The first is to evaluate them in a model, just as we evaluated English derivations in §2:

```
data R a = R { unR :: a }
instance Lambda R where
  john'           = R John
  ...             = ...
  app (R f) (R x) = R (f x)
  lam f           = R (\x -> unR (f (R x)))
```

$$\begin{tikzcd} & {} \\ \boxed{\text{Lambda}} \arrow[ur, "R"] \arrow[r, "C"] \arrow[loop below, "P"] & {} \end{tikzcd}$$

The second is to print them as text, but this interpretation (unlike the yield calculation in §2) cannot compose mere strings: we need two pieces of context, namely the number of variables already bound `i` (to generate fresh names) and the current precedence level `p` (to minimize printing parentheses).

```
data C a = C { unC :: Int -> Int -> String }
instance Lambda C where
  john'           = C (\i p -> "john'")
  ...             = ...
  conj (C x) (C y) = C (\i p -> if p > 3
                              then "(" ++ x i 4 ++ " ∧ " ++ y i 3 ++ ")"
                              else        x i 4 ++ " ∧ " ++ y i 3      )
  exists          = C (\i p -> "∃")
```

We can now choose whether to evaluate an expression to an under-informative truth value or to print it as an over-informative unnormalized formula:

```
*Semantics> lsenf :: R Bool
False
*Semantics> lsenf :: C Bool
(λx1. ¬(∃ (λx2. (λx3. ⊤) x2 ∧ ¬(x1 x2)))) (λx1. like' mary' x1)
```

For checking truth conditions, we want to see the formula with obvious simplifications applied such as β-reductions and removing conjunctions with ⊤ (`true`). In programming-language terms, we want to *partially evaluate* the formula. Such a third interpretation turns out to be easy and elegant to define using Haskell's existing computing machinery in types and terms [2]. Actually we define a family of interpretations `P lrepr`, parameterized by the interpretation `lrepr` for the partially evaluated formula. For lack of space, we only demonstrate the payoff:

```
*Semantics> lsenf :: (P C) Bool
¬(∃ (λx1. ¬(like' mary' x1)))
```

Finally we link the `Lambda` language to the `Symantics` language in §2. Our partial evaluator `P lrepr` translates `Lambda` to `Lambda` and interprets the result using `lrepr`. In `CFG.hs`, we likewise turn the `Sem` interpretation in §2.4 into an interpretation family `Sem lrepr` that translates `Symantics` to `Lambda` and interprets the result using `lrepr`. We can then use the pipeline `Sem (P C)` to calculate *simplified* truth *conditions*, such as `like' mary' john'` for `sentence`.

4 Growing languages and interpretations

Our approach scales up to the following map of languages (boxes) and interpretations (arrows). A hooked arrow means that a language is a subset of another.

$$\begin{tikzcd}
{} & \boxed{\texttt{Symantics}} \arrow[l, bend right, "\texttt{JA}"'] \arrow[l, bend left, "\texttt{EN}"] \arrow[r, "\texttt{Sem}"] \arrow[d, hook] & \boxed{\texttt{Lambda}} \arrow[r, bend left, "\texttt{R}"] \arrow[r, "\texttt{C}"'] \arrow[loop right, "\texttt{P}"] \arrow[dr, hook] \arrow[dr, bend left, "\texttt{D}"] \arrow[dd, hook] & {} & \\
{} & \boxed{\texttt{Quantifier}} \arrow[l, bend right, "\texttt{JA}"'] \arrow[l, bend left, "\texttt{EN}"] \arrow[ur, "\texttt{Sem}"] \arrow[d, hook] & & \boxed{\texttt{States}} \arrow[r, bend left, "\texttt{R}"] \arrow[r, "\texttt{C}"'] \arrow[loop below, "\texttt{P}"] & {} \\
{} & \boxed{\texttt{Pronoun}} \arrow[l, "\texttt{EN}"'] \arrow[r, "\texttt{Sem}"] & \boxed{\texttt{Dynamics}} \arrow[ur, "\texttt{D}"] \arrow[r, "\texttt{C}"'] \arrow[loop right, "\texttt{P}"] & {} &
\end{tikzcd}$$

To the English fragment on the left, we add context-free rules for quantifiers (whose restrictors may be subject relative clauses) and pronouns. The quantifiers fragment can also be spelled out in Japanese. (Another way to add quantifiers is to express Montague's *quantifying in* [15] using HOAS.) On the right, to explain what pronouns mean, we translate the `Dynamics` language (STT with a constant `it'` [12]) to the `States` language (STT with information states [11]). The step-by-step extensions are so modular, we can write a lexical entry for *every* without anaphora in mind, then reuse it to calculate simplified truth conditions for donkey sentences such as *every farmer who owns a donkey beats it* [9]. Appendix B gives more details. (Barker and Shan's account of donkey anaphora [1] and Moortgat's symmetric categorial grammar [16] can also be expressed.)

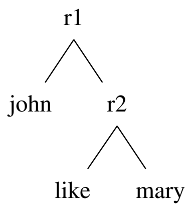
To conclude, we have showed a way to use computers to build and test linguistic theories that achieves two kinds of reuse and extensibility. First, we apply functional programming techniques to embed a syntax-semantics interface in a typed programming language. The language comes with mature tools for incremental development and interactive testing, which we use to calculate yields and meanings. Second, we use the language's modularity facilities to abstract derivations from ways to interpret them, and to grow our theory without repeating code or entangling notation. Our trivial fragment extends step by step to handle quantification and donkey anaphora. A single derivation can be interpreted in tandem to give a string in English or Japanese, a truth value in a model, and a formula in classical or dynamic logic. More language fragments and interpretations (as an audio file or a tree diagram) can be added at any time.

We stand on the shoulders of many existing linguistic formalisms in which form and meaning build in tandem: Montague grammar, *definite clause grammars* with semantics [17], *Grammatical Framework* [19], and *abstract categorial grammars* (ACGs) [10]. Our grammars are closest to ACGs because their order can be arbitrarily high, but we allow interpreting function types. Moreover, we make no attempt at parsing. Rather, we stress interactive execution and modular extension in a rational re-exposition of formal semantics. We hope it spurs more natural- and programming-language researchers to study together ideas [5] such as side effects, continuations, regions, quotation, and dependent types.

A Automatic visualization of derivation trees

The tree to the right is automatically generated using Graphviz [8] by an additional instance of `Symantics` defined in `CFG4.hs`.

```
sentence = r1 john (r2 like mary)
```



B Quantifiers and pronouns in a dynamic logic

This appendix details how to grow the languages and interpretations in §2 and §3 into an analysis of quantificational donkey anaphora.

B.1 Quantifiers

Starting with the English fragment in §2, we add a second transitive verb `own` for variety. (Instead of changing the type class `Symantics` defined in §2.4, we could define a new type class using the inheritance mechanism described below.)

```
class Symantics repr where
  john, mary :: repr NP
  like, own  :: repr TV
  r2         :: repr TV -> repr NP -> repr VP
  r1         :: repr NP -> repr VP -> repr S

instance Symantics EN where
  john            = EN "John"
  mary            = EN "Mary"
  like            = EN "likes"
  own             = EN "owns"
  r2 (EN f) (EN x) = EN (f ++ " " ++ x)
  r1 (EN x) (EN f) = EN (x ++ " " ++ f)
```

The semantics of this fragment is standard. We express it as lambda-terms that can be evaluated, printed, and simplified – as promised at the end of §3. The header `instance (Lambda lrepr) => Symantics (Sem lrepr)` below means to define an instance `Symantics (Sem lrepr)` for each instance `Lambda lrepr`.

```
data Sem lrepr a = Sem { unSem :: lrepr (Tr a) }

instance (Lambda lrepr) => Symantics (Sem lrepr) where
  john              = Sem john'
  mary              = Sem mary'
  like              = Sem like'
  own               = Sem own'
  r2 (Sem f) (Sem x) = Sem (app f x)
  r1 (Sem x) (Sem f) = Sem (app f x)
```

Next, we add quantifiers. Besides `everyone` and `someone`, which we treat as unrestricted quantifiers, we also add the restricted quantifiers `every` and `a`. The restrictors are expressed by common nouns such as `farmer` and `donkey`, so we add the new syntactic category `CN` for common nouns, alongside `QNP` for quantificational noun phrases. We also add `who`, which adjoins a subject relative clause (that is, a `VP`) to a `CN` to build a complex `CN`. Following Montague's rule numbering, an object `QNP` enters the derivation by the rule `r5`, and a subject `QNP` by the rule `r4`. The code below is in the file `QCFG.hs`. The header `class (Symantics repr) => Quantifier repr` means to define a type class `Quantifier` that *inherits* all the members of the type class `Symantics`.

```
data CN; data QNP

class (Symantics repr) => Quantifier repr where
  farmer, donkey    :: repr CN
  everyone, someone :: repr QNP
  every, a          :: repr CN -> repr QNP
  who               :: repr VP -> repr CN -> repr CN
  r5                :: repr TV -> repr QNP -> repr VP
  r4                :: repr QNP -> repr VP -> repr S

instance Quantifier EN where
  farmer           = EN "farmer"
  donkey           = EN "donkey"
  everyone         = EN "everyone"
  someone          = EN "someone"
  every (EN n)     = EN ("every " ++ n)
  a     (EN n)     = EN ("a " ++ n)
  who (EN r) (EN q) = EN (q ++ " who " ++ r)
  r5 (EN f) (EN x)  = EN (f ++ " " ++ x)
  r4 (EN x) (EN f)  = EN (x ++ " " ++ f)
```

We assign completely standard meanings to these new constructions.

```
type instance Tr CN  = Entity -> Bool
type instance Tr QNP = (Entity -> Bool) -> Bool

instance (Lambda lrepr) => Quantifier (Sem lrepr) where
  farmer               = Sem farmer'
  donkey               = Sem donkey'
  everyone             = Sem forall
  someone              = Sem exists
  every (Sem cn)       = Sem (forall_ cn)
  a     (Sem cn)       = Sem (exists_ cn)
  who (Sem r) (Sem q)   = Sem (lam (\x -> conj (app q x) (app r x)))
  r5 (Sem tv) (Sem qnp) = Sem (lam (\s -> app qnp
                              (lam (\o -> app (app tv o) s))))
  r4 (Sem qnp) (Sem vp) = Sem (app qnp vp)
```

Below is an example derivation.

```
sen5 = r4 (every (who (r5 own (a donkey)) farmer)) (r5 like (a donkey))
```

We calculate the yield and simplified truth conditions of this sentence.

```
*QCFG> sen5 :: EN S
every farmer who owns a donkey likes a donkey
*QCFG> sen5 :: Sem (P C) S
¬(∃ (λx1. farmer' x1 ∧ ∃ (λx2. donkey' x2 & own' x2 x1)
                      ∧ ¬(∃ (λx2. donkey' x2 ∧ like' x2 x1))))
```

It is because `Quantifier` inherits from `Symantics` that `sen5` above can reuse the definitions and interpretations of `donkey` and `farmer`. In fact, we can use any `Symantics` derivation (such as `sentence` in §2.4) as a `Quantifier` derivation.

B.2 Pronouns

Our final goal is to add pronouns. Our first step towards this goal, following de Groote [11], is to add to STT a base type `State` of information states, along with two operations on them, `update` and `select`. The `update` operation adds an `Entity` to a `State`, and the `select` operation retrieves an `Entity` from a `State`. Just as the type class `Quantifier` above inherits from `Symantics`, the new type class `States` below inherits from `Lambda`.

```
class (Lambda lrepr) => States lrepr where
  update :: lrepr Entity -> lrepr State -> lrepr State
  select :: lrepr State -> lrepr Entity
```

By defining a few instances (in `Dynamics.hs`, omitted here), we extend our interpretations of the `Lambda` language to the `States` language, so as to evaluate, print, and simplify lambda-terms that contain `update` and `select`. Below is a demonstration; `update` is printed as infix :: and `select` is printed as `sel`.

```
*Dynamics> lam (\e -> app (lam (\e' -> select (update john' e')))
                          (update mary' e)) :: (P C) (State -> Entity)
λx1. sel (john' :: mary' :: x1)
```

Our second step is to extend STT in a different direction by adding a constant `it'` of type `(Entity -> Bool) -> Bool`.

```
class (Lambda lrepr) => Dynamics lrepr where
  it' :: lrepr ((Entity -> Bool) -> Bool)
```

This new type class `Dynamics` expresses de Groote's dynamic logic [12]. However, where de Groote proliferates connectives such as ⊓ for dynamic conjunction in addition to ∧ for static conjunction, we can reuse the same names such as `conj` because the `Dynamics` language inherits from the `Lambda` language. This name overloading is important because it lets us reuse the lexical meanings we have already specified for the English fragment above. For example, we do not need to specify another meaning for `who` in order to interpret it dynamically below.

We interpret the `Dynamics` language by translating it into the `States` language. That is, we define a family of `Dynamics` interpretations `D lrepr`, parameterized by a `States` interpretation `lrepr`:

```
type family Dynamic (a :: *)
type instance Dynamic (a -> b) = Dynamic a -> Dynamic b
type instance Dynamic Entity   = Entity
type instance Dynamic Bool     = State -> (State -> Bool) -> Bool
```

```
data D lrepr a = D { unD :: lrepr (Dynamic a) }

instance (States lrepr) => Lambda (D lrepr) where
  app (D f) (D x)  = D (app f x)
  lam f            = D (lam (\x -> unD (f (D x))))
  john'            = D john'
  ...              = ...
  conj (D x) (D y) = D (lam (\e -> lam (\phi -> app (app x e)
                         (lam (\e -> app (app y e) phi)))))
  exists           = D (lam (\p -> lam (\e -> lam (\phi -> app exists
                         (lam (\x -> app (app (app p x) (update x e))
                                         phi))))))

instance (States lrepr) => Dynamics (D lrepr) where
  it' = D (lam (\p -> lam (\e -> lam (\phi ->
           app (app (app p (select e)) e) phi))))
```

The type function `Dynamic`, like the type function `Tr` in §2.3, specifies that, for example, the interpretation of `farmer'` in `States` has the type `Entity -> State -> (State -> Bool) -> Bool`.

Finally, we are ready to add a pronoun `it_` to our English fragment.

```
class (Quantifier repr) => Pronoun repr where
  it_ :: repr QNP

instance Pronoun EN where
  it_ = EN "it"

instance (Dynamics lrepr) => Pronoun (Sem lrepr) where
  it_ = Sem it'
```

We derive the classic donkey sentence in this fragment.

```
sen6 = r4 (every (who (r5 own (a donkey)) farmer)) (r5 like it_)
```

We calculate the yield and simplified truth conditions of this sentence.

```
*Dynamics> sen6 :: EN S
every farmer who owns a donkey likes it
*Dynamics> sen6 :: Sem (D (P C)) S
λx1. λx2. ¬(∃ (λx3. farmer' x3 ∧ ∃ (λx4. donkey' x4 ∧ own' x4 x3
            ∧ ¬(like' (sel (x4::x3::x1)) x3))))
          ∧ x2 x1
```

By defining two more short instances (in `Dynamics.hs`, omitted here), we can also simplify formulas in the dynamic logic while treating `it'` as a primitive:

```
*Dynamics> sen6 :: Sem (P C) S
¬(∃ (λx1. farmer' x1 ∧ ∃ (λx2. donkey' x2 ∧ own' x2 x1)
                       ∧ ¬(it (λx2. like' x2 x1))))
```

Bibliography

[1] Barker, Chris, and Chung-chieh Shan. 2008. Donkey anaphora is in-scope binding. *Semantics and Pragmatics* 1(1):1–46.

[2] 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.

[3] Champollion, Lucas, Joshua Tauberer, and Maribel Romero. 2007. The Penn Lambda Calculator: Pedagogical software for natural language semantics. In *Proceedings of the workshop on grammar engineering across frameworks*, ed. Tracy Holloway King and Emily M. Bender, 106–127. Stanford, CA: Center for the Study of Language and Information.

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

[5] van Eijck, Jan, and Christina Unger. 2010. *Computational semantics with functional programming.* Cambridge: Cambridge University Press.

[6] Gallin, Daniel. 1975. *Intensional and higher-order modal logic, with applications to Montague semantics.* Mathematics studies 19, Amsterdam: North-Holland.

[7] Gamut, L. T. F. 1991. *Logic, language and meaning. Volume 2: Intensional logic and logical grammar.* Chicago: University of Chicago Press.

[8] Gansner, Emden R., and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience* 30(11):1203–1233.

[9] Geach, Peter Thomas. 1962. *Reference and generality: An examination of some medieval and modern theories.* Ithaca: Cornell University Press.

[10] de Groote, Philippe. 2002. Towards abstract categorial grammars. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 148–155. San Francisco, CA: Morgan Kaufmann.

[11] ———. 2006. Towards a Montagovian account of dynamics. In *Proceedings from Semantics and Linguistic Theory XVI.* Ithaca: Cornell University Press.

[12] ———. 2010. Dynamic logic: a type-theoretic view. Talk slides at 'Le modèle et l'algorithme', Rocquencourt.

[13] Karttunen, Lauri. 2006. The insufficiency of paper-and-pencil linguistics: The case of Finnish prosody. In *Intelligent linguistic architectures: Variations on themes by Ronald M. Kaplan*, ed. Miriam Butt, Mary Dalrymple, and Tracy Holloway King, 287–300. Stanford, CA: Center for the Study of Language and Information.

[14] 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.

[15] Montague, Richard. 1974. The proper treatment of quantification in ordinary English. In *Formal philosophy: Selected papers of Richard Montague*, ed. Richmond H. Thomason, 247–270. New Haven: Yale University Press.

[16] Moortgat, Michael. 2009. Symmetric categorial grammar. *Journal of Philosophical Logic* 38:681–710.

[17] Pereira, Fernando C. N., and Stuart M. Shieber. 1987. *Prolog and natural-language analysis*. Stanford, CA: Center for the Study of Language and Information. Reprinted 2002 by Brookline, MA: Microtome.

[18] 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.

[19] Ranta, Aarne. 2004. Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming* 14(2):145–189.

[20] Steedman, Mark J. 1996. *Surface structure and interpretation*. Cambridge: MIT Press.

[21] ———. 2000. *The syntactic process*. Cambridge: MIT Press.