CPSC 311: "Expanding" rules, strings, type safety
(DRAFT)
("lec-safety")

Joshua Dunfield
University of British Columbia

October 30, 2016

2016 note: We already covered some of the material on strings, so feel free to skip it, if you remember it.

1 Review

Defining programming languages:

- Defining syntax: BNF
- Defining semantics: rules

1.1 BNFs

Here's a BNF:

Characters	⟨ch⟩	::=	`a` | `b` | `c` | ⋯
Strings	⟨S⟩	::=	`"`⟨ch⟩ ... `"` ⟨ch⟩ ... means zero or more repetitions of ⟨ch⟩
Cats	⟨C⟩	::=	⟨S⟩
			| `{+` ⟨C⟩ ⟨C⟩`}`

Read "::=" as "can have the form". Other readings you may come across are "expands to" or "rewrites to", which are reasonable in some contexts, but I feel they're misleading in the context of programming languages. We are given an input string (a program) and want to parse it; if there is "rewriting" happening (and I'm not sure there is), it should be going from right to left: the parser sees `b` and realizes it is a character ⟨ch⟩.

Symbols on the **left** of the "::=", like ⟨ch⟩, ⟨S⟩, ⟨C⟩, are called *nonterminals*. On the right hand side, alternatives separated by "|" are called *productions*.

In a BNF, when a nonterminal appears twice, it can (and usually does) represent a different string. For example,

`{+ "ab" "cd"}`

is a ⟨C⟩ because `"ab"` and `"cd"` are each ⟨C⟩s (because they are each an ⟨S⟩ (because ...)).

By itself, a BNF only tells you what input strings (programs) are syntactically valid. You might be able to *guess* that I want to define a simple language of string concatenation, where you can "run" `{+ "ab" "cd"}` and get `"abcd"`, but the BNF doesn't say that.

Remark. Unlike "formal semantics" (rules), "formal syntax" (BNF) is used for most "real" programming languages, so it's important to understand it. You also have to be prepared for variations in notation (which is why I try to be careful to always tell you what "..." means).

1.2 Abstract syntax

Lisp was supposed to have a "real" syntax, which was never finished. But this piece of vaporware led to something useful: abstract syntax.

Unlike most "real" syntaxes, abstract syntax is not ambiguous; it doesn't need to resolve the ambiguity of `a + b * c`, because everything is in brackets/parentheses/braces.

The **define-type** feature of Racket/PLAI is ideally suited to defining abstract syntax: everything is in parentheses, because everything in Racket is in parentheses.

```
(define-type CatExpr
   [Str (s string?)]
   [Cat (left CatExpr?) (right CatExpr?)]
)
```

In the concrete syntax BNF, I could just write ⟨S⟩ by itself as a production of ⟨C⟩, In abstract syntax, each alternative has to begin with a variant name (like `Str`).

Here, I have written `Cat` instead of `+`, but this is still only syntax. Using the name `Cat` strongly suggests that this is meant to be string concatenation, but so far, that's only a name.

1.3 Rules

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_m}{\text{conclusion}}\ \text{rule name}$$

There might be no premises. We've seen that with rules like

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num}$$

The conclusion is always some *judgment*, like $e \Downarrow v$ or $\Gamma \vdash e : A$. The conclusion and premises usually have *meta-variables*, which we can replace with instances.

To apply a rule, you replace all its meta-variables. Eval-num has one meta-variable, n. If I instantiate it with 4, I get a *derivation* of $(\mathsf{Num}\ 4) \Downarrow (\mathsf{Num}\ 4)$.

$$\frac{}{(\mathsf{Num}\ 4) \Downarrow (\mathsf{Num}\ 4)}\ \text{Eval-num}$$

In a rule, unlike a BNF, repeated occurrences of the same meta-variable refer to the same thing. So you can't replace the first n with 4, and the second n with 5. (This is a confusing difference, but it's now too standard to change.)

2 A useful way to read typing rules

The next page (CURRENTLY HANDWRITTEN; WILL BE SCANNED FOLLOWING LECTURE) illustrates how to "expand" typing rules so you can implement them more directly. When you make a recursive call to derive a premise, you can't constrain in advance what result you get. If the rule requires something about a type, you have to check that *after* you use `let` or `let*` to bind that type in your Racket code. If you write the rule a little differently, you get something that's closer to the code you need to write.

3 Strings, continued

3.1 BNFs

Strings ⟨S⟩ ::= *whatever a Racket string is*

Cats ⟨C⟩ ::= ⟨S⟩
| {+ ⟨C⟩ ⟨C⟩}

Instead of studying the above (very small!) language, we'll add its features to one of our versions of Fun, so that we can see, in a slightly more realistic language, how to define evaluation and typing for these features.

Expressions ⟨E⟩ ::= ...*whatever is in* `typed-lam.rkt`
| ⟨S⟩
| ~~{Cat ⟨C⟩ ⟨C⟩}~~

| {Cat ⟨E⟩ ⟨E⟩}
| {Nth ⟨E⟩ ⟨E⟩}

I've crossed out one of the productions, because I want strings to be interoperable with Fun expressions, so that we can `Cat` two identifiers, or `Cat` the result of applying two functions, etc.

What is the semantics of `Nth`? It will return the 1-character string at a given index into the string, which will illustrate some language design alternatives.

3.2 Abstract syntax

```
(define-type E
   .
   .
   .
   [Str (s string?)]
   [Cat (str1 E?) (str2 E?)]
   [Nth (str E?) (index E?)]
)
```

3.3 Evaluation rules

The following is from 2015. Keeping it to minimize changes on my end.

$\boxed{e \Downarrow v}$ Expression e evaluates to value v

$$\frac{}{(\mathsf{Str}\ s) \Downarrow (\mathsf{Str}\ s)}\ \text{Eval-str} \qquad \frac{e1 \Downarrow (\mathsf{Str}\ s1) \qquad e2 \Downarrow (\mathsf{Str}\ s2)}{(\mathsf{Cat}\ e1\ e2) \Downarrow (\mathsf{Str}\ s1\ s2)}\ \text{Eval-cat}$$

$$\frac{eS \Downarrow (\mathsf{Str}\ s1) \qquad eIdx \Downarrow (\mathsf{Num}\ n) \qquad n \in \mathbb{N} \qquad n < \mathit{len}(s1)}{(\mathsf{Nth}\ eS\ eIdx) \Downarrow (\mathsf{Str}\ s1_n)}\ \text{Eval-nth}$$

The difference between the string $s1$ and $(\mathsf{Str}\ s1)$ is that $s1$ is a sequence of characters, for which we can define (or assume) various mathematical functions, while $(\mathsf{Str}\ s1)$ is abstract syntax.

In writing the rule Eval-nth, we assumed that $\mathbb{N}$ are the natural numbers (and that they start at 0, which is the usual convention in computer science but not necessarily other fields), that $\mathit{len}(s)$ is a (mathematical) function that returns the number of characters in s, and that a subscript like

$$s1_n$$

denotes the nth character of the string $s1$.

We arrived at the third and fourth premises of Eval-nth by something like the following process.

- First, we voted overwhelmingly (apparently influenced by the federal election) that strings should be indexed from 0 rather than 1.
- Second, we decided (less democratically) to require n to be an integer, rather than taking the floor $\lfloor n \rfloor$. (Because Fun's numbers are the same as Racket's numbers, a Num in Fun can be floating-point, rational, or even complex.)
- Third, we decided that n should be required to be in the range $0 \leq n < \mathit{len}(s1)$, rejecting a suggestion that we define it "circularly" by taking n mod $\mathit{len}(s1)$.

  (Another possible suggestion: "pin" n to the range, by returning the 0th character when $n < 0$, and the last character when $n \geq \mathit{len}(s1)$. Both this suggestion and the "circular" suggestion don't entirely succeed in their questionable goal of always returning *something*: what should evaluation do if the string's length is zero?)

3.4 Errors

What if eIdx evaluates to something that isn't a Num?
What if eS evaluates to something that isn't a Str?
Both of these can be easily prevented using types.

What if eIdx does evaluate to $(\mathsf{Num}\ n)$, but n is not an integer? This is feasible to prevent using types, say, by removing the type num and putting in int and float types instead, but we won't pursue that now.

What if n falls outside the string? This is much more difficult to prevent with a type system, but it *is* possible.

3.5 "Going wrong"

A slogan of types advocates is: "Well-typed programs don't go wrong."

This slogan only makes sense if we specifically define what "wrong" means. Then, there are *particular kinds of errors* that are prevented by the typing rules.

The slogan comes from a paper by Robin Milner (the main inventor of Standard ML), who—in his defence—*did* precisely define what he thought "wrong" meant: essentially, it prevented "agreement errors" like trying to apply a number (that is, to call a number as if it were a function), or passing a list to a function that expects an integer, and so on. As you all know by now, such errors happen fairly often, so there's a strong argument for preventing them.

4 Typing rules

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

$$\frac{}{\Gamma \vdash (\mathsf{Str}\ s) : \mathsf{string}}\ \text{Type-str} \qquad \frac{\Gamma \vdash e1 : \mathsf{string} \qquad \Gamma \vdash e2 : \mathsf{string}}{\Gamma \vdash (\mathsf{Cat}\ e1\ e2) : \mathsf{string}}\ \text{Type-cat}$$

$$\frac{\Gamma \vdash eS : \mathsf{string} \qquad \Gamma \vdash eIdx : \mathsf{num}}{\Gamma \vdash (\mathsf{Nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

"Expanding" the above rules as discussed above gives:

$$\frac{}{\Gamma \vdash (\mathsf{Str}\ s) : \mathsf{string}}\ \text{Type-str} \quad \frac{\Gamma \vdash e1 : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash e2 : A2 \qquad A2 = \mathsf{string}}{\Gamma \vdash (\mathsf{Cat}\ e1\ e2) : \mathsf{string}}\ \text{Type-cat}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \mathsf{num}}{\Gamma \vdash (\mathsf{Nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

5 Type safety

The standard way of showing that a type system really prevents (certain kinds of) errors is to prove *type safety*.

Type safety is a result about the *relationship* between the static semantics and the dynamic semantics. Thus, changing either set of rules can break type safety.

Type safety is more usefully stated for a small-step semantics ($e1 \longrightarrow e2$) rather than for a big-step evaluation semantics, but you're more familiar with the big-step semantics, so we'll start with that.

Type safety can be divided into two parts: *preservation* and *progress*.

5.1 Preservation

Preservation says, roughly, that evaluation "preserves types": if you run a program of type bool, and it evaluates to a value, that value will also have type bool.

For the **big-step semantics** $e \Downarrow v$, preservation can be stated as:

> If $\emptyset \vdash e : A$
> and $e \Downarrow v$
> then $\emptyset \vdash v : A$.

Preservation is a limited statement that can best be characterized as: "If you got a value, *then* it is a reasonable value." For example, preservation tells you that if the typing rules say that the expression (App (Lam x num x) (Num 3)) has type num, you won't somehow get a bool instead.

The above preservation statement is actually even more limited than it might appear: if evaluation loops infinitely due to a rec, the above preservation result doesn't help us, because we can only apply it if $e \Downarrow v$ holds.

Nonetheless, preservation should still tell us that some, maybe all, of the errors that `interp` can raise will never happen. (You should be skeptical of even this claim! What could go wrong?)

For the **small-step semantics** $e1 \longrightarrow e2$, preservation can be stated as:

> If $\emptyset \vdash e1 : A$
> and $e1 \longrightarrow e2$
> then $\emptyset \vdash e2 : A$.

5.2 Progress

For the small-step semantics, **progress** can be stated as:

> If $\emptyset \vdash e1 : A$ then **either**
>
> - $e1$ is a value, or
> - $e1 \longrightarrow e2$, or
> - $e1$ raises an error that is not one of the errors we claim is caught by the type system.

For a big-step semantics $e \Downarrow v$, there is (for most languages) no directly corresponding progress result. The following doesn't hold for Typed Fun, for example, because of rec.

If $\emptyset \vdash e : A$
then $e \Downarrow v$.

A key benefit of small-step semantics is that preservation and progress tell us that running a program, even one that loops infinitely, won't launch the missiles along the way.