CPSC 311: Definition of Programming Languages: Assignment 3: Type Checking

Joshua Dunfield
University of British Columbia

October 14, 2016

1 Logistics

You may work in teams of 2 on this assignment. (You *can* work individually; if there are an odd number of students doing the assignment, someone will have to. But we recommend that you collaborate, mostly for your benefit, but—I'll be honest—also for ours: there are many students taking 311, but not very many TAs, and working individually means one more assignment to mark. **Repeating this reminder for a3. Also, you can submit as a team even if you complete the assignment separately, and meet only to decide on a combined solution. We don't particularly recommend that, but it's still one less assignment to mark, and you'll almost certainly learn something from the other person's solution.**)

You must include a README.txt file based on this template:

`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/assignments/support/README.txt`

For your final submission, be sure you have replaced **all** of the "`TODO`"s in `README.txt`.

handin has not yet been set up for this assignment. When handin has been set up, we will make an announcement on Piazza.

Download

`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/assignments/a3.rkt`

1.1 Important! (new in a3)

If your code is rejected by "Check Syntax", or the `handin` script can't run your code, you will receive a mark of 0 **for the entire assignment**.

So, if you get stuck on one problem, **comment out the code that doesn't work**, and try to explain in a comment what you were trying to do. **Make sure that your final handin, at minimum, prints test results**—even if all the tests fail!

This assignment is due at 22:00 (10:00pm) on Friday, 2016/10/21.

1.2 Handin

You must turn in the assignment using the **command-line version** of the handin program. For handin, this assignment is called `a3`. Submit two files: `a3.rkt` and `README.txt`, with contents as described above. (If you choose to try one or more bonus problems, also include a file `bonus.rkt`). Include your name(s) at the top of each file.

If you are working in a team, submit only one set of files. If you have both run handin for a3 already, have one person overwrite their handin with a directory containing only a file called

`please-mark-aaaaa`

where aaaaa is the CS ugrad username of your partner.

Just a reminder, late assignments are not accepted (except for the “grace period” of a few minutes, which you shouldn’t rely on), and (basically) no excuses will be entertained. So, handin your assignments early and often!

Avoid using DrRacket comment boxes, because handin is still afraid of them. Comments using “;” and “#| ... |#” are fine.

2 Overview

In this assignment, you’ll take an implementation of the Fun language with a (partially written) type checker, and extend the type checker—and some other functions—to the full language.

3 Syntax

Familiarize yourself with the syntax in `a3.rkt`.

For Typed Fun, we added types to the concrete syntax:

```
⟨Type⟩ ::= num
         | bool
         | {tree ⟨Type⟩}
         | {* ⟨Type⟩ ⟨Type⟩}
         | {-> ⟨Type⟩ ⟨Type⟩ ...}
```

In the last production, the “...” denotes one or more occurrences of a type, *in addition to* the ⟨Type⟩ immediately following the “->”. Thus, `{-> num num}` and `{-> num num bool}` are valid ⟨Type⟩s, but `{-> num}` is not.

An occurrence of `->` followed by three or more types is syntactic sugar for the right-associative unfolding:

```
{-> num num bool}
```

is syntactic sugar for

```
{-> num {-> num bool}}
```

You can experiment with the function `build->` in a3.rkt.

We also added several productions to the concrete syntax of expressions; see `a3.rkt`.

- `Lam` has a new second argument: the domain type of the function.
- `Rec` has a new second argument: the type of the recursive body.
- `{Leaf ⟨Type⟩}` represents a leaf node of a tree whose keys are of the given type. We have to write the type for reasons similar to `Lam` and `Rec`.
- `{Branch ⟨E⟩ ⟨E⟩ ⟨E⟩}` represents a branch node of a key (the first ⟨E⟩), a left child (the second ⟨E⟩), and a right child (the third ⟨E⟩).
- `{Tree-case ⟨E⟩ {Leaf => ⟨E⟩} {Branch ⟨symbol⟩ ⟨symbol⟩ ⟨symbol⟩ => ⟨E⟩}}` is a kind of **type-case** for trees. If the first ⟨E⟩ (called the “scrutinee”) evaluates to a `Leaf`, the ⟨E⟩ in the `Leaf` part (the second ⟨E⟩) is evaluated. Otherwise, the ⟨E⟩ in the `Branch` part is evaluated, after substituting the key, left child, and right child for the given ⟨symbol⟩s.

§3 Syntax

In the abstract syntax of Typed Fun, note the following:

- The Lam variant has a new second argument: the domain of the function.
- The Rec variant has a new second argument: the type of the recursive body.
- Three new variants have been added for trees: Leaf, Branch, and Tree-case.
- The abstract syntax does not have Let*: now, the parser treats `Let*` as syntactic sugar, turning it into a sequence of Lets.

4 Evaluation semantics

Much of this section is the same as in a2, except that `Let*` is syntactic sugar and therefore omitted.

4.1 Values

As in Fun, $(\mathsf{Num}\ n)$ and $(\mathsf{Lam}\ x\ A\ e)$ and (Btrue) and (Bfalse) and pairs of values are values.

In addition, $(\mathsf{Leaf}\ A)$ is a value, and $(\mathsf{Branch}\ e1\ e2\ e3)$ is a value if $e1$, $e2$, and $e3$ are values.

4.2 Evaluation rules

4.2.1 Rules reused from Fun

These rules are the same as in Fun, except for the extra types in Lam and Rec, which evaluation ignores.

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let}$$

$$\frac{}{(\mathsf{Lam}\ x\ A\ e1) \Downarrow (\mathsf{Lam}\ x\ A\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\mathsf{Lam}\ x\ A\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

$$\frac{e1 \Downarrow v1 \qquad e2 \Downarrow v2 \qquad v1\ \mathsf{op}\ v2\ =\ v}{(\mathsf{Binop}\ \mathsf{op}\ e1\ e2) \Downarrow v}\ \text{Eval-binop}$$

$$\frac{e1 \Downarrow v1 \qquad e2 \Downarrow v2}{(\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2)}\ \text{Eval-pair} \qquad \frac{ePair \Downarrow (\mathsf{Pair}\ v1\ v2) \qquad [v2/x2][v1/x1]eB \Downarrow v}{(\mathsf{Pair\text{-}case}\ \ ePair\ \ x1\ \ x2\ \ eB) \Downarrow v}\ \text{Eval-pair-case}$$

$$\frac{}{(\mathsf{Btrue}) \Downarrow (\mathsf{Btrue})}\ \text{Eval-btrue} \qquad \frac{}{(\mathsf{Bfalse}) \Downarrow (\mathsf{Bfalse})}\ \text{Eval-Bfalse}$$

$$\frac{e \Downarrow (\mathsf{Btrue}) \qquad eThen \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-true} \qquad \frac{e \Downarrow (\mathsf{Bfalse}) \qquad eElse \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-false}$$

$$\frac{\big[(\mathsf{Rec}\ u\ B\ e)/u\big]e \Downarrow v}{(\mathsf{Rec}\ u\ B\ e) \Downarrow v}\ \text{Eval-rec}$$

There are new rules on the next page!

4.2.2 New evaluation rules

$$\frac{}{(\mathsf{Leaf}\ A) \Downarrow (\mathsf{Leaf}\ A)}\ \text{Eval-Leaf} \qquad \frac{eKey \Downarrow vKey \qquad eL \Downarrow vL \qquad eR \Downarrow vR}{(\mathsf{Branch}\ eKey\ eL\ eR) \Downarrow (\mathsf{Branch}\ vKey\ vL\ vR)}\ \text{Eval-Branch}$$

$$\frac{eTree \Downarrow (\mathsf{Leaf}\ A) \qquad eLeaf \Downarrow v}{(\mathsf{Tree\text{-}case}\ \ eTree\ eLeaf\ \ xKey\ \ xL\ xR\ \ eBranch) \Downarrow v}\ \text{Eval-Tree-case-Leaf}$$

$$\frac{eTree \Downarrow (\mathsf{Branch}\ vKey\ vL\ vR) \qquad [vR/xR]\,[vL/xL]\,[vKey/xKey]\,eBranch \Downarrow v}{(\mathsf{Tree\text{-}case}\ \ eTree\ eLeaf\ \ xKey\ \ xL\ xR\ \ eBranch) \Downarrow v}\ \text{Eval-Tree-case-Branch}$$

4.3 Substitution

This has all been implemented for you... but make sure you understand the new parts of the definition!

Substitution for Typed Fun abstract syntax

$$
\begin{aligned}
[e2/x](\mathsf{Num}\ n) &= (\mathsf{Num}\ n)\\
[e2/x](\mathsf{Leaf}\ A) &= (\mathsf{Leaf}\ A)\\
[e2/x](\mathsf{Branch}\ eKey\ eL\ eR) &= (\mathsf{Branch}\ [e2/x]eKey\ [e2/x]eL\ [e2/x]eR)\\
[e2/x](\mathsf{Tree\text{-}case}\ e\ eLeaf\ x1\ x2\ x3\ eBranch) &= (\mathsf{Tree\text{-}case}\ [e2/x]e\ [e2/x]eLeaf\ x1\ x2\ x3\ eBranch)\\
&\quad \text{if } x = x1 \text{ or } x = x2 \text{ or } x = x3\\
[e2/x](\mathsf{Tree\text{-}case}\ e\ eLeaf\ x1\ x2\ x3\ eBranch) &= (\mathsf{Tree\text{-}case}\ [e2/x]e\ [e2/x]eLeaf\ x1\ x2\ x3\ [e2/x]eBranch)\\
&\quad \text{if } x \neq x1 \text{ and } x \neq x2 \text{ and } x \neq x3\\
[e2/x](\mathsf{Id}\ x) &= e2\\
[e2/x](\mathsf{Id}\ y) &= (\mathsf{Id}\ y) \quad \text{if } x \neq y\\
[e2/x](\mathsf{Lam}\ x\ A\ eB) &= (\mathsf{Lam}\ x\ A\ eB)\\
[e2/x](\mathsf{Lam}\ y\ A\ eB) &= (\mathsf{Lam}\ y\ A\ [e2/x]eB)\\
&\quad \text{if } x \neq y\\
[e2/x](\mathsf{App}\ eFun\ eArg) &= (\mathsf{App}\ [e2/x]eFun\ [e2/x]eArg)\\
[e2/x](\mathsf{Binop}\ op\ eL\ eR) &= (\mathsf{Binop}\ op\ [e2/x]eL\ [e2/x]eR)\\
[e2/x](\mathsf{Pair}\ eL\ eR) &= (\mathsf{Pair}\ [e2/x]eL\ [e2/x]eR)\\
[e2/x](\mathsf{Bfalse}) &= (\mathsf{Bfalse})\\
[e2/x](\mathsf{Btrue}) &= (\mathsf{Btrue})\\
[e2/x](\mathsf{Ite}\ e\ eThen\ eElse) &= (\mathsf{Ite}\ [e2/x]e\ [e2/x]eThen\ [e2/x]eElse)\\
[e2/x](\mathsf{Let}\ x\ e\ eB) &= (\mathsf{Let}\ x\ [e2/x]e\ eB)\\
[e2/x](\mathsf{Let}\ y\ e\ eB) &= (\mathsf{Let}\ y\ [e2/x]e\ [e2/x]eB)\\
&\quad \text{if } x \neq y\\
[e2/x](\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eB) &= (\mathsf{Pair\text{-}case}\ [e2/x]e\ x1\ x2\ eB)\\
&\quad \text{if } x = x1 \text{ or } x = x2\\
[e2/x](\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eB) &= (\mathsf{Pair\text{-}case}\ [e2/x]e\ x1\ x2\ [e2/x]eB)\\
&\quad \text{if } x \neq x1 \text{ and } x \neq x2\\
[e2/x](\mathsf{Rec}\ x\ A\ eB) &= (\mathsf{Rec}\ x\ A\ eB)\\
[e2/x](\mathsf{Rec}\ y\ A\ eB) &= (\mathsf{Rec}\ y\ A\ [e2/x]eB)\\
&\quad \text{if } x \neq y
\end{aligned}
$$

5 Typing

5.1 Notation

Racket comes from the Lisp/Scheme tradition, so it is (1) untyped and (2) favours prefix notation for everything. Your instructor comes from the "type theory" tradition, which is—not surprisingly—typed, but also inclined towards infix notation.

For consistency with the vast majority of work on type systems, we will use *infix* types, like $\mathsf{num} \rightarrow \mathsf{bool}$, in the typing rules. But we will use *prefix* types, like `{-> num bool}`, in the concrete syntax. The **define-type** for `Type` uses prefix notation, e.g. `(T-> (Tnum) (Tbool))`, because **define-type** only supports that notation.

5.2 Typing rules

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

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) : A}\ \text{Type-var}$$

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{Type-num} \qquad \frac{op : A1 * A2 \rightarrow B \qquad \Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Binop}\ op\ e1\ e2) : B}\ \text{Type-binop}$$

$$\frac{}{\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\ \text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{Btrue}) : \mathsf{bool}}\ \text{Type-true}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ eBody) : A \rightarrow B}\ \text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \rightarrow B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}\ \text{Type-app}$$

$$\frac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) : A1 * A2}\ \text{Type-pair} \qquad \frac{\Gamma \vdash e : A1 * A2 \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eBody) : B}\ \text{Type-pair-case}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Let}\ x\ e\ eBody) : B}\ \text{Type-with} \qquad \frac{u : B, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Rec}\ u\ B\ e) : B}\ \text{Type-rec}$$

$$\frac{}{\Gamma \vdash (\mathsf{Leaf}\ A) : \mathsf{tree}\ A}\ \text{Type-leaf} \qquad \frac{\Gamma \vdash eKey : A \qquad \Gamma \vdash eL : \mathsf{tree}\ A \qquad \Gamma \vdash eR : \mathsf{tree}\ A}{\Gamma \vdash (\mathsf{Branch}\ eKey\ eL\ eR) : \mathsf{tree}\ A}\ \text{Type-branch}$$

$$\frac{\Gamma \vdash e : \mathsf{tree}\ A \qquad \Gamma \vdash eLeaf : B \qquad xk : A, xL : \mathsf{tree}\ A, xR : \mathsf{tree}\ A, \Gamma \vdash eBranch : B}{\Gamma \vdash (\mathsf{Tree\text{-}case}\ e\ eLeaf\ xk\ xL\ xR\ eBranch) : B}\ \text{Type-tree-case}$$

Figure 1 Typing rules, including trees (Leaf/Branch/Tree-case)

6 Problems

The first problem is not a coding problem. Turn it in on paper in the box in room X235. You can print this sheet and write on it, or write out your answers on a separate sheet. **Be sure to write your name, student number, and CS userid.** If you are working with someone else, turn in **one** solution.

	Name	Student ID#	CS userid
Student #1:			
Student #2:			

Problem 1: Typing derivations

Following the "method of hope" from `lec-functions.pdf`, complete the following two **typing** derivations. The rules are given above.

When you write a horizontal line, write the rule name next to it.

Remember: this is *typing*. Follow the typing rules, not the evaluation rules!

1(a).

$$n : \mathsf{num}, f : \mathsf{num} \to \mathsf{bool}, \emptyset \vdash (\mathsf{App}\ (\mathsf{Id}\ f)\ (\mathsf{Id}\ n)) : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$$

1(b). Hint: Do 1(a) first. Write a checkmark above a judgment if you've already derived it elsewhere.

$$\emptyset \vdash \Big(\mathsf{Rec}\ f\ \mathsf{num}{\to}\mathsf{bool}\ \big(\mathsf{Lam}\ n\ \mathsf{num}\ (\mathsf{App}\ (\mathsf{Id}\ f)\ (\mathsf{Id}\ n))\big)\Big)\ :\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$$

IMPORTANT: Read this before you start coding!

- Don't worry about making your code fast; clarity and correctness are much more important.

  You must implement code that follows the rules; you should also make your code similar to the rules.

- For a2, we said that you must write test cases for the features you're implementing.

  For this assignment, we will strongly recommend, but not require, that you write test cases for the features you're implementing *to test type checking*; you don't have to worry about whether we implemented the evaluation semantics correctly. As usual, if your code is "almost" correct, but you wrote a good set of test cases, you will get more marks for having written tests.

Problem 2: Products

In `typeof`, fill in the Pair and Pair-case branches, following the rules in Figure 1.

Problem 3: Boolean balloons

For each of the following, **either** write the specified Fun expression, **or** explain why you think no such expression exists.

Look for the section of `a3.rkt` marked "Problem 3". You can write the expressions in concrete syntax and call `parse`, or write them in abstract syntax. Writing in concrete syntax is usually easier. Each Racket expression that `expr3a`, `expr3b`, etc. is bound to must be either `#false` or an `E`. Some tests in `a3.rkt` will check this for you.

- **Part 3a**: Write a Fun expression that is successfully evaluated, but is rejected by the type checker `typeof`.

  The Fun expression you write must *not* be a value, and the value it evaluates to must not be a Lam. (But you might want to come up with those examples first.)

- **Part 3b**: Write a Fun expression that does not evaluate to a value, but is accepted by the type checker.

- **Part 3c**: Write a Fun function—that is, an expression that evaluates to a Lam—that has type bool $\rightarrow$ bool, and that, when applied, evaluates to a value *if and only if* its argument evaluates to (Bfalse).

- **Part 3d**: Write a Fun function that has type (bool $\rightarrow$ bool) $\rightarrow$ bool and that, when applied to *any* function `f` of type bool $\rightarrow$ bool, evaluates to a value *if and only if* (`App f` (Bfalse)) evaluates to a value.

- **Part 3e**: Write a Fun function that has type (bool $\rightarrow$ bool) $\rightarrow$ bool, and that, when applied to *any* function of type bool $\rightarrow$ bool, evaluates to a value *if and only if* (`App f` (Bfalse)) does *not* evaluate to a value.

Problem 4: Trees

In `typeof`, fill in the Leaf, Branch and Tree-case branches, following the rules in Figure 1.

7 Bonus problems

If you choose to complete a bonus, submit a separate `bonus.rkt` file with your bonus work in it. (If your bonus work passes all of our handin tests, this can just be a copy of your regular submission, but it's conceivable that your bonus will violate our tests! Plus, this way you can solve the main assignment, set that work aside, and then try for the bonus without endangering your main grade.) Also note that we reserve the right to summarily give no credit to ill-documented or ill-tested bonus submissions.

Warning! Some bonus problems might be unsolvable.

Bonus problem X4: define-type + type-case

Add your own features analogous to **define-type** and **type-case** to Typed Fun.

This is a deliberately open-ended problem, and you're free to scale it down to fit whatever remaining time you have. For example, allowing only **define-type**s with a single variant will be easier (reminiscent of the `Binding` **define-type** used to implement a2, or the `Pair` in a3).