TP n° 12

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Le but de ce TP est d'enrichir le langage orienté objet typé vu dans `typed-class.rkt`. Les fonctionnalités à ajouter sont pour la plupart inspirées du langage Java. Les exercices sont globalement indépendants. Vous pouvez les traiter dans n'importe quel ordre.

Lors de l'introduction de nouvelles expressions, les règles de typage associées ne sont pas fournies. Il vous revient de les déduire à partir de la sémantique et de vos connaissances. De manière générale, tout problème que l'on peut détecter statiquement (sans évaluer le code) **doit** produire une erreur lors de la vérification des types.

Sauf mention contraire, vous êtes libres de définir vos propres messages d'erreur. Cela vous permettra de trouver vos éventuelles bugs plus facilement. La seule règle à respecter concerne le symbole sur lequel est basé l'erreur. Il dépend de l'étape dans l'interpréteur et de la cause du problème.

Étape	Cause	Erreur autorisée
Analyse syntaxique	Toute	`(error 'parse msg)`
Vérification des types	Classe, méthode ou variable non trouvée	`(error 'find msg)`
	Autres erreurs	`(error 'typecheck msg)`
Interprétation	Toute	`(error 'interp msg)`

Covariance

Lorsqu'une méthode est une redéfinition d'une autre, autorisez la covariance sur le type de retour et la contravariance sur l'argument.

Valeur `null`

Ajoutez une constante `null` au langage :

```
<Exp> ::= ...
        | null
```

L'interprétation de cette constante produit une nouvelle valeur appelée valeur null. Cette valeur n'est pas un objet. À ce titre, tenter d'accéder à un champ ou appeler une méthode sur la valeur null doit produire une erreur à l'interprétation. Cependant la valeur null peut-être utilisée à la place de

n'importe quel objet (initialisation d'un champ, argument d'une méthode, ...). S'il n'y a ni accès à un champ ni appel de méthode, aucune erreur ne se produit.

Lors de la vérification des types, on doit s'assurer que `null` n'est jamais utilisé à la place d'un nombre (ou d'un booléen, cf dernier exercice). On doit également refuser les expressions qui tentent d'accéder directement à un membre de la valeur null :

— `{get null x}`,
— `{send null m 0}`, ...

En dehors de cela, la valeur null est autorisée partout où est attendu un objet.

Transtypage

Ajoutez une expression `cast` au langage :

```
<Exp> ::= ...
        | {cast <Exp> <Type>}
```

Il s'agit d'une instruction de transtypage comme en Java. L'expression `{cast e t}` a la même valeur que `e` mais le type `t`. Pendant l'interprétation, on vérifie que `e` s'évalue en un objet d'un type compatible avec `t`. Pour que le transtypage ait une chance de réussir, il est nécessaire que le type de `e` soit un sous-type ou un super-type de `t`. Cette dernière propriété doit être assurée dès la vérification des types.

Branchement conditionnel

Ajoutez les booléens, le branchement conditionnel et l'égalité entre nombres au langage.

```
<Exp> ::= ...
        | true
        | false
        | {= <Exp> <Exp>}
        | {if <Exp> <Exp> <Exp>}

<Type> ::= ...
         | bool
```

Les règles de typage et la sémantique sont les mêmes que vu en cours à l'exception de la règle de typage du branchement conditionnel :

$$\frac{\Gamma \vdash \texttt{e1 : bool} \qquad \Gamma \vdash \texttt{e2 : t1} \qquad \Gamma \vdash \texttt{e3 : t2} \qquad \texttt{t3 = sup(t1, t2)}}{\Gamma \vdash \texttt{\{if e1 e2 e3\} : t3}}$$

La fonction `sup` est la fonction qui renvoie le plus petit super-type commun à deux sous-types (s'il existe). Autrement dit,

$$\texttt{sup(t1,t2) = t3} \Leftrightarrow \begin{cases} \texttt{t1} \leq \texttt{t3} \\ \texttt{t2} \leq \texttt{t3} \\ \forall \texttt{t}, \texttt{t1} \leq \texttt{t} \text{ et } \texttt{t2} \leq \texttt{t} \Rightarrow \texttt{t3} \leq \texttt{t} \end{cases}$$