Compiler Construction/Fall 2014/Project Milestone 2 part A
Solution

Eva Rose
evarose@cs.nyu.edu

Kristoffer Rose
krisrose@cs.nyu.edu

Released Wednesday 11/5/2014

This is a proposed solution to part A of project milestone 2, which was to formalize the type system of the fictional programming language JST as a syntax-directed definition (SDD). The following should be seen as the content of the requested pr2a-*YourName*.pdf file.

---

1 SDD for JST

Our task is to formalize a type checker for the "JST" language, as a syntax-directed definition. We provide that with a series of definitions below, each referering with "A." references to the relevant definition in the part A assignment.

1.1 Notation (abstract syntax notation)**.** We will use an abstract version of the syntax, where we systematically abbreviate all nonterminal symbols to their first letter (or first few), so P is Program, D is Declaration, *etc.* We also use starred versions for lists, so D^* is Declarations, for example. We stick with abstract syntax, so the production rules will sometimes ignore details such as commas between arguments.

1.2 Definition (syntactic type)**.** JST already has a notion of type, namely the syntactic types described by the grammar. For the purpose of the SDD we will model JST syntactic types with the following production:

$$T \to \textbf{boolean} \mid \textbf{int} \mid \textbf{string} \mid \textbf{void} \mid \textbf{id}$$

where T stands for "type" and the only special token is **id**, which denotes the JST identifiers.

1.3 Definition (maps)**.** We will frequently work with maps from keys to values. These structures are written $\{\dots, k \mapsto v, \dots\}$, collecting a list of key k to value v mappings. A map m allows the following auxiliary operations:

$$\begin{aligned}
\mathrm{Defined}(m,k)&\text{ whether } m \text{ has a value for key } k\\
\mathrm{Lookup}(m,k)&\text{ the value } v \text{ that } m \text{ has for key } k\text{, or default to } \textbf{error}\\
\mathrm{Extend}(m,k,v)&\text{ a } \textit{new} \text{ map which adds the mapping } k \mapsto v \text{ to those of } m
\end{aligned}$$

A JST program consists of a sequence of declarations, which all add to the top scope. In particular the **class** declarations add named types to the top scope.

The top scope means that when we type check a program fragment then we must have the *full* type environment available, already for the code inside the first class declaration. This means that our SDD cannot be left-to-right but has to create all the top level types before type checking can be done. We achieve this by separating the two issues: a synthesized attribute dl contains a *descriptor list*, and a separate synthesized attribute ok reports if every declaration is well typed under the assumption of the types in the descriptor list.

1.4 Definition (descriptor list and type environments)**.** Since the SDD needs to describe the composition of the top environment, we give the structure here: a *descriptor list* has the structure

$$DL = (\ \textbf{id} \mapsto TD, \dots\ )$$

where each top level **id** is thus associated with a "type descriptor" described below. A *type environment* is a descriptor list used as a map from identifiers to type descriptors.

1.5 Definition (type descriptor)**.** To fully describe all the types, we *extend* the notion of type with a semantic notion of "type descriptor," which is associated with a name. Type descriptors have the structure

$$TD = T \mid \text{Call } T(T, \dots) \mid \text{Class}(\ \textbf{id} \mapsto TD, \dots\ ) \mid \textbf{error}$$

The idea is that we describe a declared function by its call signature written $\text{Call } T(T_1, \dots, T_n)$ with T the result type and T_i the argument types, and a class as $\text{Class}(m_1 \mapsto TD_1, \dots, m_n \mapsto TD_n)$ where the m_i are the members (fields and methods) and the TD_i are the "member descriptors" which are type descriptors except. Finally, in addition we have added the special value "**error**" to capture type errors.

1.6 Definition (assignment compatible types, A.2.1, A.2.2)**.** This formalizes what it means to be *assignment compatible* in the assignment. Notice that the notion is not symmetric, and that we have restricted assigment to not allow assigning from void values. Part A has one omission here: assignment compatibility is extended for object literals in A.2.2, which is not included in A.2.1. We fix this with a special case for class types.

$$\text{AssignCompat}(e, t_1, t_2) = \begin{cases} \text{true} & \text{if } t_1 = t_2 \notin \{\textbf{void}, \textbf{error}\} \\ \text{true} & \text{if } t_1 = \textbf{any} \text{ and } t_2 \notin \{\textbf{void}, \textbf{error}\} \\ \text{true} & \text{if } t_1 = \textbf{id} \text{ and } \text{SameClass}(\text{Lookup}(e, \textbf{id}), t_2) \\ \text{false} & \text{otherwise} \end{cases}$$

where e must be the current mapping from (type) names to type descriptors, and $\text{SameClass}(t_1, t_2)$ is true only if $t_1 = \text{Class}(\textbf{id} \mapsto T \dots)$ and $t_2 = \text{Class}(\textbf{id}' \mapsto T' \dots)$ and the two classes have the same members with the same types but in any order. We take care when checking the assignment expression later that this rule only applies to object literals, not assignment between variables.

Now we are ready to define our strategy.

PRODUCTION	SEMANTIC RULES
$P \to D_1^*$	$D_1^*.e = D_1^*.dl; P.ok = D_1^*.ok$

Figure 1: SDD for type checking main program.

1.7 Definition (strategy for program typing)**.** The top SDD rule in Figure 1 for an entire program uses these attributes:

- dl is a top level descriptor lists *DL*, synthesised for programs P, and declarations D^* and D.
- e is a type environment inherited by declarations; it is a *DL* used as a map.
- ok is a boolean synthesized attribute from the program and declarations that indicates typing is alright. (Note that this goes beyond what is asked in part A.)

The strategy requires two passes over the declarations of the program: the first to synthesize the dl attribute, and the second to synthesize the ok attribute with access to the inherited e attribute: this second pass will also populate all expressions with type annotations (which os what the part A assignment asks).

Notice that this goes a bit beyond what part A requires: part A only requires that the SDD annotates all expressions with their type, not that it is checked! Our SDD checks as a way to force the second traversal of the program.

Now we can define the details of the SDD fragment for collecting the top level types and checking the types. For clarity we split the SDD for the declarations level in two: one that details the actions related to top level collection, one that details the type checking itself.

PRODUCTION	SEMANTIC RULES
$D^* \to D_1\ D_2^*$	$D^*.dl = D_1.dl + D_2^*.dl$
$\mid \epsilon$	$D^*.dl = ()$
$D \to$ **class id**$_1\{M_2^*\}$	$D.dl = (\ \mathbf{id}_1 \mapsto M_2^*.mds\ )$
$\mid$ **function** T_1 **id**$_2$ **(** TI_3^* **)** { S_4 }	$D.dl = (\ \mathbf{id}_2 \mapsto \text{Call}\ T_1(TI_3^*.ts)\ )$
$M^* \to M_1\ M_2^*$	$M^*.mds = (M_1.md) + M_2^*.mds$
$\mid \epsilon$	$M^*.mds = ()$
$M \to T_1$ **id**$_2$ **;**	$M.md = \mathbf{id}_2 : T_1$
$\mid T_1$ **id**$_2$ **(** TI_3^* **) {** S_4 **} ;**	$M.md = \mathbf{id}_2 : \text{Call}\ T_1(TI_3^*.ts)$
$TI^* \to T_1\ I_1,\ TI_2^*$	$TI^*.ts = (T_1) + TI_2^*.ts$
$\mid \epsilon$	$TI^*.ts = ()$

Figure 2: SDD fragment for collecting top level types.

1.8 Definition (SDD for collecting top level typing, A.2.7, A.2.8)**.** Collecting types is described by the SDD fragment in Figure 2, with these attributes:

- dl is a top level descriptor lists *DL*, synthesised for declarations D^* and D.
- mds is a list of member descriptors *MD*s, syntesized by member lists M^*.
- md is a single type descriptor *MD*, synthesized by members M.
- ts is a list of types, synthesized for lists of type/identifier pairs *TI**.

where

$$MD = \mathbf{id} : TD$$

We use the $+$ operator for list concatenation.

1.9 Definition (SDD for type checking declarations, A.2.6)**.** Figure 3 has the SDD fragment for type checking declarations. It uses these attributes:

- e is the type environment (a *DL* used as a map), inherited by the nonterminals S, D, M^*, M, S.
- ok is a boolean synthesized attribute from D^*, D, M^*, M, S, and indicates whether type checking was successful.

PRODUCTION	SEMANTIC RULES
$D^* \to D_1\ D_2^*$	$D_1.e = D^*.e; D_2^*.e = D^*.e; D^*.ok = D_1.ok \wedge D_2^*.ok$
$\quad \mid \epsilon$	$D^*.ok = \text{true}$
$D \to \textbf{class}\ \textbf{id}_1 \{ M_2^* \}$	$M_2^*.e = \text{Extend}(D.e, \textbf{this}, \textbf{id}_1); D.ok = M^*.ok$
$\quad \mid \textbf{function}\ T_1\ \textbf{id}_2\ (\ TI_3^*\ )\ \{\ S_4\ \}$	$S_4.e = \text{Extend}(\text{Extend}(D.e, TI_3^*), \textbf{return}, T_1); D.ok = S_4.ok$
$M^* \to M_1\ M_2^*$	$M_1.e = M^*.e; M_2^*.e = M^*.e; M^*.ok = M_1.ok \wedge M_2^*.ok$
$\quad \mid \epsilon$	$M^*.ok = \text{true}$
$M \to T_1\ \textbf{id}_2\ ;$	$M.ok = \text{true}$
$\quad \mid T_1\ \textbf{id}_2\ (\ TI_3^*\ )\ \{\ S_4\ \}\ ;$	$S_4.e = \text{Extend}(\text{Extend}(M.e, TI_3^*), \textbf{return}, T_1); M.ok = S_4.ok$

Figure 3: SDD fragment for type checking declarations.

We permit the usual logical operations on boolean values, as well as extending an environment based on a list of type-identifier pairs:

$$\text{Extend}(e,\ T\ I\ TI^*\ ) = \text{Extend}(\ \text{Extend}(e,\ TI^*\ )\ , I, T)$$
$$\text{Extend}(e,\ \epsilon\ ) = e$$

Notice that in the environment we map the special "symbols" **this** and **return** to the type of the containing object of methods and the return type of functions.

PRODUCTION	SEMANTIC RULES
$S \to \epsilon$	$S.ok = \text{true}$
$\quad \mid \{S_1\} S_2$	$S_1.e = S.e; S_2.e = S.e; S.ok = S_1.ok \wedge S_2.ok$
$\quad \mid \textbf{var}\ T_1\ I_2\ ;\ S_3$	$S_3.e = \text{Extend}(S.e, I_2, T_1); S.ok = S_3.ok$
$\quad \mid\ ;\ S_1$	$S_1.e = S.e; S.ok = S_1.ok$
$\quad \mid E_1\ ;\ S_2$	$E_1.e = S.e; S_2.e = S.e; S.ok = (E_1.ts$ contains no **error**$) \wedge S_2.ok$
$\quad \mid \textbf{if}\ (E_1)\ S_2\ S_3$	$E_1.e = S.e; S_2.e = S.e; S_3.e = S.e;$ $S.ok = (E_1.ts = (\textbf{boolean})) \wedge S_2.ok \wedge S_3.ok$
$\quad \mid \textbf{if}\ (E_1)\ S_2\ \textbf{else}\ S_3\ S_4$	$E_1.e = S.e; S_2.e = S.e; S_3.e = S.e; S_4.e = S.e;$ $S.ok = (E_1.ts = (\textbf{boolean})) \wedge S_2.ok \wedge S_3.ok \wedge S_4.ok$
$\quad \mid \textbf{while}\ (E_1)\ S_2\ S_3$	$E_1.e = S.e; S_2.e = S.e; S_3.e = S.e;$ $S.ok = (E_1.ts = (\textbf{boolean})) \wedge S_2.ok \wedge S_3.ok$
$\quad \mid \textbf{return}\ E_1\ ;\ S_2$	$E_1.e = S.e; S_2.e = S.e;$ $S.ok = \text{AssignCompat}(\text{Lookup}(S.e, \textbf{return}), E_1.ts) \wedge S_2.ok$
$\quad \mid \textbf{return}\ ;\ S_1$	$S_1.e = S.e; S.ok = (\text{Lookup}(S.e, \textbf{return}) = \textbf{void}) \wedge S_1.ok$

Figure 4: SDD fragment for type checking statements.

1.10 Definition (SDD for type checking statements, A.2.5)**.** We will use a tail recursive syntax for statements:

$$S \to \epsilon \mid \{S\}S \mid \textbf{var}?T\ I\ ;\ S \mid\ ;\ S \mid E\ ;\ S$$
$$\mid \textbf{if}\ (E)\ S\ S \mid \textbf{if}\ (E)\ S\ \textbf{else}\ S\ S$$
$$\mid \textbf{while}\ (E)\ S\ S \mid \textbf{return}\ E\ ;\ S \mid \textbf{return}\ ;\ S$$

This makes it easier to pass the type environment around, and is used in the SDD fragment in Figure 4 for type checking statements using the following attributes:

- e is the type environment (a *DL* used as a map), inherited by S, E.
- t is a type synthesized by E with the type of the expression.
- ok is a boolean synthesized attribute from S, E, and indicates whether type checking was successful.

Notice that the definitions of the project formulation (project milestone 2 part A) does not mention that a **return** statement should be type checked: the type should be consistent with the enclosing callable unit (function or method).

1.11 Definition (SDD for type checking expressions, A.2.3, A.2.4, A.2.9)**.** Figure 5 has the rules for type checking expressions. The following attributes are used:

- e is the type environment, a map from identifiers (including **this** and **return**) to type descriptors *TD*, inherited by E.
- ts is a list of type descriptors, synthesized by E, with the types of that E, including the special values **error** and object literal type descriptors.

In addition we add the following helpers:

$$\mathrm{If}(b, x, y) = \begin{cases} x & \text{if } b \text{ evaluates to true} \\ y & \text{otherwise} \end{cases}$$

$$\mathrm{ResolveCall}(ts_1, ts_2) = \begin{cases} T & \text{if } ts_1 = (\text{Call } T(T_1, \ldots, T_n)) \text{ and } ts_2 = (T_1, \ldots, T_n) \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\mathrm{ResolveMember}(ts, \textbf{id}) = \begin{cases} T_i & \text{if } ts = (\mathrm{Class}(\textbf{id}_1 \mapsto T_1, \ldots, \textbf{id}_n \mapsto T_n)) \text{ with } \textbf{id} = \textbf{id}_i \\ \textbf{int} & \text{if } ts = (\textbf{string}) \text{ and } \textbf{id} = \mathrm{length} \\ \text{Call } \textbf{string}(\textbf{int}) & \text{if } ts = (\textbf{string}) \text{ and } \textbf{id} = \mathrm{charCodeAt} \\ \text{Call } \textbf{string}(\textbf{int}, \textbf{int}) & \text{if } ts = (\textbf{string}) \text{ and } \textbf{id} = \mathrm{substr} \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\mathrm{LValue}(E) = \begin{cases} \mathrm{true} & \text{if } E \text{ has the form } \textbf{id} \text{ or } E.\textbf{id} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

Notice how the special operators on **string** values are incorporated in $\mathrm{ResolveMember}$ and the special typing for **string**/**int** concatenation is incorporated in the SDD rules.

1.12 Definition (SDD for type checking literals, A.2.2)**.** Figure 6 contains the type checking for literals. Literals only have the synthesized t type attribute. Notice how object literals build a "Class" structure: this works with assignment compatibility to ensure that object initializations are allowed.

PRODUCTION	SEMANTIC RULES
$E \to \mathbf{id}_1$	$E.ts = (\text{Lookup}(E.e, \mathbf{id}_1))$
$\mid L_1$	$E.ts = (L.t)$
$\mid$ **this**	$E.ts = (\text{Lookup}(E.e, \textbf{this}))$
$\mid E_1(E_2)$	$E_1.e = E_2.e = E.e; E.ts = (\text{ResolveCall}(E_1.ts, E_2.ts))$
$\mid E_1()$	$E_1.e = E.e; E.ts = (\text{ResolveCall}(E_1.ts, ()))$
$\mid E_1$ **.** $\mathbf{id}_2$	$E_1.e = E.e; E.ts = (\text{ResolveMember}(E_1.ts, \mathbf{id}_2))$
$\mid$ **!** E_1	$E_1.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{boolean}), (\textbf{boolean}), (\textbf{error}))$
$\mid -E_1$	$E_1.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid +E_1$	$E_1.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid E_1 * E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid E_1/E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid E_1 \% E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid E_1 + E_2$	$E_1.e = E_2.e = E.e;$ $E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{int}),$ $\text{If}((E_1.ts = (\textbf{string}) \vee E_1.ts = (\textbf{int}))$ $\wedge (E_2.ts = (\textbf{string}) \vee E_2.ts = (\textbf{int})), (\textbf{string}), (\textbf{error})))$
$\mid E_1 - E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{int}), (\textbf{error}))$
$\mid E_1 < E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1 > E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1 <= E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1 >= E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1 == E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = E_2.ts, (\textbf{boolean}), (\textbf{error}))$
$\mid E_1 != E_2$	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = E_2.ts \wedge E_1.ts = (\textbf{error}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1$ **&&** E_2	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{boolean}) \wedge E_2.ts = (\textbf{boolean}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1$ **||** E_2	$E_1.e = E_2.e = E.e; E.ts = \text{If}(E_1.ts = (\textbf{boolean}) \wedge E_2.ts = (\textbf{boolean}), (\textbf{boolean}), (\textbf{error}))$
$\mid E_1$ **=** E_2	$E_1.e = E_2.e = E.e; E.ts = \text{If}(\text{LValue}(E_1) \wedge \text{AssignCompat}(E_1.ts, E_2.ts), E_2.ts, (\textbf{error}))$
$\mid E_1$ **+=** E_2	$E_1.e = E_2.e = E.e;$ $E.ts = \text{If}(\text{LValue}(E_1)$ $\wedge((E_1.ts = (\textbf{int}) \wedge E_2.ts = (\textbf{int}))$ $\vee(E_1.ts = (\textbf{string}) \wedge (E_2.ts = (\textbf{int}) \vee E_2.ts = (\textbf{int})))), E_1.t, (\textbf{error}))$
$\mid E_1$ **,** E_2	$E_1.e = E_2.e = E.e; E.ts = E_1.ts + E_2.ts$

Figure 5: SDD fragment for type checking expressions.

PRODUCTION	SEMANTIC RULES
$L \to$ **int**	$L.t =$ **int**
$\mid$ **string**	$L.t =$ **string**
$\mid \{\mathbf{id}_1 : L_1, \ldots, \mathbf{id}_n : L_n\}$	$L.t = \text{Class}(\mathbf{id}_1 : L_1.t, \ldots, \mathbf{id}_n : L_n.t)$

Figure 6: SDD fragment for type checking literals.