Prévention du déréférencement de références nulles dans un langage à objets

Jean-Sébastien Gélinas, Étienne M. Gagnon, Jean Privat

Université du Québec à Montréal
gelinas.jean-sebastien@courrier.uqam.ca, {gagnon.etienne_m,privat.jean}@uqam.ca

Résumé. Le déréférencement de références nulles est une erreur de programmation courante dans les langages à objets. Pour la prévenir, certaines approches garantissent statiquement son absence à l'aide de systèmes de types ou d'annotations mais réduisent l'expressivité du langage. D'autres approches analysent plutôt le code source pour identifier les erreurs potentielles, mais peuvent trouver des faux-positifs et ne garantissent pas l'absence d'erreurs à l'exécution. Dans cet article, nous proposons une approche offrant la garantie statique d'absence d'erreur de déréférencement dans une grande portion du code. L'approche consiste en un système de types statiques simple, des vérifications dynamiques et un opérateur de test dynamique. En plus de préserver l'expressivité du langage, notre approche limite la zone de danger à la construction des instances et permet la détection précoce des erreurs à l'exécution. Nos mesures expérimentales démontrent une grande étendue des garanties statiques et l'efficacité de la détection précoce des erreurs.

1 Introduction

Le déréférencement d'une référence nulle est une erreur de programmation fréquente dans les langages à objets. Normalement cette erreur cause l'arrêt prématuré du logiciel en cours d'exécution. Certains langages permettent de récupérer cette erreur avec un système d'exception (`try`/`catch` en JAVA et C#), mais ceci constitue une mesure extraordinaire pour les logiciels qui ne peuvent se permettre d'arrêter. En fait, le problème réel n'est souvent pas dû au déréférencement lui-même ; il est plutôt dû au fait que la variable (ou l'attribut) déréférencé *contient* la valeur `null` à ce moment. Le code erroné se situe donc normalement ailleurs, à un endroit potentiellement lointain où on a soit oublié d'initialiser la variable, soit stocké erronément `null` dans la variable.

Dans cet article, nous présentons le résultat de notre recherche d'une approche à utiliser dans un nouveau langage de programmation à objets pour prévenir le déréférencement de références nulles sans sacrifier, pour autant, l'expressivité du langage.

Il pourrait sembler simple de modifier un système de types similaire à celui des langages JAVA et C# pour indiquer, à même le type, si `null` est une valeur permise ou non. Malheureusement, la construction des objets cause problème car il n'y a généralement pas de valeur non-nulle appropriée à stocker par défaut dans un attribut de type non-nul lors de l'allocation de

```
class A {
 notnull X x;
 A() {
  ... // où x n'est pas initialisé
 }
 void m() {
  x.p(); // boum!
 }
}
// ailleurs dans le code
...
A a = new A();
...
a.m();
```

FIG. 1 – *Problème d'accès à un attribut avant son initialisation.*

l'objet, avant l'appel des constructeurs[1]. L'exemple de la figure 1 illustre le problème[2] : lors de l'exécution du constructeur `A()`, l'attribut `x` n'est pas initialisé. Plus tard, lors de l'invocation de la méthode `m()`, l'attribut `x` est déréférencé dans le but d'appeler la méthode `p()` causant une erreur de déréférencement d'un attribut non-initialisé.

Plutôt que d'essayer de prévenir une telle erreur en restreignant l'expressivité par un système de types plus complexe, nous proposons un modèle de programmation comprenant trois parties : un système de types statiques simple, des vérifications dynamiques et un opérateur de test dynamique. Ce système permet de détecter rapidement les erreurs de programmation par sa politique d'échec précoce (*fail fast*) et offre la garantie de non-déréférencement de référence nulle dans une grande portion du code. L'originalité de notre approche est que la zone de danger (où la garantie n'est pas offerte) est limitée dynamiquement à la construction des instances.

Le reste de cet article est structuré comme suit. Dans la section 2, nous détaillons notre approche. Dans la section 3, nous expliquons la portée dynamique de la garantie de non-déréférencement de références nulles. Dans la section 4, nous prouvons que notre système ne réduit pas l'expressivité du language de programmation. Dans la section 5, nous présentons une validation expérimentale de notre proposition. Dans la section 6, nous comparons notre approche aux travaux d'autres chercheurs. Enfin, nous concluons notre article dans la section 7.

2 Approche proposée

Nous proposons une approche hybride qui combine des règles de typage statique et des vérifications dynamiques. De plus, nous ajoutons au langage un opérateur qui permet de tester dynamiquement si un attribut a été initialisé ou non.

[1] Normalement, les attributs sont pré-initialisés avec la valeur `null`, ce qui n'est pas approprié pour un attribut de type statique non nullable.

[2] Afin de simplifier la présentation, nous utilisons une syntaxe apparentée à celle de JAVA dans nos exemples.

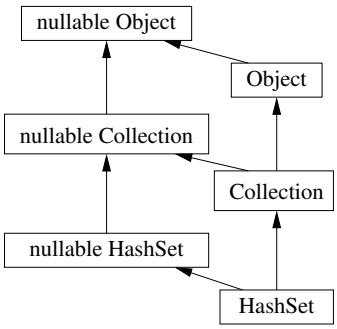

FIG. 2 – *Système de types.*

2.1 Typage statique

Modification du système de types

Nous modifions un système de types standard similaire à celui de JAVA en ajoutant le mot clé `nullable` pour annoter les types qui peuvent contenir la valeur `null`. Par opposition, un type qui n'est pas annoté `nullable` ne peut pas contenir la valeur `null`[3].

On intègre les types nullables au système de types comme suit. Pour chaque type non-nullable t, on note $n(t)$ le type nullable correspondant. Un type t est sous-type de sa version nullable $n(t)$. De plus, la relation de sous-typage est préservée par l'annotation `nullable`. Plus précisément, on pose :

$$\forall t, \quad t <: n(t)$$

et

$$\forall t, t', \quad t <: t' \Rightarrow n(t) <: n(t')$$

La figure 2 illustre la relation de sous-typage entre des types nullables et non-nullables. On peut noter que la racine de la hiérarchie des types est maintenant `nullable Object`.

Règles d'utilisation des nullables

On considère qu'un type nullable $n(t)$ n'a pas de méthodes ni d'attributs. Par conséquent, les deux règles suivantes sont ajoutées :

1. l'accès à un attribut est interdit sur une expression de type nullable ;
2. l'invocation d'une méthode est interdite sur une expression de type nullable.

Exemple d'utilisation des nullables

Afin de contourner les restrictions d'utilisation des nullables, le programmeur doit s'assurer que la valeur est non-nulle par un test dynamique et effectuer la coercition appropriée. L'exemple de la figure 3 illustre un cas typique : le programmeur teste, en premier, si l'attribut `telephone` est différent de `null` ; si oui, il utilise une coercition pour invoquer la méthode `toString()`.

[3]Le mot clé `notnull` présenté dans l'exemple de la figure 1 n'est pas utilisé.

```
class Personne {
 String nom;
 nullable String telephone;
 ...
 String toString() {
  String resultat = nom;
  if (telephone != null) {
   // noter la coercition permettant l'invocation de méthode
   resultat += "; Tel.: " + ((String) telephone).toString();
  }
  return resultat;
 }
}
```

FIG. 3 – *Utilisation du mot clé* `nullable` *en pratique.*

```
...
 String toString() {
  var resultat = nom;
  var telephone = this.telephone;  // copie de l'attribut
  if (telephone != null) {
   // utilisation de '(notnull)' plutôt que '(String)'
   resultat += "; Tel.: " + ((notnull) telephone).toString();
  }
  return resultat;
 }
...
```

FIG. 4 – *Syntaxe allégée.*

Toutefois, dans cet exemple, il aurait été préférable de stocker la valeur de l'attribut dans une variable locale puis d'effectuer le test sur la variable locale pour se protéger du cas où l'attribut est modifié entre le test et la coercition. De plus, sans changer le modèle de programmation sous-jacent, on peut alléger la syntaxe en ajoutant l'inférence légère de type de C# et la coercition partielle « `(notnull)` », comme l'illustre la figure 4.

Règles d'initialisation des variables locales

Nous proposons de réutiliser les règles d'initialisation des variables locales de JAVA et C# qui imposent, par une analyse statique simple, l'initialisation des variables locales avant leur utilisation. Les règles de compatibilité des assignations (basées sur les relations de sous-typage) s'appliqueront naturellement lors de l'initialisation et préviendront les erreurs.

Notons que l'on pourrait réduire les exigences envers les variables locales de type nullable en les initialisant par défaut à `null`.

```
class A {
 T t;
 ...
 protected nullable T early_get_t() {
  if (isset t) return t;
  else return null;
 }
}
```

FIG. 5 – *Utilisation de l'opérateur* `isset`.

2.2 Vérifications dynamiques

Puisque les règles de typage statique ne garantissent pas l'initialisation des attributs non-nullables[4], nous ajoutons les deux règles de vérification dynamique suivantes :

1. l'accès à un attribut non-initialisé lève une exception E1 ;
2. à la fin de la construction d'un objet, une exception E2 est levée si un de ses attributs non-nullables n'a pas été initialisé.

Ces vérifications ont lieu à l'exécution, lors de l'accès à un attribut ou lors de la création d'une instance.

Ces vérifications précoces distinguent notre système des autres. Elles empêchent la propagation de valeurs incohérentes dans le système et réduisent considérablement la zone de danger. Nous en discuterons d'avantage dans la section 3.

2.3 Opérateur de test dynamique

Comme l'accès aux attributs non-initialisés lève immédiatement une exception E1, nous proposons l'ajout de l'opérateur `isset` qui permet dynamiquement de savoir si un attribut est initialisé ou non. Cet opérateur permet la manipulation sûre d'objets en cours de construction.

La figure 5 illustre l'utilisation de cet opérateur dans une méthode destinée à être utilisée tôt pendant la construction d'un objet : la méthode `early_get_t()` retourne la valeur de l'attribut `t` s'il est initialisé, ou `null` sinon.

2.4 Gestion des exceptions

Dans le système proposé, les exceptions E1 et E2 ne sont pas récupérables. Autrement, les garanties statiques ne tiendraient plus car l'instance qui survivrait à une erreur de construction pourrait avoir des attributs non-nullables non-initialisés.

Notons que les autres exceptions levées pendant la construction ne changent rien à nos règles dynamiques. Ainsi, lors de l'interruption de la construction d'un objet par une exception autre que E1 ou E2, une nouvelle exception E2 est levée si un de ses attributs non-nullables n'a pas encore été initialisé ; sinon, l'exception originale est propagée.

[4]Les attributs nullables sont initialisés avec `null` par défaut.

```
B b = new B(...);
b.y = a.x; // [1]: où a.x n'est pas encore initialisé
...
b.y.m(); // boum, si [1] n'a pas déjà sauté!
```

FIG. 6 – *Propagation d'une valeur indéterminée.*

3 Portée de la sûreté

Notre approche permet de garantir statiquement une sûreté stricte du code, sauf lorsqu'un objet est en cours de construction. Plus précisément :

- l'absence de déréférencement de références nulles est garantie lors de la manipulation d'objets construits ;
- l'accès à un attribut non-initialisé d'un objet en cours de construction provoque un échec précoce à l'exécution ;
- une construction déficiente (n'initialisant pas un attribut non-nullable) provoque un échec dès la fin de la construction.

3.1 Justification des règles dynamiques

La levée de l'exception E1 lors de l'accès à un attribut non-initialisé prévient la propagation de valeurs indéterminées dans le code et la déconstruction d'objets déjà construits.

Ainsi, dans l'exemple de la figure 6, si l'accès à `a.x` ne levait pas une exception, l'objet déjà construit `b` serait contaminé par la désinitialisation de son attribut `y`.

La levée de l'exception E2 lors de la détection d'un attribut non-initialisé, à la fin de la construction d'un objet, assure (en tandem avec la règle précédente) que tous les attributs non-nullables d'un objet construit sont toujours initialisés.

3.2 Zone de danger

La zone de danger est clairement circonscrite :

- les exceptions E1 et E2 ne peuvent être levées que pendant l'exécution d'un constructeur. Plus précisément, l'exception E1 ne peut être levée que si le constructeur a laissé s'échapper `this` (implicitement ou explicitement) vers une méthode invoquée, ou s'il accède à un attribut non-initialisé. L'exception E2 ne peut être levée qu'à la fin du constructeur ;
- une fois un objet construit, l'accès à ses attributs (nullables ou non) ne peut plus causer ces exceptions.

Notons que l'exception E1 peut survenir n'importe où dans le code source. C'est dynamiquement que la zone de danger est réduite par la nécessité d'avoir un constructeur sur la pile d'exécution pour qu'une exception soit levée. En d'autres mots, nos garanties statiques sont circonscrites par des limites dynamiques.

En pratique, la zone de danger est d'avantage réduite par les techniques de programmation modernes à un petit sous-ensemble du code exécuté, car les programmeurs sont déjà habitués de ne pas laisser, tant que possible, s'échapper un objet partiellement construit et la construction des objets ne constitue, habituellement, qu'une faible partie du code source.

```
class A {
 ...
 A(..., List<A> liste) {
  ...
  liste.add(this);
 }
 ...
}
```

a – *Structure passée en paramètre.*

```
class B {
 static List<B> instances = ...;
 ...
 B(...) {
  ...
  instances.add(this);
 }
 ...
}
```

b – *Structure statique.*

FIG. 7 – *Échappement d'une instance en construction vers une structure externe.*

3.3 Détection des bogues

Lorsqu'une exception E1 ou E2 est levée, le constructeur fautif est toujours dans la pile d'exécution et est facilement identifiable. Pour l'exception E1, le constructeur fautif est celui de l'objet dont l'attribut non-initialisé a été accédé ; pour l'exception E2, le constructeur fautif est celui en cours d'exécution au dessus de la pile d'exécution.

De plus, la détection des accès aux champs non-initialisés et la vérification systématique de l'initialisation de tous les champs à la fin de la construction permettent aux tests unitaires de détecter la majorité des cas de construction déficiente dès la première exécution d'un constructeur.

4 Préservation de l'expressivité

Il est simple de prouver que notre approche ne réduit pas l'expressivité d'un langage de programmation. Soient un langage de programmation L et un programme p écrit dans ce langage. On peut transformer p en un programme équivalent p' dans le langage L auquel on a ajouté `nullable`, les règles de typage statique reliées et les vérifications dynamiques, en procédant comme suit :

- on remplace toutes les déclaration de type t par des déclarations $n(t)$;
- on remplace tous les appels de méthodes `exp.m()` par `((notnull)exp).m()`[5] ;
- on remplace tous les accès aux attributs `exp.x` par `((notnull)exp).x`.

Une telle transformation simple est possible car notre approche ne restreint aucunement l'échappement des instances en construction. L'exemple de la figure 7.a montre l'échappement de l'instance en construction vers une structure externe passée en paramètre. Celui de la figure 7.b montre un échappement vers une structure statique. La figure 8 montre un autre cas d'échappement utilisé pour la création d'une structure circulaire. Dans tous ces cas, notre système garantit l'absence d'erreur de déréférencement une fois la construction de l'objet terminée.

Notons qu'en pratique, il est désirable de minimiser l'utilisation de `nullable` pour profiter des garanties de notre approche. Il est souvent possible de ne pas ajouter `nullable` sans

[5]La coercition ajoutée lève une exception `NullPointerException` lorsque l'expression est nulle, préservant la sémantique originale.

```
class Circ {
 Circ next;
 Circ() { next = new Circ(this); }
 private Circ(Circ next) { this.next = next; }
}
... = new Circ();
```

FIG. 8 – *Structure circulaire.*

perte d'expressivité. Toutefois, l'utilisation de `null` n'est pas proscrite, en général, dans notre approche ; elle est même parfois nécessaire. Par exemple, l'utilisation d'attributs `nullables` permet de réaliser la construction progressive d'une structure de données complexe.

5 Expérimentation

5.1 Zone de danger

Dans un premier temps, nous avons cherché à quantifier l'étendue de la zone de danger dans un programme réel. Le programme que nous avons considéré est `nitc`, le compilateur NIT[6]. NIT est une évolution du langage PRM[7] dans le but d'en faire un langage robuste à usage général. `nitc` est écrit en 57487 lignes de code NIT réparties sur 904 classes.

La zone de danger est principalement liée à l'exception E1, car l'exception E2 est levée à un endroit bien défini dans le code, soit à la fin des constructeurs. L'exception E1, quant à elle, peut être levée n'importe où dans le code lors de l'accès à un attribut, du moment qu'il y a un constructeur sur la pile. Dans les deux cas, c'est le constructeur qui est fautif.

Pour obtenir une borne supérieure grossière de l'étendue de la zone de danger, nous avons évalué la part des constructeurs qui accèdent *possiblement* (en lecture) à un attribut non-initialisé. Pour ce faire, nous avons implémenté une analyse statique simple qui approxime l'ensemble des constructeurs sûrs pour E1. Les constructeurs retenus :

- n'accèdent pas en lecture à un attribut de `this` ;
- ne passent pas `this` en paramètre lors de l'invocation de méthodes ;
- n'assignent pas `this` dans des variables ou des attributs ;
- n'appellent pas de constructeurs non-sûrs ;
- n'utilisent pas `this` comme receveur d'une méthode non-sûre.

En utilisant cette analyse[8] sur `nitc`, nous avons identifié 127 constructeurs non-sûrs sur un total de 1035 constructeurs, soit 12,27 %. L'ensemble des constructeurs constitue 3419 lignes d'un programme de 57487 lignes, soit 5,95 % du code source. Parmi ceux-ci, les non-sûrs constituent 1677 lignes, soit 2,92 % du code source.

Notons que l'analyse utilisée ci-dessus est très grossière. Par exemple, elle considère comme non-sûr un constructeur qui initialise tous ses attributs avant de passer `this` en paramètre

[6] `http://nitlanguage.org/`.

[7] PRM est un langage de programmation développé pour l'expérimentation de techniques de compilation séparée (Privat, 2006).

[8] Nous utilisons CHA (Dean et al., 1995) pour construire le graphe d'appel et approximer l'ensemble des méthodes et des constructeurs possiblement appelés à partir d'un site d'appel.

Composant	Lignes de code	Classes	Types explicites nullable/total (%)	Attributs nullable/total (%)
Bibliothèque standard NIT				
Collections	1835	34	24/374 (6,42 %)	14/35 (40,00 %)
Autre	2636	52	19/560 (2,14 %)	1/39 (2,56 %)
Total	4471	86	36/934 (3,85 %)	15/74 (20,27 %)
Compilateur `nitc`				
Parser/Lexer	48529	815	1369/10495 (13,04 %)	166/213 (77,93 %)
Autre	9622	89	94/1633 (5,91 %)	16/252 (6,50 %)

TAB. 1 – *Le compilateur* `nitc` *à la fin de la phase 1.*

d'une méthode. Effectivement, une vérification manuelle des constructeurs considérés non-sûrs nous a permis de constater que la plupart correspondent aux cas illustrés par la figure 7 où les constructeurs se contentent d'enregistrer l'objet dans une collection une fois celui-ci entièrement initialisé.

Les résultats de cette expérimentation nous permettent de confirmer les pratiques habituelles citées à la section 3.2 : les programmeurs sont habitués à contrôler l'échappement des objets en cours de construction et les constructeurs ne constituent qu'une faible part du code source.

5.2 Utilisation des nullables

Dans un second temps, afin d'évaluer notre approche, nous l'avons intégrée au langage NIT puis nous avons modifié la bibliothèque standard et le compilateur, qui sont écrits dans ce même langage. Avant les modifications nécessaires pour supporter les types nullables, `nitc` contenait près de 900 classes distribuées dans 27 fichiers, totalisant 56127 lignes de code. La bibliothèque standard de NIT contenait 86 classes distribuées dans 18 fichiers, pour un total de 4369 lignes de code.

Nous avons procédé en deux étapes lors de la modification de la bibliothèque et du compilateur : en premier lieu, nous avons ajouté `nullable` lorsque nécessaire et modifié le code de façon appropriée pour éliminer les erreurs de compilation ; ensuite, nous nous sommes servis des erreurs dynamiques précoces pour identifier les `nullables` manquants qui n'avaient pas été détectés statiquement.

Compilation statique

En utilisant un compilateur pour le langage NIT modifié[9], nous avons intégré les types nullables dans le code source de la bibliothèque et du compilateur et avons éliminé toutes les erreurs de compilation. Le tableau 1 présente les statistiques du code transformé.

Le nombre total de lignes de code augmente de 3,5 % et, globalement, 6 % des types explicites[10] sont nullables. Dans la bibliothèque, 20 % des attributs sont nullables. Ce taux est gonflé par les classes des collections où 40 % des attributs sont nullables. Le reste de la bibliothèque ne contient qu'un seul attribut nullable. De façon similaire, le taux d'attributs nullables est sensiblement plus élevé (78 %) dans les parties du compilateur (Lexer/Parser) qui

[9] Ce compilateur est écrit en « ancien » NIT.

[10] NIT supporte l'inférence de type légère ; les types explicites sont tous ceux écrits par le programmeur.

Composant	Lignes de code	Classes	Types explicites nullable/total (%)	Attributs nullable/total (%)
Bibliothèque standard NIT				
Collections	1860	34	31/374 (8,29 %)	18/35 (51,43 %)
Autre	2643	52	15/560 (2,68 %)	4/39 (10,26 %)
Total	4503	86	46/934 (4,92 %)	22/74 (29,73 %)
Compilateur `nitc`				
Parser/Lexer	48531	815	3171/10495 (13,06 %)	168/213 (78,87 %)
Autre	10343	89	179/1633 (10,96 %)	81/252 (32,14 %)

TAB. 2 – *Le compilateur `nitc` à la fin de la phase 2.*

ont été générées par SABLECC[11], ce taux s'expliquant par le fait que le code généré n'a pas été optimisé pour l'utilisation des types nullables. Dans le reste du compilateur, 6,5 % des attributs sont nullables.

Vérification dynamique

Une fois le code compilé, dès la première exécution, les validations dynamiques ont détecté des erreurs potentielles que nous avons dû corriger. Le tableau 2 présente les statistiques du code corrigé.

Les corrections ont requis 1 % de plus de lignes de code. Les changements notables sont l'augmentation de 1 à 4 attributs nullables dans la partie « Autre » de la bibliothèque, et celle de 6,5 % à 32 % d'attributs nullables dans la partie « Autre » du compilateur. Ces corrections sont toutes dues à des exceptions E2 correspondant à l'initialisation incomplète des objets. Dans le code, la valeur des attributs fautifs est calculée et stockée après la construction et elle n'est pas accédée avant ce calcul. Puisqu'en ancien NIT les attributs non-initialisés sont *implicitement* initialisés à `null`, la compilation statique n'a pas détecté d'erreur.

6 Travaux connexes

Chalin et James (2007) démontrent, à l'aide de mesures sur des applications d'envergure, que les propriétés et les variables sont plus souvent non-nullables que nullables et qu'il est donc plus approprié d'utiliser l'annotation `nullable` que `notnull`, tel que nous l'avons fait.

Fähndrich et Leino (2003) proposent un système de types statiques similaire à celui de la sous-section 2.1 auquel ils ajoutent le marqueur *raw* indiquant qu'un type peut contenir une valeur en cours de construction. Le marqueur peut être paramétré pour indiquer que l'objet est partiellement construit (*raw up-to S*). Pour garantir la sûreté statique complète du code, ce système complique passablement l'écriture des constructeurs : `this` est de type *raw* dans les constructeurs ; il en résulte de nombreuses restrictions lorsqu'il est passé en paramètre ou utilisé comme receveur. En particulier, lorsque `this` est le receveur, la méthode doit être marquée *raw*. Ceci est une entrave à l'évolution et à la maintenance du code, car un nouveau constructeur ajouté à une classe ne pourra pas faire appel à une méthode existante non-*raw* lorsque le receveur est `this`. Quoique les auteurs ne soulèvent pas ce problème, il nous paraît

[11] `http://sablecc.org` modifié pour générer du NIT.

clair que cette approche force l'écriture de plusieurs versions d'une même méthode pour réduire la perte d'information de type. Contrairement à notre approche, cette approche ne peut pas gérer les exemples de la figure 7, ni les structures circulaires exprimées avec des attributs non-nuls (voir figure 8).

Fähndrich et Xia (2007) proposent une amélioration aux types *raw* sous la forme des types *delayed*. Cette version augmente l'expressivité en permettant la construction de structures circulaires. Par contre, le système de types s'en trouve significativement plus complexe que celui des types *raw* ; donc, l'écriture des constructeurs devient encore plus ardue. L'approche souffre également des problèmes d'évolution et de duplication de code.

Hubert et al. (2008) proposent d'inférer automatiquement les annotations *non-nul* à l'aide d'analyses statiques globales. Il présentent des preuves de rectitude de leur approche et de relative équivalence avec l'approche de Fähndrich et Leino (2003). Dans leurs travaux, ils se concentrent principalement sur la validité théorique et ne proposent pas de solutions pour les programmes qui sont rejetés. Cette approche a l'avantage de ne pas imposer au programmeur l'usage d'annotations complexes de type lors de l'écriture des constructeurs ; toutefois, dans le cas du rejet d'un programme, le message d'erreur expose un problème de types inférés complexe.

Au fil des ans, il y a eu de nombreux travaux sur la détection automatique d'erreurs de déréférencement de références nulles (Hovemeyer et Pugh, 2007), ou même de variété d'erreurs (Dillig et al., 2007)[12]. Ces approches analysent le code des programmes pour identifier les erreurs potentielles. Ces approches ne sont pas exactes et rapportent des faux-positifs.

7 Conclusion

Dans cet article, nous avons présenté une approche qui permet la prévention du déréférencement de références nulles dans un langage à objets. Ce modèle de programmation comprend trois parties : un système de types statiques simple (mot clé `nullable`), des vérifications dynamiques et un opérateur de test dynamique (`isset`). L'originalité de ce modèle tient à sa politique d'échec précoce (*fail fast*), à la garantie que ces échecs ne peuvent survenir que dans une petite portion du code (limitée dynamiquement à la construction des instances) et à l'absence de perte d'expressivité.

Nous avons implémenté et validé notre proposition dans le langage NIT et montré que l'adaptation d'un programme déjà écrit est facilement réalisable (autour de 10% des types sont annotés `nullable`), que la politique d'échec précoce permet de détecter et de corriger rapidement de nombreux bogues potentiels et, enfin, que la zone de danger est petite en pratique.

Finalement, nous croyons que l'utilisation de notre approche aidera à augmenter la précision d'une grande variété d'analyses statiques connues grâce à ses garanties de robustesse des objets construits.

[12] Vu leur nombre considérable, nous n'avons cité que deux articles récents représentatifs.

Références

Chalin, P. et P. R. James (2007). Non-null references by default in Java : alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 4609 of *Lecture Notes in Computer Science*, pp. 227–247. Springer.

Dean, J., D. Grove, et C. Chambers (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, Volume 952/-1 of *Lecture Notes in Computer Science*, pp. 77–101. Springer.

Dillig, I., T. Dillig, et A. Aiken (2007). Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 435–445. ACM.

Fähndrich, M. et K. R. M. Leino (2003). Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)*, pp. 302–312. ACM.

Fähndrich, M. et S. Xia (2007). Establishing object invariants with delayed types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA)*, pp. 337–350. ACM.

Hovemeyer, D. et W. Pugh (2007). Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, pp. 9–14. ACM.

Hubert, L., T. P. Jensen, et D. Pichardie (2008). Semantic foundations and inference of non-null annotations. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Volume 5051 of *Lecture Notes in Computer Science*, pp. 132–149. Springer.

Privat, J. (2006). *De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc*. Thèse d'informatique, Université Montpellier II.

Summary

Dereferencing null references is a common programming error in object-oriented languages. To prevent it, some approaches statically guarantee its absence using type systems or annotations, but reduce the expressiveness of the language. Other approaches analyse the source code, instead, to identify potential errors, but they can find false-positives and they do not guarantee the absence of errors at run time. In this paper, we propose a different approach which offers absence of dereferencing errors static guarantees in big portions of the code. The approach consists of a simple static type system, dynamic checks, and a dynamic test operator. Our approach preserves the expressiveness of the language, limits the danger zone to instance construction, and allows the early detection of errors at execution time. Our experimental measurements demonstrate wide-reaching static guarantees on the absence of dereferencing errors and the efficiency of early error detection.