École de technologie supérieure

LOG792 Projet de fin d'étude
Rapport d'étape

Formalisation de systèmes de types à l'aide d'Isabelle/HOL

Auteur :
Martin Desharnais

Superviseur :
Dr. David Labbé

21 octobre 2014

Table des matières

1 Problématique et contexte

Ce projet s'intéresse à l'étude des systèmes de types, dans le contexte des langages de programmation, dont voici une définition [Pie02] :

> Un système de types est une méthode syntaxique tractable pour prouver l'absence de certains comportements des programmes par la classification des phrases selon le genre de valeurs qu'elles calculent.

L'objectif est donc de garantir, sans l'exécuter, qu'un programme est exempt de certaines erreurs telles qu'une faute typographique dans un nom de variable, l'appel d'une fonction non supportée dans un certain contexte ou encore une tentative de diviser un nombre par une chaine de caractères. De tels exemples peuvent sembler simplistes, mais sont très fréquents et peuvent avoir des conséquences désastreuses : une incohérence informatique entre les systèmes d'unités métrique et anglo-saxon a provoqué la destruction du Mars Climate Orbiter en 1999. Bien sûr, tous les défauts ne peuvent pas être décelés par un système de types. Cependant, il en existe un très grand nombre, de divers niveaux d'expressivité et de complexité, qui permettent de détecter un éventail varié d'erreurs.

Lors du développement d'un nouveau système de type, un ensemble de preuves est réalisé afin de démontrer que celui-ci respecte ses objectifs. L'étude de ces systèmes ainsi que des preuves qui les accompagnent est le sujet du présent projet.

2 Objectifs du projet

Les objectifs de ce projet sont quadruples : 1) S'initier à la formalisation avec l'assistant de preuve Isabelle/HOL ; 2) S'initier à la théorie des types ; 3) Valider les preuves manuelles existantes ; 4) Clarifier les cas limites de ces dernières.

2.1 S'initier à la formalisation Isabelle/HOL

Il existe deux grandes catégories d'outils pour effectuer une formalisation : les prouveurs automatiques et les assistants de preuve interactifs. Quel que soit l'outil utilisé, il faut définir un contexte de travail et une équation que l'on veut prouver. La différence est que, dans le premier cas, l'outil tente de trouver une preuve entièrement automatiquement alors que, dans le second cas, il faut travailler interactivement avec l'outil pour prouver le théorème.

Isabelle/HOL est un assistant de preuve interactif utilisant la logique d'ordre supérieure. Il permet de spécifier des formules mathématiques, algorithmes et objets dans un langage déclaratif, fonctionnel et typé. Il est alors possible de spécifier des propriétés sur l'interaction entre les divers éléments. Une fois le système

formalisé, il est possible d'en extraire du code exécutable correspondant aux spécifications. Ce projet est l'occasion de s'initier à la définition d'un système formel et aux preuves interactives à l'aide d'Isabelle/HOL.

2.2 S'initier à la théorie des types

Plusieurs des langages de programmation dominant actuellement ont une étape de validation des types appliquée à la compilation. La majorité des programmeurs sont donc familiarisés avec le concept. Malheureusement, les systèmes de types présents dans ces langages sont généralement simples, imposent nombre de contraintes à leurs utilisateurs et n'offrent qu'un nombre limité de garanties en retour. Certaines de ces limitations ont été mitigées dans de nouvelles versions du langage (e.g. l'ajout des types et fonctions génériques en Java).

Cependant, des options plus expressives et plus puissantes sont bien connues, ou bien actuellement en développement par les acteurs du milieu. Ce projet est l'occasion de consolider les acquis fondamentaux et d'en apprendre plus sur les concepts plus avancés de la théorie des types.

2.3 Valider les preuves manuelles existantes

La théorie des types étant un sujet de recherche très actif depuis plusieurs dizaines d'années, un grand nombre de publications décrivent les caractéristiques de différents systèmes de type. Cependant, une preuve manuelle étant validée par des êtres humains, il est toujours possible que des erreurs s'y soient glissées. La formalisation de celles-ci à l'aide d'un assistant de preuve permet de valider, sous réserve que l'assistant de preuve soit correct, qu'aucune erreur logique n'est présente. S'il s'avérait qu'une erreur soit découverte dans le cadre de ce projet, l'information serait transmise à l'auteur initial afin de l'informer de la situation.

2.4 Clarifier les cas limites des preuves manuelles

Les propriétés énoncées et prouvées manuellement semblent souvent évidentes dès lors qu'elles sont appliquées à un exemple concret. Cette méthode de visualisation a toutefois ses limites puisque certaines constructions plus complexes peuvent entrainer des résultats inattendus. La formalisation de ces propriétés à l'aide d'un assistant de preuve oblige son auteur à considérer la liste exhaustive des constructions du langage et permet ainsi d'acquérir une meilleure compréhension de la propriété et des cas limites. Cette technique fut utilisée avec succès afin d'enseigner l'ingénierie logicielle à des étudiants de premier cycle [PEF08, Pie09, Nip12].

3 Méthodologie

L'ouvrage de référence de ce projet est le livre *Types and Programming Languages* de Benjamin C. Pierce. Ce dernier est composé de six sections : les systèmes non typés, les types simples, le sous-typage, les types récursifs, le polymorphisme et les systèmes d'ordre supérieur. Chaque section est composée de plusieurs chapitres décrivant un système de type bonifiant un système décrit précédemment en lui adjoignant un concept supplémentaire. Les figures 1 et 2 présentent les chapitres des deux premières sections sur lesquelles se concentre ce projet — ceux en gras sont ceux qui seront formalisés — et la figure 3 présente les dépendances entre ceux-ci.

Ces chapitres furent choisis, car ils décrivent des langages et leurs théorèmes au lieu d'en expliquer la théorie ou d'en présenter une implémentation. De plus, ils culminent au λ-calcul simplement typé, qui a la propriété de pouvoir représenter la majorité des langages de programmation existants [1].

§ 3 **Expressions arithmétiques non typées**
§ 4 Une implémentation en ML des expressions arithmétiques
§ 5 **Le lambda-calcul non typé**
§ 6 Représentation non nommée des termes
§ 7 Une implémentation en ML du lambda-calcul

Figure 1: Section I du livre de référence — Les systèmes non typés

§ 8 **Expressions arithmétiques typées**
§ 9 **Le lambda-calcul simplement typé**
§ 10 Une implémentation en ML des types simples
§ 11 Extensions simples
§ 12 Normalisation
§ 13 Références
§ 14 Exceptions

Figure 2: Section II du livre de référence — Les types simples

1. Ceci n'est pas tout à fait exact puisqu'il faudrait y ajouter la fonctionnalité de pouvoir communiquer avec l'environnement d'exécution afin de pouvoir faire des opérations telles que lire ou écrire dans un fichier, effectuer de la communication interprocessus, etc.

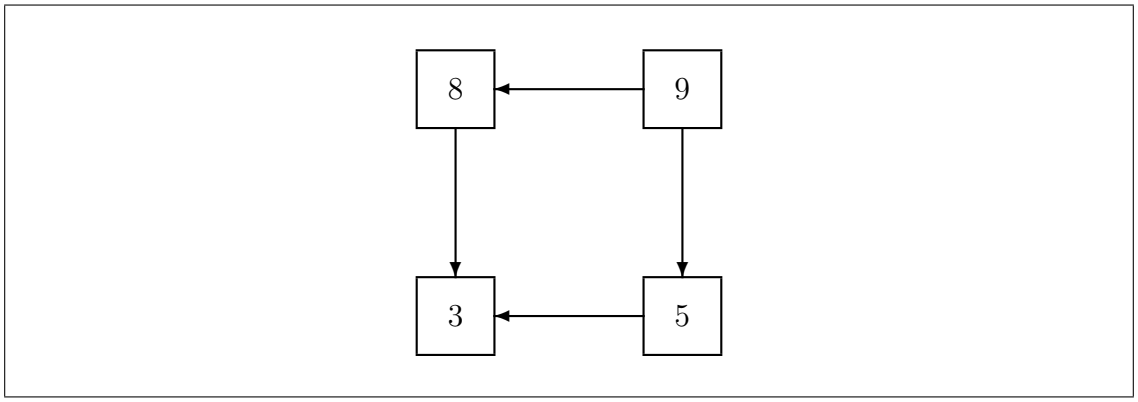

Figure 3: Dépendances entre les chapitres

3.1 Formalisation

Le projet consiste à formaliser séquentiellement les différents chapitres en se basant sur le travail des chapitres précédents. Pour cette raison, le premier chapitre fut le plus long à formaliser : tout devant être fait à partir de zéro. Pour les chapitres suivants, la première étape est de copier la formalisation des dépendances et de la modifier pour inclure les nouveaux concepts introduits.

Chacune des formalisations se fait en quatre étapes :

1. Lecture attentive du chapitre et compréhension générale des concepts en jeu ;
2. Définition dans Isabelle/HOL des structures nécessaires à la formalisation ;
3. Preuve des différents lemmes et théorèmes ainsi que, optionnellement, des exercices ;
4. Simplification des définitions et preuves.

L'objectif principal du projet étant de formaliser le chapitre 9 et ce genre de formalisation étant un projet ambitieux, un certain nombre d'actions pourraient être entreprises si le respect des échéanciers s'avère menacé :

1. Ne pas prendre en compte les exercices proposés ;
2. Définir certains lemmes comme des axiomes au lieu de les prouver ;
3. Sauter les chapitres 5 ou 8 afin de passer directement au chapitre 9.

3.2 Techniques et outils

Isabelle Système générique pour l'implémentation de formalismes logiques.

Isabelle/HOL Spécialisation d'Isabelle pour la logique d'ordre supérieur (*Higher-Order Logic* en anglais).

Isabelle/Isar Langage structuré permettant d'écrire des preuves plus lisibles.

Isabelle/jEdit Environement de développement intégré pour Isabelle basé sur l'éditeur jEdit.

Sledgehammer Outil appliquant des prouveurs automatiques de théorèmes ainsi que des solveurs de « satisfaisabilité modulo théories ».

4 Sommaire des travaux réalisés et recommandations

4.1 Sommaire des traveaux réalisés

Au cours de la première partie de ce projet, le travail d'apprentissage et la formalisation des deux premiers chapitres du livre de référence ont été complétés. La liste exhaustive des activitées réalisés se trouve dans le plan de travail de l'annexe A.

4.1.1 Apprentissage de la formalisation avec Isabelle/HOL

La réalisation la plus importante fut certainement d'apprendre les bases du processus de formalisation et du fonctionnement de l'assistant de preuve.

Lors d'une formalisation, la décision la plus importante à prendre est celle de la représentation des concepts manipulés. Dans le case présent, un type inductif est utilisé pour représenter la synthaxe abstraite du langage, des fonctions à récursion primitive pour certaines manipulations mineures et des définitions inductives pour les relations d'évaluation. En Isabelle/HOL ces définitions sont introduites respectivement avec les commandes `datatype`, `primrec` et `inductive`.

Une fois les définitions de base mises en place, prouver les différents théorèmes nécessita d'apprendre la notation Isabelle/Isar qui est utilisé pour écrire des preuves lisibles à la fois par l'ordinateur et l'utilisateur.

4.1.2 Expressions arithmétiques non-typés

Ce chapitre contient deux langages : le premier permet de représenter des expressions booléennes (i.e. vrai, faux et conditions) et le second, en plus des expressions booléennes, des expressions numériques (i.e. zéro, incrémenter, décrémenter et tester l'égalité avec zéro). Dans les deux cas, la formalisation consiste à définir le langage, les règles d'évaluations pour, ensuite, prouver certains théorèmes de la relation d'évaluation tels que la transitivité, l'univalence, la terminaison, etc. Elle peut être consultée à l'annexe B.

4.1.3 Le lambda-calcul non typé

Ce chapitre contient la définition minimale du λ-calcul : variable, abstraction de fonction et application de fonction. La formalisation prend la même forme que pour les expressions arithmétiques non typées et, lorsque possible, les mêmes théorèmes sont prouvés. Cependant, certaines propriétés qui étaient vraies pour les autres langages à l'étude ne le sont pas pour le λ-calcul simplement typé (e.g. la terminaison de l'évaluation). Dans ces situations, une preuve que la propriété ne tient pas été développée. Elle peut être consultée à l'annexe C.

4.2 Recommandations

Au vu du temps et du travail qui fut nécessaire pour formaliser les deux premiers chapitres, les objectifs du projet sont raisonnables et ne nécessitent pas d'être modifiés ; la formalisation des deux derniers chapitres devrait ressembler aux précédents. Il est recommandable que la majorité des ressources soit investie dans la rédaction du rapport et l'intégration des formalisations à celui-ci.

5 Livrables et planification

5.1 Description des artéfacts

Fiche de renseignement Formulaire fournissant le titre du projet, un cours résumé ainsi que les noms des étudiants impliqués et du professeur superviseur.

Proposition de projet Document présentant la problématique du projet, les objectifs, la méthodologie, les livrables, le plan de travail, les risques ainsi que les techniques et outils utilisés.

Rapport d'étape Document présentant une version étoffée et mise à jour de la proposition de projet, ainsi qu'une version partielle du rapport technique.

Diapositives de la présentation orale Diapositives utilisées pour la présentation orale finale du projet.

Rapport final Document présentant la problématique du projet, les objectifs, la méthodologie employée, les hypothèses, les résultats, l'analyse des résultats, les conclusions, les recommandations et les références.

Théories Isabelle/HOL Fichiers sources utilisés par l'assistant de preuve Isabelle/HOL pour sauvegarder les définitions et théorèmes formalisés au cours de ce projet.

5.2 Planification

Le plan de projet se trouve à l'annexe A. Par rapport à la proposition de projet, les modifications suivantes ont été apportées :

— Les efforts actuels, en heures, ont été ajoutés ;

— Les tâches complétées ont été identifiées ;

— L'échéancer des tâches 3.3.*, 3.4.* et 4 a été repoussé afin de prendre en considération que le projet ne pourra pas progresser du 25 octobre au 2 novembre.

6 Risques

Un certain nombre de risques sont identifiés dans le tableau 1. Par rapport à la proposition de projet, les modifications suivantes ont été apportées :

— Ajout d'une évaluation subjective des impacts ;

— Ajout d'une explication pour les probabilités ;

— Diminution de la probabilité du risque 2 de « Forte » à « Moyenne » ;

— Diminution de la probabilité du risque 3 de « Forte » à « Moyenne ».

Risque	**Impact**	**Probabilité**	**Mitigation / atténuation**
Manque d'expérience avec la théorie des types	Faible : Monopolise du temps pour apprendre la théorie.	Faible : Le λ-calcul simplement typé est le plus simple de sa catégorie.	Étudier attentivement l'ouvrage de référence.
Manque d'expérience avec l'assistant de preuve Isabelle/HOL	Moyen : Monopolise du temps pour apprendre le fonctionnement de l'outil.	Moyenne : La majorité de l'apprentissage a été effectuée lors de la formalisation du premier chapitre.	Étudier attentivement et se référer au besoin à la documentation de l'outil.
Manque d'expérience en formalisation	Fort : Monopolise du temps pour apprendre la méthodologie.	Moyenne : La majorité de l'apprentissage a été effectuée lors de la formalisation du premier chapitre.	S'informer auprès des chercheurs expérimentés de la chaire de recherche.
Objectifs trop ambitieux	Fort : Ne pas réaliser toutes les formalisations prévues.	Moyenne : La formalisation des chapitres progresse selon l'échéancier prévu.	Un certain nombre de mitigations sont décrites à la section 3.1.

TABLE 1 – Risques et mitigations

7 Références

[Bla14] Jasmin C. Blanchette. *Hammering Away : A User's Guide to Sledgehammer for Isabele/HOL*, 2014.

[Nip12] Tobias Nipkow. Teaching semantics with a proof assistant : No more lsd trip proofs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, 2012.

[Nip14] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*, 2014.

[NPW14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*, 2014.

[PEF08] Rex Page, Carl Eastlund, and Matthias Felleisen. Functional programming and theorem proving for undergraduates : A progress report. In *Functional and Declarative Programming in Education (FDPE08)*, 2008.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie09] Benjamin C. Pierce. Lambda, the ultimate TA : Using a proof assistant to teach programming language foundations, September 2009. Keynote address at *International Conference on Functional Programming (ICFP)*.

8 Table des matières du rapport

Sur recommandation du directeur du département de génie logiciel et des TI, monsieur Jean-Marc Robert, le doyen des études, monsieur Pierre Bourque, a autorisé que la rédaction du rapport soit effectuée en anglais.

This section discusses a possible thesis outline. It consists of seven main sections: three to introduce the reader to the formalization of type systems and four containing the proper formlizations.

I Introduction

1 Introduction to Type Systems
This section will provide a brief introduction to type systems, their history, current uses and possible future directions.

2 Introduction to Formalization
This section will provide a brief introduction to logic, formal reasoning and formalization.

3 Introduction to Isabelle/HOL
This section will provide a brief introduction to Isabelle/HOL: a proof assistant based on higher-order logic.

II Untyped Systems

4 Untyped Arithmetic Expression

This section will present the formalization of the language introduced in the Chapter 3 of the reference book. It will consist of the definitions and theorems defining the formal system.

5 Untyped Lambda Calculus

This section will present the formalization of the language introduced in the Chapter 5 of the reference book. It will consist of the definitions and theorems defining the formal system. It will also feature a brief introduction to the λ-calculus and its relation to programming language theory.

III Simple Types

6 Typed Arithmetic Expressions

This section will present the formalization of the language introduced in the Chapter 8 of the reference book. It will consist of the definitions and theorems defining the formal system.

7 Simply Typed Lambda Calulus

This section will present the formalization of the language introduced in the Chapter 9 of the reference book. It will consist of the definitions and theorems defining the formal system.

IV conclusion

Annexe A Plan de travail

#	**Commence**	**Termine**	**Efforts estimés (heures)**	**Efforts actuels (heures)**	**Complété**	**Jalon**	**Artéfacts**
1		2014-09-02	1	1	✓	Remise de la fiche de renseignement	Fiche de renseignements
1.1		2014-09-01	1	1	✓	Rencontre — professeur superviseur	
2	2014-09-02	2014-09-19	5	5	✓	Remise de la proposition de projet	Proposition de projet
2.1		courriels	1	1	✓	Rencontre — professeur superviseur	
2.2.1	2014-09-19		2	2	✓	Étude du chapitre 3	
2.2.2			10	8	✓	Définitions du chapitre 3	
2.2.3			15	25	✓	Preuves du chapitre 3	
2.2.4		2014-10-10	2	2	✓	Simplification du chapitre 3	Théorie Isabelle/HOL
3	2014-09-19	2014-10-24	10	10	✓	Remise du rapport d'étape	Rapport d'étape
3.1		à déterminer	1			Rencontre — professeur superviseur	
3.2.1	2014-10-10		2	2	✓	Étude du chapitre 5	
3.2.2			5	2	✓	Définitions du chapitre 5	
3.2.3			10	10	✓	Preuves du chapitre 5	
3.2.4		2014-10-24	3	2	✓	Simplification du chapitre 5	Théorie Isabelle/HOL
3.3.1	**2014-11-03**		2	2	✓	Étude du chapitre 8	
3.3.2			5			Définitions du chapitre 8	
3.3.3			10			Preuves du chapitre 8	
3.3.4		**2014-11-10**	3			Simplification du chapitre 8	Théorie Isabelle/HOL
3.4.1	**2014-11-10**		2			Étude du chapitre 9	
3.4.2			5			Définitions du chapitre 9	
3.4.3			15			Preuves du chapitre 9	
3.4.4		**2014-11-21**	3			Simplification du chapitre 9	Théorie Isabelle/HOL
4	**2014-11-25**	2014-12	10			Présentation	Diapositives de la présentation orale
5	2014-12-01	2014-12	30			Remise du travail	Rapport final
5.1		à déterminer	1			Rencontre — professeur superviseur	

Annexe B Expressions arithmétiques non typées

theory *Untyped-Arithmetic-Expressions*
imports *Main*
begin

datatype *B-term*
  = *BTrue*
  | *BFalse*
  | *BIf B-term B-term B-term*

primrec *consts-B* :: *B-term* ⇒ *B-term set* **where**
  consts-B BTrue = {*BTrue*} |
  consts-B BFalse = {*BFalse*} |
  consts-B (*BIf t1 t2 t3*) = *consts-B t1* ∪ *consts-B t2* ∪ *consts-B t3*

primrec *size-B* :: *B-term* ⇒ *nat* **where**
  size-B BTrue = *1* |
  size-B BFalse = *1* |
  size-B (*BIf t1 t2 t3*) = *1* + *size-B t1* + *size-B t2* + *size-B t3*

primrec *depth-B* :: *B-term* ⇒ *nat* **where**
  depth-B BTrue = *1* |
  depth-B BFalse = *1* |
  depth-B (*BIf t1 t2 t3*) = *1* + *max* (*depth-B t1*) (*max* (*depth-B t2*) (*depth-B t3*))

lemma *card-union-leq-sum-card*: *card* (*A* ∪ *B*) ≤ *card A* + *card B*
  by (*cases finite A* ∧ *finite B*) (*simp only*: *card-Un-Int*, *auto*)

lemma *card* (*consts-B t*) ≤ *size-B t*
proof (*induction t*)
  case *BTrue*
  show *?case* **by** *simp*
next
  case *BFalse*
  show *?case* **by** *simp*
next
  case (*BIf t1 t2 t3*)
  show *?case*
  proof −
    let *?t1* = *consts-B t1*
    let *?t2* = *consts-B t2*
    let *?t3* = *consts-B t3*

```
    have card (?t1 ∪ ?t2 ∪ ?t3) ≤ card ?t1 + card ?t2 + card ?t3
    by (smt card-union-leq-sum-card add-le-imp-le-right le-antisym le-trans nat-le-linear)
    also have ... ≤ size-B t1 + size-B t2 + size-B t3
      using BIf.IH by simp
    finally show ?thesis by simp
  qed
qed



inductive is-value :: B-term ⇒ bool where
  is-value-BTrue: is-value BTrue |
  is-value-BFalse: is-value BFalse

inductive-cases is-value-BIfD: is-value (BIf t1 t2 t3)

inductive eval-once :: B-term ⇒ B-term ⇒ bool where
  e-if-true: eval-once (BIf BTrue t2 t3) t2 |
  e-if-false: eval-once (BIf BFalse t2 t3) t3 |
  e-if: eval-once t1 t1' ⟹ eval-once (BIf t1 t2 t3) (BIf t1' t2 t3)

inductive-cases eval-once-BTrueD: eval-once BTrue t
inductive-cases eval-once-BFalseD: eval-once BFalse t

inductive eval :: B-term ⇒ B-term ⇒ bool where
  e-once: eval-once t t' ⟹ eval t t' |
  e-self: eval t t |
  e-transitive: eval t t' ⟹ eval t' t'' ⟹ eval t t''

inductive eval' :: B-term ⇒ B-term ⇒ bool where
  e-base': eval' t t |
  e-step': eval-once t t' ⟹ eval' t' t'' ⟹ eval' t t''

lemma e-once': eval-once t t' ⟹ eval' t t'
  by (simp add: e-base' e-step')

lemma e-transitive': eval' t t' ⟹ eval' t' t'' ⟹ eval' t t''
proof (induction t t' arbitrary: t'' rule: eval'.induct)
  case (e-base' t t'') thus ?case .
next
  case (e-step' t t' t'' t''')
  thus ?case using eval'.e-step' by blast
qed



lemma eval-eq-eval': eval = eval'
  apply (rule ext)+
  apply (rule iffI)
```

```
   apply (rename-tac t t')
   apply (erule eval.induct)
     apply (erule e-once')
    apply (rule e-base')
   apply (erule e-transitive')
   apply assumption

  apply (erule eval'.induct)
   apply (rule e-self)
  using e-once e-transitive by blast

definition is-normal-form :: B-term ⇒ bool where
  is-normal-form t ⟷ (∀ t'. ¬ eval-once t t')




lemma
  assumes
    s: s = BIf BTrue BFalse BFalse and
    t: t = BIf s BTrue BTrue and
    u: u = BIf BFalse BTrue BTrue
  shows eval-once (BIf t BFalse BFalse) (BIf u BFalse BFalse)
proof −
  have eval-once s BFalse unfolding s by (rule e-if-true)
  hence eval-once t u unfolding t u by (rule e-if)
  thus ?thesis by (rule e-if)
qed




theorem eval-single-determinacy:
  fixes t t' t'' :: B-term
  shows eval-once t t' ⟹ eval-once t t'' ⟹ t' = t''
proof (induction t t' arbitrary: t'' rule: eval-once.induct)
  case (e-if-true t1 t2)
  thus ?case by (auto elim: eval-once.cases)
next
  case (e-if-false t1 t2)
  thus ?case by (auto elim: eval-once.cases)
next
  case (e-if t1 t1' t2 t3)
  show ?case
    apply (rule eval-once.cases[OF e-if.prems])
    using e-if.hyps by (auto dest: eval-once-BTrueD eval-once-BFalseD e-if.IH)
qed
```

theorem *value-imp-normal-form*:
  fixes *t* :: *B-term*
  shows *is-value t* ⟹ *is-normal-form t*
by (*auto simp*: *is-normal-form-def elim*: *is-value.cases dest*: *eval-once-BTrueD eval-once-BFalseD*)

theorem *normal-form-imp-value*:
  fixes *t* :: *B-term*
  shows *is-normal-form t* ⟹ *is-value t*
proof (*rule ccontr*, *induction t rule*: *B-term.induct*)
  case *BTrue*
  thus *?case* **by** (*simp add*: *is-value-BTrue*)
next
  case *BFalse*
  thus *?case* **by** (*simp add*: *is-value-BFalse*)
next
  case (*BIf t1 t2 t3*)
  thus *?case* **by** (*metis e-if e-if-false e-if-true is-normal-form-def is-value.cases*)
qed

corollary *uniqueness-of-normal-form*:
  fixes *t u u′* :: *B-term*
  assumes
    eval t u **and**
    eval t u′ **and**
    is-normal-form u **and**
    is-normal-form u′
  shows *u* = *u′*
using *assms*
unfolding *eval-eq-eval′*
proof (*induction t u rule*: *eval′.induct*)
  case (*e-base′ t*)
  thus *?case* **by** (*metis eval′.simps is-normal-form-def*)
next
  case (*e-step′ t t′ t″*)
  thus *?case* **by** (*metis eval′.cases is-normal-form-def eval-single-determinacy*)
qed

lemma *eval-once-size-B*:

```
  assumes eval-once t t'
  shows size-B t > size-B t'
using assms
proof (induction rule: eval-once.induct)
  case e-if-true
  thus ?case by simp
next
  case e-if-false
  thus ?case by simp
next
  case e-if
  thus ?case by simp
qed
```

```
theorem eval-always-terminate:
  ∃ t'. eval t t' ∧ is-normal-form t'
unfolding eval-eq-eval'
proof (induction rule: measure-induct-rule[of size-B])
  case (less t)
  show ?case
    apply (cases is-normal-form t)
    using e-base' apply blast
    using e-step' is-normal-form-def eval-once-size-B less.IH by blast
qed
```

```
datatype NBTerm
  = NBTrue
  | NBFalse
  | NBIf NBTerm NBTerm NBTerm
  | NBZero
  | NBSucc NBTerm
  | NBPred NBTerm
  | NBIs-zero NBTerm

primrec size-NB :: NBTerm ⇒ nat where
  size-NB NBTrue = 1 |
  size-NB NBFalse = 1 |
  size-NB NBZero = 1 |
  size-NB (NBSucc t) = 1 + size-NB t |
  size-NB (NBPred t) = 1 + size-NB t |
  size-NB (NBIs-zero t) = 1 + size-NB t |
  size-NB (NBIf t1 t2 t3) = 1 + size-NB t1 + size-NB t2 + size-NB t3
```

inductive *is-numeric-value-NB* :: *NBTerm* ⇒ *bool* **where**
is-numeric-value-NBZero: *is-numeric-value-NB NBZero* |
is-numeric-value-NBSucc: *is-numeric-value-NB nv* ⟹ *is-numeric-value-NB* (*NBSucc nv*)

inductive *is-value-NB* :: *NBTerm* ⇒ *bool* **where**
is-value-NBTrue: *is-value-NB NBTrue* |
is-value-NBFalse: *is-value-NB NBFalse* |
is-value-NB-numeric-value: *is-numeric-value-NB nv* ⟹ *is-value-NB nv*

inductive *eval-once-NB* :: *NBTerm* ⇒ *NBTerm* ⇒ *bool* **where**
eval-once-NBIf-NBTrue: *eval-once-NB* (*NBIf NBTrue t2 t3*) *t2* |
eval-once-NBIf-NBFalse: *eval-once-NB* (*NBIf NBFalse t2 t3*) *t3* |
eval-once-NBIf: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBIf t1 t2 t3*) (*NBIf t1' t2 t3*) |
eval-once-NBSucc: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBSucc t1*) (*NBSucc t1'*) |
eval-once-NBPred-NBZero: *eval-once-NB* (*NBPred NBZero*) *NBZero* |
eval-once-NBPred-NBSucc: *is-numeric-value-NB nv1* ⟹ *eval-once-NB* (*NBPred* (*NBSucc nv1*)) *nv1* |
eval-once-NBPred: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBPred t1*) (*NBPred t1'*) |
eval-once-NBIs-zero-NBZero: *eval-once-NB* (*NBIs-zero NBZero*) *NBTrue* |
eval-once-NBIs-zero-NBSucc: *is-numeric-value-NB nv1* ⟹ *eval-once-NB* (*NBIs-zero* (*NBSucc nv1*)) *NBFalse* |
eval-once-NBIs-zero: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBIs-zero t1*) (*NBIs-zero t1'*)

inductive *eval-NB* :: *NBTerm* ⇒ *NBTerm* ⇒ *bool* **where**
eval-NB-base: *eval-NB t t* |
eval-NB-step: *eval-once-NB t t'* ⟹ *eval-NB t' t''* ⟹ *eval-NB t t''*

definition *is-normal-form-NB* :: *NBTerm* ⇒ *bool* **where**
is-normal-form-NB t ⟷ (∀ *t'*. ¬ *eval-once-NB t t'*)

lemma *eval-once-NB-impl-eval-NB*: *eval-once-NB t t'* ⟹ *eval-NB t t'*
by (*simp add*: *eval-NB-step eval-NB-base*)

lemma *eval-NB-transitive*: *eval-NB t t'* ⟹ *eval-NB t' t''* ⟹ *eval-NB t t''*
proof (*induction t t' arbitrary*: *t'' rule*: *eval-NB.induct*)
case (*eval-NB-base t t'*)
thus *?case* **.**
next
case (*eval-NB-step t1 t2 t3*)
thus *?case* **using** *eval-NB.eval-NB-step* **by** *blast*
qed

```
inductive-cases eval-once-NBTrueD: eval-once-NB NBTrue t
inductive-cases eval-once-NBFalseD: eval-once-NB NBFalse t
inductive-cases eval-once-NBZeroD: eval-once-NB NBZero t

lemma not-eval-once-numeric-value: is-numeric-value-NB nv ⟹ eval-once-NB
nv t ⟹ P
proof (induction nv arbitrary: t rule: is-numeric-value-NB.induct)
  case is-numeric-value-NBZero
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case is-numeric-value-NBSucc
  show ?case
    by (auto
      intro: is-numeric-value-NBSucc.prems[THEN eval-once-NB.cases]
      elim: is-numeric-value-NBSucc.IH)
qed



theorem eval-once-NB-right-unique:
  fixes t t' t'' :: NBTerm
  shows eval-once-NB t t' ⟹ eval-once-NB t t'' ⟹ t' = t''
proof (induction t t' arbitrary: t'' rule: eval-once-NB.induct)
  case (eval-once-NBIf-NBTrue t1 t2)
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case (eval-once-NBIf-NBFalse t1 t2)
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case (eval-once-NBIf t1 t1' t2 t3)
  from eval-once-NBIf.prems eval-once-NBIf.hyps show ?case
    by (auto
      intro: eval-once-NB.cases
      dest: eval-once-NBTrueD eval-once-NBFalseD eval-once-NBIf.IH)
next
  case (eval-once-NBSucc t1 t2)
  from eval-once-NBSucc.prems eval-once-NBSucc.IH show ?case
    by (auto elim: eval-once-NB.cases)
next
  case eval-once-NBPred-NBZero
  thus ?case by (auto intro: eval-once-NB.cases dest: eval-once-NBZeroD)
next
  case (eval-once-NBPred-NBSucc nv1)
  show ?case
    apply (rule eval-once-NBPred-NBSucc.prems[THEN eval-once-NB.cases])
    using eval-once-NBPred-NBSucc.hyps
    by (auto elim: is-numeric-value-NBSucc not-eval-once-numeric-value[rotated])
next
```

```
  case (eval-once-NBPred t1 t2)
  from eval-once-NBPred.hyps eval-once-NBPred.prems show ?case
    using  is-numeric-value-NBZero
    by (auto
      intro: eval-once-NBPred.IH
      elim: eval-once-NB.cases
      dest: not-eval-once-numeric-value is-numeric-value-NBSucc)
next
  case eval-once-NBIs-zero-NBZero
  thus ?case
    by (auto intro: eval-once-NB.cases dest: eval-once-NBZeroD)
next
  case (eval-once-NBIs-zero-NBSucc nv)
  thus ?case
   by (auto intro: eval-once-NB.cases not-eval-once-numeric-value dest: is-numeric-value-NBSucc)
next
  case (eval-once-NBIs-zero t1 t2)
  show ?case
    apply (rule eval-once-NBIs-zero.prems[THEN eval-once-NB.cases])
    using eval-once-NBIs-zero.hyps
    by (auto
      intro: eval-once-NBZeroD eval-once-NBIs-zero.IH
      elim: not-eval-once-numeric-value[rotated] is-numeric-value-NBSucc)
qed
```

```
theorem value-imp-normal-form-NB:
  is-value-NB t ⟹ is-normal-form-NB t
  by (auto
    simp: is-normal-form-NB-def
    elim: is-value-NB.cases
    dest: eval-once-NBFalseD eval-once-NBTrueD not-eval-once-numeric-value)
```

```
theorem not-normal-form-imp-value-NB: ∃ t. is-normal-form-NB t ∧ ¬ is-value-NB
t (is ∃ t. ?P t)
proof
  have a: is-normal-form-NB (NBSucc NBTrue)
    by (auto elim: eval-once-NB.cases simp: is-normal-form-NB-def)
  have b: ¬ is-value-NB (NBSucc NBTrue)
    by (auto elim: is-numeric-value-NB.cases simp: is-value-NB.simps)
  from a b show ?P (NBSucc NBTrue) by simp
qed
```

corollary *uniqueness-of-normal-form-NB*:
  assumes
    eval-NB t u **and**
    eval-NB t u' **and**
    is-normal-form-NB u **and**
    is-normal-form-NB u'
  shows $u = u'$
using *assms*
proof (*induction t u rule*: *eval-NB.induct*)
  case (*eval-NB-base t*)
  thus *?case* **by** (*metis eval-NB.simps is-normal-form-NB-def*)
next
  case (*eval-NB-step t1 t2 t3*)
 thus *?case* **by** (*metis eval-NB.cases is-normal-form-NB-def eval-once-NB-right-unique*)
qed

lemma *eval-once-size-NB*:
  eval-once-NB t t' $\Longrightarrow$ *size-NB t* $>$ *size-NB t'*
by (*induction rule*: *eval-once-NB.induct*) *auto*

theorem *eval-NB-always-terminate*:
  $\exists\, t'$. *eval-NB t t'* $\wedge$ *is-normal-form-NB t'*
proof (*induction rule*: *measure-induct-rule*[*of size-NB*])
  case (*less t*)
  show *?case*
    apply (*case-tac is-normal-form-NB t*)
    using *eval-NB-base* **apply** *blast*
    using *eval-NB-step eval-once-size-NB is-normal-form-NB-def less.IH* **by** *blast*
qed

Annexe C Lambda-calcul non typées

theory *Untyped-Lambda-Calculus*
imports *Complex-Main*
begin

datatype *Term*
= *Var nat*
| *Abs Term*
| *App Term Term*

Definition 6.1.2 — n_terms

inductive *n-term* :: *nat* ⇒ *Term* ⇒ *bool* **where**
n-term-Var: *0* ≤ *k* ⟹ *k* < *n* ⟹ *n-term n* (*Var k*) |
n-term-Abs: *n-term n t* ⟹ *n* > *0* ⟹ *n-term* (*n* − *1*) (*Abs t*) |
n-term-App: *n-term n t1* ⟹ *n-term n t2* ⟹ *n-term n* (*App t1 t2*)

Definition 6.2.1 — Shifting

primrec *shift* :: *int* ⇒ *nat* ⇒ *Term* ⇒ *Term* **where**
shift-Var: *shift d c* (*Var k*) = *Var* (*if k* < *c then k else nat* (*k* + *d*)) |
shift-Abs: *shift d c* (*Abs t*) = *Abs* (*shift d* (*Suc c*) *t*) |
shift-App: *shift d c* (*App t1 t2*) = *App* (*shift d c t1*) (*shift d c t2*)

Exercice 6.2.2

lemma *shift 2 0* (*Abs* (*Abs* (*App* (*Var 1*) (*App* (*Var 0*) (*Var 2*))))) =
Abs (*Abs* (*App* (*Var 1*) (*App* (*Var 0*) (*Var 4*))))
by *simp*

lemma *shift 2 0* (*Abs* (*App* (*Var 0*) (*App* (*Var 1*) (*Abs* (*App* (*Var 0*) (*App* (*Var 1*) (*Var 2*)))))))) =
Abs (*App* (*Var 0*) (*App* (*Var 3*) (*Abs* (*App* (*Var 0*) (*App* (*Var 1*) (*Var 4*)))))))
by *simp*

Exercice 6.2.3

lemma *n-term n t* ⟹ *n-term* (*n* + *d*) (*shift d c t*)
proof (*induction n t arbitrary*: *d c rule*: *n-term.induct*)
case (*n-term-Var k n*)
from *n-term-Var.hyps* **show** *?case*
using *n-term.n-term-Var* **by** *simp*
next
case (*n-term-Abs n t*)
from *n-term-Abs.hyps* **show** *?case*
using *n-term.n-term-Abs* **by** (*auto intro*: *n-term-Abs.IH*)
next
case (*n-term-App n t1 t2*)
show *?case*
by (*simp add*: *n-term.n-term-App n-term-App.IH*)
qed

Definition 6.2.4 — Substitution

```
primrec subst :: nat ⇒ Term ⇒ Term ⇒ Term where
  subst-Var: subst j s (Var k) = (if k = j then s else Var k) |
  subst-Abs: subst j s (Abs t) = Abs (subst (Suc j) (shift 1 0 s) t) |
  subst-App: subst j s (App t1 t2) = App (subst j s t1) (subst j s t2)
```

Exercice 6.2.5

```
lemma subst 0 (Var 1) (App (Var 0) (Abs (Abs (Var 2)))) =
                    App (Var 1) (Abs (Abs (Var 3)))
  by simp

lemma subst 0 (App (Var 1) (Abs (Var 2))) (App (Var 0) (Abs (Var 1))) =
  App (App (Var 1) (Abs (Var 2))) (Abs (App (Var 2) (Abs (Var 3))))
  by simp

lemma subst 0 (Var 1) (Abs (App (Var 0) (Var 2))) =
                    Abs (App (Var 0) (Var 2))
  by simp

lemma subst 0 (Var 1) (Abs (App (Var 1) (Var 0))) =
                    Abs (App (Var 2) (Var 0))
  by simp
```

Exercice 6.2.6

```
lemma n-term-shift: n-term n t ⟹ n-term (n + j) (shift j i t)
  by (induction n t arbitrary: j i rule: n-term.induct)
    (auto intro: n-term-Var n-term-Abs[unfolded One-nat-def] n-term-App)

lemma n-term n t ⟹ n-term n s ⟹ j ≤ n ⟹ n-term n (subst j s t)
proof (induction n t arbitrary: s j rule: n-term.induct)
  case (n-term-Var k n)
  thus ?case
    by (auto intro: n-term.n-term-Var)
next
  case (n-term-Abs n t)
  thus ?case
   using n-term.n-term-Abs n-term-shift[OF n-term-Abs.prems(1), where j = 1]
    by (auto
      intro: n-term-Abs.IH
      intro!: n-term.n-term-Abs[unfolded One-nat-def]
      simp: n-term-shift[OF n-term-Abs.prems(1), where j = 1])
next
  case (n-term-App n t1 t2)
  thus ?case
    by (simp add: n-term.n-term-App)
qed
```

Single step evaluation

```
inductive is-value :: Term ⇒ bool where
  is-value (Abs t)
```

```
inductive eval-once :: Term ⇒ Term ⇒ bool where
  eval-once-App1: eval-once t1 t1' ⟹ eval-once (App t1 t2) (App t1' t2) |
  eval-once-App2: is-value v1 ⟹ eval-once t2 t2' ⟹ eval-once (App v1 t2) (App
v1 t2') |
  eval-once-App-Abs: is-value v2 ⟹ eval-once (App (Abs t12) v2) (shift (−1) 0
(subst 0 (shift 1 0 v2) t12))
```

Theorem 3.5.4 for Untyped Lambda Calculus

```
theorem eval-once-right-unique:
  eval-once t t' ⟹ eval-once t t'' ⟹ t' = t''
proof (induction t t' arbitrary: t'' rule: eval-once.induct)
  case (eval-once-App1 t1 t1' t2)
  from eval-once-App1.hyps eval-once-App1.prems show ?case
    by (auto elim: eval-once.cases is-value.cases intro: eval-once-App1.IH)
next
  case (eval-once-App2 t1 t2 t2')
  from eval-once-App2.hyps eval-once-App2.prems show ?case
    by (auto elim: eval-once.cases is-value.cases intro: eval-once-App2.IH)
next
  case (eval-once-App-Abs v2 t12)
  from eval-once-App-Abs.prems eval-once-App-Abs.hyps show ?case
    by (auto elim: eval-once.cases simp: is-value.simps)
qed
```

Definition 3.5.6 for Untyped Lambda Calculus

```
definition is-normal-form :: Term ⇒ bool where
  is-normal-form t ⟷ (∀ t'. ¬ eval-once t t')
```

Theorem 3.5.7 for Untyped Lambda Calculus

```
theorem value-imp-normal-form: is-value t ⟹ is-normal-form t
  by (auto elim: is-value.cases eval-once.cases simp: is-normal-form-def)
```

Theorem 3.5.8 does not hold for Untyped Lambda calculus

```
theorem normal-form-does-not-imp-value:
  ∃ t. is-normal-form t ∧ ¬ is-value t (is ∃ t. ?P t)
proof
  have a: is-normal-form (Var 0)
    by (auto simp: is-normal-form-def elim: eval-once.cases)
  have b: ¬ is-value (Var 0)
    by (auto simp: is-normal-form-def dest: is-value.cases)
  from a b show ?P (Var 0) by simp
qed
```

Multistep evaluation

```
inductive eval :: Term ⇒ Term ⇒ bool where
  eval-base: eval t t |
  eval-step: eval-once t t' ⟹ eval t' t'' ⟹ eval t t''
```

Corollary 3.5.11 for Untyped Lambda Calculus

corollary *uniqueness-of-normal-form*:
  assumes
    eval t u **and**
    eval t u' **and**
    is-normal-form u **and**
    is-normal-form u'
  shows $u = u'$
using *assms*
proof (*induction t u rule*: *eval.induct*)
  case (*eval-base t*)
  thus *?case* **by** (*metis eval.simps is-normal-form-def*)
next
  case (*eval-step t1 t2 t3*)
  thus *?case* **by** (*metis eval.cases is-normal-form-def eval-once-right-unique*)
qed

lemma *eval-once-VarD*: *eval-once* (*Var x*) *t* $\Longrightarrow$ *P*
  by (*induction Var x t rule*: *eval-once.induct*)

lemma *eval-once-AbsD*: *eval-once* (*Abs x*) *t* $\Longrightarrow$ *P*
  by (*induction Abs x t rule*: *eval-once.induct*)

theorem *eval-does-not-always-terminate*:
  $\exists\, t.\ \forall\, t'.$ *eval t t'* $\longrightarrow \neg$ *is-normal-form t'* (**is** $\exists\, t.\ \forall\, t'.$ *?P t t'*)
proof
  let *?ω* = *Abs* (*App* (*Var 0*) (*Var 0*))
  let *?Ω* = *App ?ω ?ω*
  { fix *t*
    have *eval-once ?Ω t* $\Longrightarrow$ *?Ω* = *t*
      by (*induction ?Ω t rule*: *eval-once.induct*) (*auto elim*: *eval-once-AbsD*)
  } note *eval-once-Ω* = *this*
  { fix *t*
    have *eval-Ω*: *eval ?Ω t* $\Longrightarrow$ *?Ω* = *t*
      by (*induction ?Ω t rule*: *eval.induct*) (*blast dest*: *eval-once-Ω*)+
  } note *eval-Ω* = *this*
  show $\forall\, t'.$ *?P ?Ω t'*
    by (*auto*
      simp: *is-normal-form-def*
      intro: *eval-once-App-Abs is-value.intros*
      dest!: *eval-Ω*)
qed

end