COMP3161/COMP9161 Supplementary Lecture Notes

Syntax

Gabriele Keller

August 15, 2015

1 Concrete Syntax versus Abstract Syntax

The concrete syntax of a programming language is designed with the user/programmer in mind: it should be well structured and easy to read. The parser checks if a given program adheres to the concrete syntax and translates it into a suitable internal representation. The internal representation is usually quite different form the concrete syntax expression. To demonstrate why, let us go back to the arithmetic expressions example. Consider the following three expressions:

- $1 + 2 * 3$
- $(1) + (2) * (3)$
- $((1)) + (2 * 3)$

Syntactically, all three are different, but semantically, they denote exactly the same computation. Therefore they should ideally have the same internal representation. If we would have chosen a term representation of arithmetic expressions instead of an infix notation, we would not have had to worry about the ambiguity of the grammar nor about superfluous parentheses, and we could have defined the language with the following three simple rules:

$$\frac{i \in Int}{\texttt{(Num}\ i\texttt{)}\ expr}$$

$$\frac{t_1\ expr \qquad t_2\ expr}{\texttt{(Plus}\ t_1\ t_2\texttt{)}\ expr} \qquad \frac{t_1\ expr \qquad t_2\ expr}{\texttt{(Times}\ t_1\ t_2\texttt{)}\ expr}$$

and all three expressions above would correspond to the term `Plus (Num 1) (Times (Num 2) (Num 3))`. Such a term-based syntax is obviously not well suited as concrete syntax of a practical language — it would be a nightmare to write a program in this style. However, it is the appropriate format for the internal representation. A parser, therefore, has to translate expressions of the concrete syntax into terms of the abstract syntax.

2 First Order Abstract Syntax

How can we specify the translation of expressions of concrete syntax into abstract syntax terms? As we discussed previously, inference rules and judgements cannot only be used to define simple properties of single objects, but also relationships between a number of objects. We use inference rules, therefore to define a relationship

$$e_1\ SExpr \longleftrightarrow e_2\ expr$$

which holds if and only if the (concrete grammar) expression e_1 corresponds to (abstract grammar) expression e_2.

We take the inference rules of *SExpr* as basis, and add the translation for each case:

$$\frac{e_1\ \mathit{SExpr} \longleftrightarrow e_1'\ \mathit{expr} \quad e_2\ \mathit{PExpr} \longleftrightarrow e_2'\ \mathit{expr}}{e_1\ \texttt{+}\ e_2\ \mathit{SExpr} \longleftrightarrow \texttt{(Plus}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}} \qquad \frac{e\ \mathit{PExpr} \longleftrightarrow e'\ \mathit{expr}}{e\ \mathit{SExpr} \longleftrightarrow e'\ \mathit{expr}}$$

$$\frac{e_1\ \mathit{PExpr} \longleftrightarrow e_1'\ \mathit{expr} \quad e_2\ \mathit{FExpr} \longleftrightarrow e_2'\ \mathit{expr}}{e_1\ \texttt{*}\ e_2\ \mathit{PExpr} \longleftrightarrow \texttt{(Times}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}} \qquad \frac{e\ \mathit{FExpr} \longleftrightarrow e'\ \mathit{expr}}{e\ \mathit{PExpr} \longleftrightarrow e'\ \mathit{expr}}$$

$$\frac{e\ \mathit{SExpr} \longleftrightarrow e'\ \mathit{expr}}{\texttt{(}e\texttt{)}\ \mathit{FExpr} \longleftrightarrow e'\ \mathit{expr}} \qquad \frac{i \in \mathit{Int}}{i\ \mathit{FExpr} \longleftrightarrow \texttt{(Num}\ i\texttt{)}\ \mathit{expr}}$$

Previously, we used the inference rules to prove that an object had a certain property, e.g., that $1 + 3$ is indeed an *SExpr*. With relations, inference rules become more interesting. For example, we can view the rules as a description of how to construct a abstract syntax term for a given arithmetic expression, say $1 + 3 * 4$. In this case, we start of with the proof/derivation goal $1 + 2 * 3\ \mathit{SExpr} \longleftrightarrow ???$, that is, the right hand side of the relation is not fixed yet. To apply the addition rule, however, it has to have the form `Plus` ?? ??. With every rule application, we know more about the exact form of this term, until we finally end up with `Plus (Num 1) (Times (Num 2) (Num (3)))` :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1\ \mathit{FExpr} \longleftrightarrow \texttt{(Num 1)}\ \mathit{expr}}}{1\ \mathit{PExpr} \longleftrightarrow \texttt{(Num 1)}\ \mathit{expr}}}{1\ \mathit{SExpr} \longleftrightarrow \texttt{(Num 1)}\ \mathit{expr}} \quad \dfrac{\dfrac{\dfrac{}{2\ \mathit{FExpr} \longleftrightarrow \texttt{(Num 2)}\ \mathit{expr}}}{2\ \mathit{PExpr} \longleftrightarrow \texttt{(Num 2)}\ \mathit{expr}} \quad \dfrac{}{3\ \mathit{FExpr} \longleftrightarrow \texttt{(Num 3)}\ \mathit{expr}}}{2 * 3\ \mathit{PExpr} \longleftrightarrow \texttt{(Times (Num 2) (Num 3))}\ \mathit{expr}}}{1 + 2 * 3\ \mathit{SExpr} \longleftrightarrow \texttt{(Plus (Num 1) (Times (Num 2) (Num 3)))}\ \mathit{expr}}$$

The process of finding, for a given s, s *SExpr*, a term t, t *expr*, with s *SExpr* $\longleftrightarrow$ t *expr* is called *parsing*. A parser has to be complete, in the sense that for every s *SExpr* it should find the corresponding abstract syntax term. Furthermore, it has to be unambiguous, and return for every s *SExpr* a unique t. The inverse process is called *unparsing*. Since each of the parsing rules given above directly corresponds to a production rule of *SExpr*, it is trivial to show the completeness using rule induction.

We can interpret the inference rules given above also differently, and instead of deriving a term t *expr* for a given s, we can start with an abstract syntax term and derive a matching arithmetic expression. This process is called *unparsing*. Unparsing is, in our example and in general, ambiguous. It is also not necessarily complete, although it is for the arithmetic expression we defined. Going one step further and converting the concrete syntax token sequence back into a string is called *pretty printing*. Pretty printing is useful, for example, to view intermediate code in a compiler, or in tools which re-format a program. Pretty printing is even more ambiguous than unparsing, since we are free to add spaces and new lines in many places. A pretty printer aims at choosing the most readable representation, which is of course a very subjective measure.

3 Higher Order Abstract Syntax

Let us extend our simple language of arithmetic expressions by introducing variables and a let-construct to bind variables to values:

```
let x = 3                let x = 3
in x+1                   in let y = x+1
                            in x+y
```

We need to extend the set of rules which define the concrete and abstract syntax accordingly.

Concrete Syntax: A let-expression acts like a parenthesised expression, and behave therefore like a *FExpr*. As with numbers, where we use *int* to denote an integer without further specifying it, we use *id* to represent an identifier:

$$\frac{e_1\ SExpr \quad e_2\ SExpr}{\texttt{let}\ id = e_1\ \texttt{in}\ e_2\ FExpr}$$

Abstract Syntax: Variables are represented by terms, similar to numbers, with the exception that the argument of the term is a string *id* which contains the name of the variable. A let-expression is represented by a term which requires three arguments: an identifier (bound variable), a term (the term the variable is bound to), and the body of the let-expression, i.e., the term in which the variable is defined. Since the first argument has to be an identifier, we can drop the `var` operator in the argument position:

$$\frac{}{\texttt{(Var}\ id\texttt{)}\ expr} \qquad \frac{\texttt{(Var}\ id\texttt{)}\ expr \quad t_1\ expr \quad t_2\ expr}{\texttt{(Let}\ id\ t_1\ t_2\texttt{)}\ expr}$$

3.1 Scope of a Variable

Given an expression `let x =` t_1 `in` t_2, the variable `x` is bound in t_2, but not in t_1. The term t_2 is called the *scope* of `x`, and `x` is bound to the value of t_1 everywhere in t_2. In particular, in the following situation where the same variable is bound twice, the outer binding is *shadowed* by the inner binding, and the value of the expression is 10.

```
let x = 3
in let x = 5
   in x + x
```

Note that the scope of a variable in a let-binding is defined differently in Haskell: `x` is bound in t_1 *and* t_2. As a consequence, the compiler accepts an expression

```
let
   x = x+1
in x
```

even though the expression cannot be evaluated and leads to a run-time exception.

3.2 Representation of Variables

In the first order abstract syntax definition of arithmetic expressions, variables are treated just like numbers and represented by terms, although they play a special role. Higher-order abstract syntax addresses this shortcoming, and provides variables and variable bindings as part of the meta-language: first order terms can either

1. be a constant (e.g, ints, strings), or
2. have the form $(Op\ t_1\ \ldots\ t_n)$, where t_1 to t_n are terms,
   - `(Num 4)`
   - `(Plus (Num 2) (Num 1))`

In addition, a higher-order term can

3. be a variable
4. have the form $x.t'$, meaning the variable x is bound in term t'
   - `x. (Plus x (Num 1))`

```
- x.y. (Plus x y)
```

An term of the form $x.t$ is called an *abstraction*. It is a term whose value depends[1] on the value of the variable x. In this respect, it is similar to a function body.

An abstraction $x.t$ is said to *bind* all occurrence of x in t. All variables of a term which are not bound at the position they occur are called *free variables* of that term. We denote the set of free variables of a term by $FV(t)$. It is inductively defined follows:

$$\begin{array}{ll} FV\,(int) & = \{\} \\ FV\,(x) & = \{x\} \\ FV\,(o\,(t_1,\,\ldots,\,t_n)) & = FV(t_1) \cup \ldots\, FV(t_n) \\ FV\,(x.t) & = FV(t) \setminus \{x\} \end{array}$$

For example, `x` is in $FV($ `plus` (`x`, `let` (5, `x`.`x`))), which corresponds to the concrete syntax expression

```
x + (let x = 5 in x)
```

since

$$\begin{array}{l} \quad FV(\ (\texttt{Plus x (Let (Num 5) (x.x))})) \\ = FV(\texttt{x}) \cup FV(\texttt{Let (Num 5) (x.x)})) \\ = \{\texttt{x}\} \cup FV(5) \cup FV(\texttt{x.x}) \\ = \{\texttt{x}\} \cup (FV(\texttt{x}) \setminus \{\texttt{x}\}) \\ = \{\texttt{x}\} \cup (\{\texttt{x}\} \setminus \{\texttt{x}\}) \\ = \{\texttt{x}\} \cup \{\} \end{array}$$

If we want to use higher order syntax, we have to change the rules for variables and let-bindings in the definition of the abstract syntax:

$$\frac{}{id\ expr} \qquad \frac{t_1\ expr \quad t_2\ expr}{\texttt{(Let}\ t_1\ \texttt{(}id.t_2\texttt{))}\ expr}$$

and adapt the the translation accordingly:

$$\frac{}{id\ FExpr\ \longleftrightarrow\ \texttt{(}id\ expr\texttt{)}} \qquad \frac{e_1\ SExpr \longleftrightarrow t_1\ expr \quad e_2\ SExpr \longleftrightarrow t_2\ expr}{let\ id = e_1\ in\ e_2\ \ FExpr \longleftrightarrow \texttt{(Let}\ t_1\ \texttt{(}id.t_2\texttt{))}\ expr}$$

Now, the operator `let` accepts only two arguments, one being the right hand side, the second the abstraction of the body of the body.

3.3 Substitution and α-equivalence

Consider, for example, the following two expressions:

```
let              let
   x = 3           y = 3
in               in
   x+1             y+1
```

They express exactly the same computation and only differ in the choice of the variable names. They are represented by the term `Let (Num 3) (x.  Plus x (Num 1))` and `Let (Num 3) (y. Plus y (Num 1))` respectively. If two terms, as in the above example, can be made identical by renaming the variables, they are called α-equivalent, written $\equiv_\alpha$. As the name suggests, $\equiv_\alpha$ is an equivalence relation. This means $\equiv_\alpha$ is

1. reflexive: for all terms t, $t \equiv_\alpha t$

[1] More precisely, may depend, since it is possible that x does not actually occur in t, as in $x.1$

2. symmetric: for all terms t_1, t_2, if $t_1 \equiv_\alpha t_2$ then $t_2 \equiv_\alpha t_1$

3. transitive: for all terms t_1, t_2, and t_3: if $t_1 \equiv_\alpha t_2$ and $t_2 \equiv_\alpha t_3$ then $t_1 \equiv_\alpha t_3$

If we want to determine the value of a let-expression, at some point, we have to replace the variable in the body with the value the variable is bound to. This process of replacing a variable with a value, or in general, an arbitrary term, is called *substitution*. We use the notation:

$$t[x := t']$$

to describe a term t where every free occurrence of x has been replaced by t'. We can rename the variables in a term now by replacing the variable at its binding occurrence, and substituting it wherever it occurs freely in the term:

$$x.t \equiv_\alpha y.t[x := y], \quad \text{if } y \notin FV(t)$$

We have to be careful about the choice of y, though. If we try to rename `x` to `y` in the term `x. x + y` we do not want to end up with the term `y. y + y`, since the `y` in the original term is now captured, and the new term is not α-equivalent to the original term anymore. Therefore, we require that the new variable does not occur freely anywhere in the original term.

Let us now give the exact definition of substitution, first for variables:

$$\begin{array}{lcl}
x[x := y] & = & y \\
z[x := y] & = & z, \ \textit{if } x \neq z \\
(Op\ t_1 \ldots t_n)[x := y] & = & (Op\ \ t_1[x := y] \ \ldots \ t_n[x := y]) \\
x.t[x := y] & = & x.t \\
z.t[x := y] & = & z.t[x := y] \ \textit{if } x \neq z,\ y \neq z, \\
y.t[x := y] & = & \textit{undefined if } x \neq y
\end{array}$$

To avoid the problem of capturing, we require that the variable which is introduced does not occur anywhere in the binding position of a term. Similarly, if we substitute terms, we require that none of the free variables in the new term occurs at a binding position in the original term:

$$\begin{array}{lcl}
x[x := u] & = & u \\
z[x := u] & = & z, \ \textit{if } x \neq z \\
(Op\ t_1 \ldots t_n)[x := u] & = & (Op\ t_1[u := x] \ldots \ t_n[u := x]) \\
x.t[x := u] & = & x.t \\
z.t[x := u] & = & z.t[u := x] \ \textit{if } x \neq z,\ z \notin FV(u), \\
y.t[x := u] & = & \textit{undefined if } y \in FV(u)
\end{array}$$

In practice, it does not matter that substitution is only partially defined, because we can always rename the variable such that clashes do not occur. Many compilers will actually rename all the variables defined by the user and map them to unique names to simplify the further compilation steps. We silently assume from now on that the programs we are dealing with have unique names.

4 The Untyped Lamda Calculus

In the 1930, the US mathematician Alonso Church, in co-operation with his students Stephen Kleene and Barkely Rosser, developed the untyped λ-calculus. He was looking for a foundation for logic that was simpler than Zermelo's set theory, so he would be prove that a solution to Hilbert's *Entscheidungsproblem* (decision problem) is not possible. The λ-calculus is extremely simple, but surprisingly powerful enough to express the same set of calculations as the Turing machine (Alan Turing was also a doctoral student of Church).[2]

[2]See: *History of Lambda-calculus and Combinatory Logic* by Felice Cardone, J. Roger Hindley 2006.

We can characterise λ-terms with the following set of only three inference rules:

$$\frac{id \in Ident}{id \ \ \lambda{-}term}(Var) \qquad \frac{t \ \ \lambda{-}term}{\lambda id.t \ \ \lambda{-}term}(Abstraction)$$

$$\frac{t \ \ \lambda{-}term \quad s \ \ \lambda{-}term}{(t\ s) \ \ \lambda{-}term}(Application)$$

To manipulate λ-terms, we have three rules:

1. α-conversion: if $t \equiv_\alpha s$, then the two terms are equivalent in the λ-calculus.
2. β-reduction: a term of the form $(\lambda x.t)s$ can be reduced to $t[x := s]$.
3. η-conversion: the λ-term $\lambda x.(fx)$ is equivalent to f if x is not free in f.

The first rule, α-conversion, simply states that the names of bound variables are not important. The important part of the λ-calculus is β-reduction, as it tells us how function application can be evaluated. There is nothing surprising here: we simply replace every occurence of the variable the function abstracts over with the term passed as argument. The last rule, η-conversion, states that if we have a term f and apply it to a variable, and then abstract over that same variable, we end up with the term we started.

In the pure λ-calculus, we don't even have numbers or other primitive data types we usually rely on. However, it turns out, we can encode everything we need using pure λ-terms. Of course, it would be extremely tedious to use, but keep in mind that the pure λ-calculus was invented to facilitate proofs about calculations, so simplicity was much more important that usability. This was one of the reasons why the λ-calculus received very little attention by mathematicans after the 1930's, as it was viewed as irrelevant for practical purposes. It was people like Konrad Zuse, John McCarthy, Peter Landin and Corrado Böhm who re-discovered the power of this calculus for computer science.

We will come back to the pure λ-calculus later, but for now, we include numbers and arithmetic expression. Let's start with a simple function, which takes an argment and increments it by one. In the λ-calculus with arithmetic expres)sion, we express this as

$$\lambda\, x.\, (x\ +1)$$

and now we can apply it to an argument, let say the number 5:

$$\begin{array}{ll}(\lambda\, x.\, (x\ +1))\ 5 = \\ (x+1)[x := 5] & = \\ 5\ +\ 1 & = \\ 6\end{array}$$

This is really very similar to what we would do with a regular mathematical function, where we would write for the definition something like:

$$inc\,(x)\ =\ x\ +\ 1$$

and for the application $inc(5)$, which would cause x in the body to be replaced with 5. The difference is that in the λ-calculus, a function doesn't have a name. λ-abstractions are therefore also called *anonymous functions*.

We can also observe the effect of applying the increment function to a variable and abstracting over the variable:

$$\begin{array}{l}\lambda\, y.\, ((\lambda\, x.\, (x\ +1))\ y) = \\ \lambda\, y.\, (\ (x\ +1)[x := y]) = \\ \lambda\, y.\, (y\ +\ 1)\end{array}$$

That is, we end up with a λ-term which is equivalent to our original function, which is just what the η-reduction rule says.

We can express the function which takes multiple arguments by nesting λ-abstractions. For example, the function which calculates the average of two numbers as the term:

$$\lambda x.\lambda y.((x + y)/2)$$

If we apply this function to the number 3, we can apply β-reduction:

$$(\lambda\, x.\, \lambda\, y.\, ((x\ +\ y)\ /\ 2))\ 3 =$$
$$\lambda\, y.\, ((3\ +\ y)\ /\ 2)$$

That is, the result of the application is another function, which we can then apply to a second argument.

Let us get back to the pure lambda calculus. Since there are not built-in data types, we have to use terms to represent objects like numbers or boolean values.

When trying to figure out how to represent certain data types in the λ-calculus, we have to find a representation which is well suited to the operations we want to apply to them. In the case of boolean values, we basically want to be able to distinguish True from False. So we should be able to define a conditional and two alternatives, and switches to the first alternative if applied to True, to the second if applied to False.

$$\begin{aligned} \mathit{True} &:= \lambda\, x.\, \lambda\, y.\, x \\ \mathit{False} &:= \lambda\, x.\, \lambda\, y.\, y \\ \mathit{If} &:= \lambda\, c.\, \lambda\, t.\, \lambda\, e.\, ((c\ t)\ e) \end{aligned}$$

With these definitions, we have

$$\begin{aligned} (\mathit{If}\ \mathit{True}) &= (\lambda\, c.\, \lambda\, t.\, \lambda\, e.\, ((c\ t)\ e))\ (\lambda\, x.\, \lambda\, y.\, x) \\ &= (\lambda\, t.\, \lambda\, e.(\ (\lambda\, x.\, \lambda\, y.\, x)\ t\ e) \\ &= (\lambda\, t.\, \lambda\, e.(\ (\lambda\, y.\, t)\ e) \\ &= (\lambda\, t.\, \lambda\, e.\, t) \end{aligned}$$

and

$$\begin{aligned} (\mathit{If}\ \mathit{True}) &= (\lambda\, c.\, \lambda\, t.\, \lambda\, e.\, ((c\ t)\ e))\ (\lambda\, x.\, \lambda\, y.\, y) \\ &= (\lambda\, t.\, \lambda\, e.(\ (\lambda\, x.\, \lambda\, y.\, y)\ t\ e)) \\ &= (\lambda\, t.\, \lambda\, e.(\ (\lambda\, y.\, y)\ e)) \\ &= (\lambda\, t.\, \lambda\, e.\, e) \end{aligned}$$

where the terms t and e represent the *then* and *else* branch, respectively. We can build *Not*, *And*, and *Or* using these terms:

$$\begin{aligned} \mathit{Not}\ a &= \mathit{If}\ a\ \mathit{False}\ \mathit{True} \\ &= ((a\ (\lambda\, x.\, \lambda\, y.\, y))\ (\lambda\, x.\, \lambda\, y.\, x)) \end{aligned}$$

Therefore, *Not* can be expressed as the λ-term

$$\mathit{Not} = \lambda\, a.\, ((a\ (\lambda\, x.\, \lambda\, y.\, y))\ (\lambda\, x.\, \lambda\, y.\, x))$$

And similarly for *And* and *Or*:

$$\begin{aligned} \mathit{And}\ a\ b &= \mathit{If}\ a\ b\ \mathit{False} \\ &= ((a\ b)\ (\lambda\, x.\, \lambda\, y.\, y)) \\ \mathit{And} &= \lambda\, a.\, \lambda\, b.\, ((a\ b)\ (\lambda\, x.\, \lambda\, y.\, y)) \\ \mathit{Or}\ a\ b &= \mathit{If}\ a\ \mathit{True}\ b \\ &= ((a\ (\lambda\, x.\, \lambda\, y.\, x))\ b) \end{aligned}$$

$$Or \quad = \lambda\, a.\, \lambda\, b.\, ((a\ (\lambda\, x.\, \lambda\, y.\, x))\ b)$$

Encoding natural numbers is more complicated. One way is to encode numbers as λ-terms which apply a function argument f n times to the second argument, the variable x:

$$\begin{array}{ll} 0 & := \lambda\, f.\, \lambda\, x.\, x \\ 1 & := \lambda\, f.\, \lambda\, x.\, (f\ x) \\ 2 & := \lambda\, f.\, \lambda\, x.\, (f\ (f\ x)) \\ \ldots & \\ n & := \lambda\, f.\, \lambda\, x.\, (f\ (f\ \ldots\ (f\ x)) \end{array}$$

and the successor function which increments a number by one as:

$$\lambda\, n.\, \lambda\, f.\, \lambda\, x.\, (f\ ((n\ f)\ x))$$

The term $((nf)x)$ (or just nfx, because application is left associative, but for now we keep the parenthesis) can be read as *apply function f n times to x*. As there is an additional f in the term, f is applied $n+1$ times overall.

Applying the successor function to the term representing the number one then results (as we would hope) in the term representing the number two (the rules applied in each step are in curly braces, and the part of the term it affects is set in red):

$$\begin{array}{lll} (\lambda\, n.\lambda\, f.\, \lambda\, x.\, (f\ ((n\ f)\ x)))\ (\lambda\, f.\, \lambda\, x.\, (f\ x)) & = & \{\eta - \textit{conversion}\} \\ (\lambda\, n.\lambda\, f.\, \lambda\, x.\, (f\ ((n\ f)\ x)))\ (\lambda\, f.\, f)) & = & \{\alpha - \textit{renaming}\} \\ (\lambda\, n.\lambda\, f.\, \lambda\, x.\, (f\ (\ n\ f)\ x)\ (\lambda\, g.\ g)) & = & \{\beta - \textit{reduction}\} \\ \quad \lambda\, f.\, \lambda\, x.\, (f\ ((\lambda\, g.\ \ g)\ f)\ x\ ) & = & \{\beta - \textit{reduction}\} \\ \quad \lambda\, f.\, \lambda\, x.\, (f\ (f\ x)) & & \end{array}$$

As first step, we simplify the term by applying η-conversion - we could also drop this step. Then, we rename the variables of the argument, just to make it clear that the f in the argument refers to differen variable than the f in the outer binding. Again, this step isn't necessary. Then, we apply reduction twice and end up with the λ-term representing the number two.