Université Paris-Est

Habilitation à Diriger des Recherches

Spécialité : Informatique Fondamentale

Tristan Crolard

Certification de programmes impératifs d'ordre supérieur avec mécanismes de contrôle

Soutenue le 31 mars 2010 devant le jury composé de :

Olivier	Danvy	Rapporteurs
Hugo	Herbelin	
Daniel	Leivant	
Gianluigi	Bellin	Examinateurs
Thomas	Ehrhard	
Claude	Kirchner	
Anatol	Slissenko	

Première partie

Synthèse

Table des matières

Articles sélectionnés

Introduction

Les travaux présentés dans ce mémoire s'inscrivent dans un projet de recherche dont l'objectif est de définir un cadre formel pour certifier des programmes impératifs d'ordre supérieur utilisant des mécanismes de contrôle de flot. En particulier, les mécanismes de contrôle auxquels nous nous sommes intéressés sont les sauts non-locaux, les continuations délimitées et les coroutines.

Les « logiques de programmes » (*program logics*) pour les langages impératifs sont généralement des logiques dérivées des travaux [Floyd, 1967], [Hoare, 1969] et [Dijkstra, 1976]. Pour une présentation synthétique de la littérature abondante sur le sujet, nous renvoyons le lecteur vers [Apt, 1981] et [Cousot, 1990]. En comparaison, les logiques de programmes pour les programmes fonctionnels sont plus souvent basées sur l'interprétation des programmes comme des preuves [Curry and Feys, 1958, Howard, 1969] et sur des systèmes de types dépendants [Martin-Löf, 1975].

Relativement peu de ponts existent entre ces deux mondes, et il est donc difficile d'exploiter des résultats obtenus en théorie des types dans le monde des logiques de programmes. Une avancée récente remarquable en théorie des types de [Griffin, 1990] et [Murthy, 1990, Murthy, 1991] consiste en l'interprétation de mécanismes de contrôle génériques comme contenu calculatoire de raisonnements classiques (les théories des types étant traditionnellement constructives). L'intérêt de cette interprétation est en particulier d'être compatible avec la présence de fonctions d'ordre supérieur, cette combinaison étant par ailleurs réfractaire à l'axiomatisation sous forme de logique de Floyd-Hoare [O'Donnell, 1982].

L'objectif principal de ce travail est donc d'unifier ces deux approches afin de faire profiter les langages impératifs de ces résultats récents de théorie des types. L'approche de base, qui est à l'origine due à [Gordon, 1988], consiste à plonger la logique de Floyd-Hoare dans une logique d'ordre supérieur en encodant la sémantique dénotationnelle du langage impératif. Nous étendons et raffinons cette approche de plusieurs manières :

- nous factorisons ce plongement un passant par un *système de types dépendants impératif* où il est alors possible d'interpréter les programmes impératifs directement comme des preuves ;
- nous reformulons la sémantique dénotationnelle sous forme d'*une traduction de l'impératif vers le fonctionnel* qui fournit en outre une simulation de la sémantique opérationnelle ;
- nous étendons ce cadre formel aux *mécanismes de contrôle*, toujours en intégrant une simulation directe de l'impératif par le fonctionnel.

Le cadre formel que nous définissons est basé sur l'interprétation des programmes comme des preuves dans divers systèmes de déduction pour différentes logiques. Ce travail peut donc aussi être formulé comme une étude du contenu calculatoire de certaines logiques. De ce point de vue, un des objectifs est de comprendre le sens calculatoire de la logique soustractive (étudiée dans ma thèse) qui est une logique intermédiaire entre la logique intuitionniste et la logique classique.

Dans le contexte de l'interprétation des preuves comme des programmes, c'est le côté logique (théorie de la démonstration) qui détermine les constructions du langage de programmation et leur sémantique. D'une certaine façon, on ne fait donc que « découvrir » le sens calculatoire de ces constructions. Toutefois, l'expérience a montré que l'on retrouve presque toujours un mécanisme déjà existant dans les langages de programmation. L'exemple récent le plus frappant est l'interprétation du raisonnement classique par les opérateurs de contrôle [Griffin, 1990]. Parfois même, les différences de formulation des règles logiques vont correspondre à des variantes présentes dans les langages, comme par exemple, les différents types existentiels du second ordre (somme faible ou somme forte) qui correspondent à différent systèmes de modules (opaques ou translucides) [Cardelli and Leroy, 1990].

À l'inverse, évidemment, beaucoup de mécanismes des langages de programmation ne correspondent à aucune interprétation en termes de normalisation de preuve. La plupart ne possèdent même pas de définition en théorie de types et le programmeur dispose uniquement d'une sémantique purement opérationnelle. Le cadre dans lequel nous nous plaçons est donc nettement plus contraignant : les langages que nous considérons sont tous typés statiquement et, de plus, leurs systèmes de types admettent une lecture en tant que systèmes de déduction. Plus précisément, les ingrédients principaux de ce cadre formel prennent la forme suivante :

a) un langage impératif **I** muni d'un système de types dépendants **ID** et d'une sémantique opérationnelle ;

b) une traduction du langage **I** vers le langage fonctionnel **F** qui préserve le typage et qui fournit une simulation de la sémantique opérationnelle de **I** ;

c) une traduction des dérivations de typage impératives de **ID** vers des dérivations de typage fonctionnelles de **FD** ;

d) une interprétation de **FD** comme un système de déduction pour une logique (par exemple l'arithmétique classique, intuitionniste, ou soustractive).

Ce mémoire présente trois articles qui s'inscrivent dans ce cadre formel :

1. *A formulae-as-types interpretation of Subtractive Logic*
   [Crolard, 2004]

   Ce premier article, qui s'inscrit dans le point (d) ci-dessus, propose une interprétation calculatoire de la logique soustractive en termes de coroutines. Nous discuterons de l'extension de cette interprétation à l'arithmétique soustractive dans la suite de cette introduction.

2. *Extending the Loop Language with Higher-Order Procedural Variables*
   [Crolard et al., 2009]

   Dans ce second article, qui s'inscrit dans le point (b) ci-dessus, nous décrivons un langage impératif appelé Loop$^{\omega}$ qui peut être vu comme la contrepartie impérative du Système T de Gödel. Nous définissons en particulier une sémantique opérationnelle pour Loop$^{\omega}$ et une traduction vers le Système T et nous montrons que l'exécution d'un programme impératif est simulée pas-à-pas par l'évaluation de son image fonctionnelle (en appel par valeur).

3. *A program logic for higher-order procedural variables and non-local jumps*
   [Crolard and Polonowski, 2010]

   Ce troisième article complète le cadre, en particulier les points (a) et (c), en introduisant un système de types dépendants impératif pour Loop$^{\omega}$ d'une part et les sauts non-locaux d'autre part. Du côté fonctionnel, nous obtenons des programmes avec opérateurs de contrôle typés dans un système de déduction pour l'arithmétique classique.

Remarquez que nous nous limitons dans un premier temps à l'arithmétique : c'est un système logique bien compris dont le contenu calculatoire est suffisant pour exprimer de nombreux exemples non triviaux. Il est naturellement possible d'étendre ce cadre, en particulier à tout ce qui est bien compris dans le monde fonctionnel (les types de données algébriques, le polymorphisme, les modules...). Ces extensions seront considérées lors de travaux futurs.

Plan du mémoire

Les trois premières sections présentent brièvement les trois articles mentionnés ci-dessus, et quelques autres résultats connexes. Dans la Section 4, nous présentons certains travaux en cours, et nous revenons sur l'interprétation calculatoire de la logique soustractive en termes de coroutines. Pour cela, nous définissons tout d'abord une extension de la machine de Krivine pour exécuter des coroutines fonctionnelles de première classe (en appel par nom). Puis nous discutons quelques idées pour étendre le cadre formel décrit ci-dessus à des sémantiques opérationnelles impératives exprimées en termes de machines abstraites. L'objectif est d'une part de donner une sémantique opérationnelle directe pour les sauts non-locaux (et non plus uniquement une sémantique par passage de continuation) typables dans l'arithmétique classique et d'autre part de formaliser la sémantique opérationnelle d'un mécanisme de coroutines impératives (typable dans l'arithmétique soustractive).

1 Interprétation calculatoire de la logique soustractive

1.1 Présentation du premier article

La logique soustractive, définie dans [Rauszer, 1974b], (et étudiée dans ma thèse [Crolard, 1996]) est une extension de la logique intuitionniste avec un nouveau connecteur dual de l'implication (la soustraction). Une première formulation de l'interprétation calculatoire de la logique soustractive en termes de coroutines a été présentée dans [Crolard, 1999b]. Le calcul complet de coroutines ainsi que les preuves de ses propriétés principales, comme la normalisation forte et la propriété de la sous-formule ont été publiées dans [Crolard, 2004], qui est le premier article présenté dans ce mémoire.

La logique soustractive est particulièrement intéressante à étudier car elle est conservative sur la logique intuitionniste (dans le cadre propositionnel) tout en ayant une « saveur » de logique classique. Ses modèles algébriques et topologiques [Crolard, 2001] sont aussi plus élégants grâce à la symétrie complète des connecteurs et des quantificateurs :

$\top$	$\bot$
$\rightarrow$	$-$
$\wedge$	$\vee$
$\forall^2$	$\exists^2$

Une extension naturelle de ce travail consistait à comprendre le « sens calculatoire » de la logique soustractive. Le sens calculatoire comprend l'ensemble des mécanismes de calcul mis en œuvre lors de la normalisation (l'élimination des coupures) d'une preuve. Cette interprétation des preuves comme des programmes (et des formules comme des types) est communément appelée « correspondance de Curry-Howard » [Krivine and Parigot, 1990]. À l'exception de la soustraction, cette correspondance est bien comprise pour les connecteurs et quantificateurs du tableau ci-dessus [Krivine and Parigot, 1990]. Le vrai $\top$ correspond au type singleton, l'implication $\rightarrow$ au type des fonctions, la conjonction $\wedge$ au produit cartésien, le quantificateur universel $\forall^2$ au polymorphisme du second ordre. Le faux $\bot$ correspond au type universel, la disjonction $\vee$ à la somme disjointe, le quantificateur existentiel $\exists^2$ aux types abstraits de données [Mitchell and Plotkin, 1985].

L'interprétation calculatoire de la soustraction repose sur des techniques standard du domaine (comme la réalisabilité modifiée [Troelstra, 1973]) et sur le sens calculatoire de la logique classique en termes d'opérateurs de contrôle. Pour plus de détails sur le lien entre le typage des opérateurs de contrôle et le sens calculatoire des preuves en logique classique, nous renvoyons à la littérature abondante sur le sujet : [Griffin, 1990], [Murthy, 1990, Murthy, 1991], [Rehof and Sørensen, 1994], [Barbanera and Berardi, 1994b, Barbanera and Berardi, 1994a], [Krivine, 1994] et [de Groote, 1995] par exemple. Nous nous appuyons en particulier sur le $\lambda\mu$-calcul [Parigot, 1993b] et sur l'encodage des opérateurs **catch** et **throw** (étudié dans [Crolard, 1999a]).

Avant d'être plus précis, remarquons que la correspondance de Curry-Howard s'exprime généralement entre le λ-calcul et la déduction naturelle, systèmes dans lesquels la dualité n'est pas explicite. Par exemple, il faut se placer dans la théorie de catégories bicartésiennes fermées [Asperti and Longo, 1991] et le calcul des séquents [Crolard, 2001] pour observer la symétrie entre la conjonction et la disjonction. Dans [Herbelin, 1995], il est défini une interprétation calculatoire du calcul des séquents et, dans le cadre classique [Curien and Herbelin, 2000] ont montré un résultat frappant : la dualité échange les stratégies d'appel par nom et d'appel par valeur (cf. aussi [Wadler, 2003] et [Wadler, 2005]). Ce résultat donne une version syntaxique de la symétrie en sémantique catégorique étudiée tout d'abord dans [Filinski, 1989] puis plus en détail dans [Selinger, 2001].

Dans ces articles, la soustraction apparaît de manière formelle pour compléter la dualité. Son sens calculatoire est donné uniquement par la symétrie du calcul en logique classique. Même s'il est raisonnable d'espérer comprendre le contenu calculatoire de la soustraction en s'appuyant sur la dualité, il faut de plus comprendre le sens calculatoire de la restriction à la logique soustractive, et c'est l'objet de notre travail.

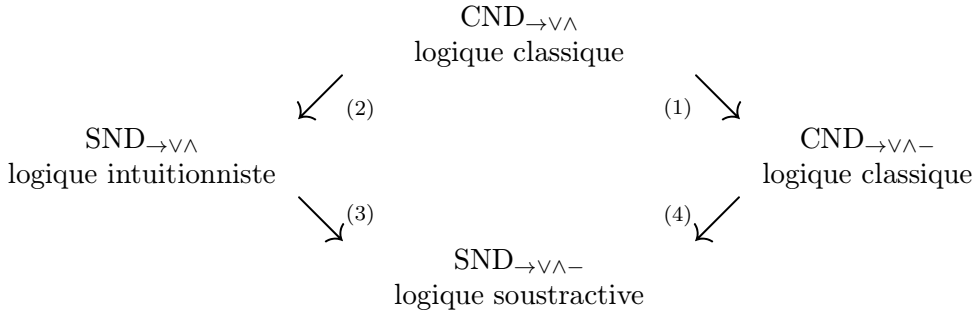
En particulier, la symétrie syntaxique entre l'implication et la soustraction disparaît des règles du calcul que nous proposons ici. La dualité est toutefois utilisée pour construire le calcul, et bien entendu, la logique associée au système de types sera la logique soustractive. Par exemple, le rôle essentiel de l'implication est de permettre le déchargement des hypothèses. Par conséquent, le rôle essentiel de la soustraction sera de permettre le déchargement des conclusions: le $\lambda\mu$-calcul [Parigot, 1992, Parigot, 1993a] et son système de types, la Déduction Naturelle Classique (CND) sont donc bien adaptés à cette étude (dans CND les séquents ont plusieurs hypothèses et plusieurs conclusions).

D'autre part, puisque l'implication est définissable en logique classique par $A \to B \equiv \neg A \vee B$, la soustraction est, par dualité, aussi définissable par $A - B \equiv A \wedge \neg B$. Cette définition nous donne une idée du sens calculatoire de la soustraction en logique classique : un objet de type $A - B$ peut être codé par un couple formé d'un terme (de type A) et d'une continuation réifiée en objet de première classe (de type $\neg B$).

Afin de comprendre le sens calculatoire de la soustraction en logique soustractive, il est raisonnable de commencer par considérer une restriction de CND où la soustraction n'est pas définissable, mais où les séquents ont toujours plusieurs conclusions. Une telle restriction a été définie dans [Crolard, 2002]. Du point de vue calculatoire, dans ce $\lambda\mu$-calcul restreint, les continuations ne peuvent pas être réifiées en objets de première classe, et la logique reste alors constructive : elle correspond à la logique intuitionniste dans le cas propositionnel et à la logique à domaine constant (CDL) [Görnemann, 1971] au premier ordre.

La méthodologie employée dans cet article consiste donc d'une part à dériver les règles d'introduction et d'élimination de la soustraction à partir de leur définition en logique classique et d'autre part de restreindre la Déduction Naturelle Classique à la logique intuitionniste avant de l'étendre à la Déduction Naturelle Soustractive (SND).

Ces différentes étapes sont résumées dans le diagramme suivant :



(1) Définition de la soustraction en logique classique ($A - B \equiv A \wedge \neg B$).
(2) Restriction de $\text{CND}_{\to\vee\wedge}$ à la logique intuitionniste (resp. CDL).
(3) Extension de $\text{SND}_{\to\vee\wedge}$ avec des règles contraintes pour la soustraction.
(4) Restriction duale de $\text{CND}_{\to\vee\wedge-}$ à la logique soustractive.

Relations de dépendance

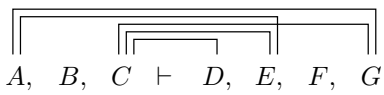
La restriction à la logique intuitionniste et sa duale, qui définissent des ensembles de preuves stables par élimination des coupures, est définie à partir d'une notion de relation de dépendance. Pour cela, nous décorons les séquents par des « liens de dépendance non orientés » entre hypothèses et conclusions. Étant donné un séquent $\Gamma_1, ..., \Gamma_n \vdash \Delta_1, ..., \Delta_m$, un lien de dépendance entre une hypothèse Γ_i et une conclusion Δ_j peut être représenté graphiquement de la manière suivante :

$$\Gamma_1 \,, \dots\ \Gamma_i \ , \dots\ \Gamma_n \ \vdash\ \Delta_1 \,, \dots\ \Delta_j \,, \dots \Delta_m$$

Pour formaliser ces liens, nous nommons les *occurrences* d'hypothèses x, y, z... et les *occurrences* de conclusion α, β, γ... de tout séquent. Un nom d'hypothèse, ou de conclusion n'apparaît jamais deux fois dans le même séquent.

Des liens de dépendance sur un séquent $\Gamma_1^{x_1}, ..., \Gamma_n^{x_n} \vdash \Delta_1^{\alpha_1}, ..., \Delta_m^{\alpha_m}$ forment donc une partie de $\{x_1, ..., x_n\} \times \{\alpha_1, ..., \alpha_m\}$ ou encore un ensemble de couples (x_i, α_j) où $1 \leq i \leq n$ et $1 \leq j \leq m$. Cette relation entre occurrences d'hypothèses et occurrences de conclusions peut donc être indifféremment représentée soit en décorant chaque conclusion par l'ensemble des hypothèses auxquelles elle est liée (on dira aussi « dont elle dépend ») soit en décorant chaque hypothèse par l'ensemble des conclusions auxquelles elle est liée (on dira aussi « qui en dépendent »).

Exemple. Considérons le séquent $A, B, C \vdash D, E, F, G$ muni des liens de dépendance suivants :



En nommant les hypothèses et les conclusions du séquent $A^x, B^y, C^z \vdash D^\alpha, E^\beta, F^\gamma, G^\delta$, les liens de dépendance correspondent aux couples $\{(x, \beta), (x, \delta), (z, \alpha), (z, \beta), (z, \delta)\}$. Ce séquent pourra donc être indifféremment représenté en décorant les conclusions :

$$A^x, B^y, C^z \vdash \{z\}{:}\, D, \{x, z\}{:}\, E, \{\}{:}\, F, \{x, z\}{:}\, G$$

ou en décorant les hypothèses :

$$\{\beta, \delta\}{:}A, \{\}{:}\, B, \{\alpha, \beta, \delta\}{:}\, C \vdash D^\alpha, E^\beta, F^\gamma, G^\delta$$

$$A^x \vdash \{x\}: A \quad (ax)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A^x \vdash \Delta}\,(W_L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \{\}: A}\,(W_R)$$

$$\frac{\Gamma, U: A, V: A \vdash \Delta}{\Gamma, U \cup V: A \vdash \Delta}\,(C_L) \qquad \frac{\Gamma \vdash \Delta, U: A, V: A}{\Gamma \vdash \Delta, U \cup V: A}\,(C_R)$$

$$\frac{\Gamma \vdash \Delta, S: A \qquad \Gamma', A^x \vdash S'_1: \Delta'_1, ..., S'_p: \Delta'_p}{\Gamma, \Gamma' \vdash \Delta, S'_1[S/x]: \Delta'_1, ..., S'_p[S/x]: \Delta'_p}\,(cut)$$

$$\frac{\Gamma, A^x \vdash S_1: \Delta_1, ..., S_n: \Delta_n, S: B}{\Gamma \vdash S_1 \setminus \{x\}: \Delta_1, ..., S_n \setminus \{x\}: \Delta_n, S \setminus \{x\}: (A \to B)}\,(I_\to)$$

$$\frac{\Gamma \vdash \Delta, U: (A \to B) \qquad \Gamma' \vdash \Delta', V: A}{\Gamma, \Gamma' \vdash \Delta, \Delta', U \cup V: B}\,(E_\to)$$

$$\frac{\Gamma \vdash \Delta, S: A \wedge B}{\Gamma \vdash \Delta, S: A}\,(E^1_\wedge) \qquad \frac{\Gamma \vdash \Delta, S: A \wedge B}{\Gamma \vdash \Delta, S: B}\,(E^2_\wedge)$$

$$\frac{\Gamma \vdash \Delta, U: A \qquad \Gamma' \vdash \Delta', V: B}{\Gamma, \Gamma' \vdash \Delta, \Delta', U \cup V: A \wedge B}\,(I_\wedge)$$

$$\frac{\Gamma, S: A \vee B \vdash \Delta}{\Gamma, S: A \vdash \Delta}\,(LE^1_\vee) \quad \frac{\Gamma, S: A \vee B \vdash \Delta}{\Gamma, S: B \vdash \Delta}\,(LE^2_\vee)$$

$$\frac{\Gamma, U: A \vdash \Delta \qquad \Gamma', V: B \vdash \Delta'}{\Gamma, \Gamma', U \cup V: A \vee B \vdash \Delta, \Delta'}\,(LI_\vee)$$

$$\frac{\Gamma, U: A - B \vdash \Delta \qquad \Gamma', V: B \vdash \Delta'}{\Gamma, \Gamma', U \cup V: A \vdash \Delta, \Delta'}\,(LE_-)$$

$$\frac{S_1: \Gamma_1, ..., S_m: \Gamma_m, S: A \vdash \Delta, B^\beta}{S_1 \setminus \{\beta\}: \Gamma_1, ..., S_m \setminus \{\beta\}: \Gamma_m, S \setminus \{\beta\}: A - B \vdash \Delta}\,(LI_-)$$

Figure 1. Annotations pour $\mathrm{CND}_{\to\vee\wedge-}$ symétrique

Le système de déduction qui infère les liens de dépendance, appelé « CND symétrique », est reproduit en Figure 1. Ses règles sont celles attendues pour $\mathrm{CND}_{\wedge\to}$ (où les conclusions des séquents sont décorées) et elles sont complétées par les règles duales pour $\vee$ et $-$ (où, par symétrie, cette fois-ci les hypothèses sont décorées). Le système $\mathrm{CND}_{\to\vee\wedge-}$ peut alors être dérivé en remettant systématiquement toutes les décorations sur les conclusions. De plus, afin de rejoindre le système de types du $\lambda\mu$-calcul, nous distinguons une conclusion dans chaque séquent, séparée par un point-virgule. Par exemple, la règle d'élimination de la soustraction dérivée est la suivante :

$$\frac{\Gamma \vdash S_1: \Delta_1, ..., S_n: \Delta_n; S: A - B \qquad \Gamma, A^x \vdash S'_1: \Delta_1, ..., S'_n: \Delta_n; U: B}{\Gamma \vdash S_1 \cup S'_1[S/x]: \Delta_1, ..., S_n \cup S'_n[S/x]: \Delta_n; \{\}: C}\,(E_-)$$

Remarquez que ce système de déduction est toujours classique, la restriction à la logique intuitionniste (resp. soustractive) est décrite ci-dessous.

Contrainte intuitionniste

On dit qu'une instance de la règle de d'introduction de l'implication respecte la *contrainte intuitionniste* si x n'apparaît dans aucun S_i. La règle devient alors :

$$\frac{\Gamma, A^x \vdash S_1: \Delta_1, ..., S_n: \Delta_n; V: B}{\Gamma \vdash S_1: \Delta_1, ..., S_n: \Delta_n; V \setminus \{x\}: (A \to B)} \quad \text{où } x \notin S_1 \cup ... \cup S_n$$

Une preuve de CND à liens de dépendance explicites est dite *intuitionniste* si toute introduction de l'implication apparaissant dans la preuve respecte la contrainte intuitionniste.

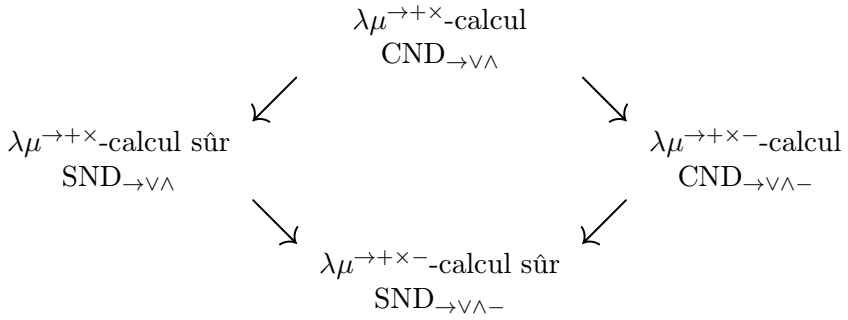
Par dualité, on dit qu'une instance de la règle d'élimination de la soustraction (E_-) respecte la *contrainte intuitionniste* ssi $U \subseteq \{x\}$. La règle devient alors :

$$\frac{\Gamma \vdash S_1 : \Delta_1, ..., S_n : \Delta_n; S : A - B \qquad \Gamma, A^x \vdash S'_1 : \Delta_1, ..., S'_n : \Delta_n; U : B}{\Gamma \vdash S_1 \cup S'_1[S/x] : \Delta_1, ..., S_n \cup S'_n[S/x] : \Delta_n; \{\} : C}\,(E_-) \quad \text{où } U \subseteq \{x\}$$

Une preuve de CND (resp. SND) à liens de dépendance explicites est dite *intuitionniste* si toute introduction de l'implication (resp. et toute élimination de la soustraction) apparaissant dans la preuve respecte la contrainte intuitionniste.

Normalisation forte et propriété de la sous-formule

La contrainte ci-dessus peut-être reformulée pour le $\lambda\mu$-calcul pur de manière à ce que les deux restrictions coïncident sur les termes typés. On appelle « $\lambda\mu^{\to+\times-}$-calcul sûr » (resp. « $\lambda\mu^{\to+\times}$-calcul sûr ») le sous-ensemble des termes contraints (la terminologie sera justifiée plus tard). Un des résultats principaux de cet article exprime la stabilité par réduction de ces sous-ensembles, ce qui nous permet de les qualifier de « calculs ». On peut donc compléter le diagramme ci-dessus :



La correspondance entre la notion de sûreté d'un $\lambda\mu^{\to+\times-}$-terme (resp. $\lambda\mu^{\to+\times}$-terme) et la restriction du système de déduction à $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$) peut s'exprimer ainsi : étant donné une dérivation de typage d'un $\lambda\mu^{\to+\times-}$-terme (resp. $\lambda\mu^{\to+\times}$-terme) t par un séquent de $\mathrm{CND}_{\to\vee\wedge-}$ (resp. $\mathrm{CND}_{\to\vee\wedge}$), si ce terme t est sûr, alors la dérivation est valide dans $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$). En conséquence, la préservation du typage par réduction dans $\mathrm{CND}_{\to\vee\wedge-}$ (resp. $\mathrm{CND}_{\to\vee\wedge}$) combinée avec la stabilité par réduction du $\lambda\mu^{\to+\times-}$-calcul sûr (resp. $\lambda\mu^{\to+\times}$-calcul sûr) nous donne la préservation du typage par réduction pour $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$).

Ces calculs sont tous basés sur le $\lambda\mu^{\to\wedge\vee\bot}$-calcul [de Groote, 2001] et possèdent donc les mêmes bonnes propriétés. En effet, la relation de réduction est définie de manière locale, sur les termes non-typés. On peut montrer que cette relation est confluente et préserve le typage et que les termes typables sont fortement normalisables. Enfin, les dérivations de typage en forme normale vérifient la propriété de la sous-formule. Comme corollaire de cette dernière propriété, nous obtenons que la forme normale d'une dérivation d'un séquent dans $\mathrm{CND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge-}$) qui ne contient pas de soustraction est une dérivation valide de $\mathrm{CND}_{\to\vee\wedge}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$). Comme sous-produit de ces propriétés, nous obtenons une nouvelle preuve de ce résultat, et en fait un résultat plus général, à savoir la normalisation forte.

Nous donnons ci-après quelques précisions concernant la définition et l'interprétation calculatoire de ces différents $\lambda\mu$-calculs.

Interprétation calculatoire

Le $\lambda\mu^{\rightarrow+\times}$-calcul « sûr » est obtenu en contraignant le $\lambda\mu^{\rightarrow+\times}$-calcul en se basant sur une notion de visibilité : en effet, les relations de dépendance définies au niveau logique (donc du typage) associent à chaque μ-variable (chaque conclusion du séquent) un sous-ensemble de l'environnement. Du point de vue calculatoire, cela se traduit par le fait que toutes les variables ne sont pas visibles pour toutes les continuations (associées aux μ-variables). Cette notion de visibilité est définie par récurrence sur le terme (puisque les relations de dépendance sont définies par récurrence sur la dérivation) : à chaque μ-variable δ nous associons un sous-ensemble $\mathcal{S}_\delta(u)$ des variables utiles à la continuation δ. La lettre $\mathcal{S}$ vient de *shared* ; en effet, si une variable est utile à plusieurs continuations, elle devient *partagée*.

Définition. *On dit qu'un $\lambda\mu^{\rightarrow+\times}$-terme t est « sûr » si et seulement si pour tout sous-terme de t de la forme $\lambda x.u$, et pour toute μ-variable δ libre dans u, on a $x \notin \mathcal{S}_\delta(u)$.*

Autrement dit, ne sont visibles pour une continuation que les λ-variables déjà déclarées avant sa propre déclaration, autrement dit son environnement de déclaration. En effet, dans $\lambda x.u$, x n'est visible que de la continuation courante et ne doit donc pas être utile à une autre continuation dans u. Cette façon d'associer un fragment de l'environnement à une continuation rappelle la manière dont sont implantées les coroutines dans un langage à pile. Pour refléter cette intuition, nous renommons donc les opérateurs dérivés **catch** et **throw** [Crolard, 1999a] en **get-context** et **set-context**. Cette terminologie est inspirée des primitives permettant de créer et de manipuler des coroutines dans un système Unix standard [The Open Group, 1997] et sera justifiée dans le contexte des machines à environnement en section 4. Pour le moment, contentons-nous de l'intuition suivante : un contexte de coroutines est une paire composée d'un environnement et d'une continuation.

Le $\lambda\mu^{\rightarrow+\times-}$-calcul « sûr »

Pour définir le $\lambda\mu^{\rightarrow+\times-}$-calcul, nous utilisons le fait que la soustraction est définissable en logique classique (par $A - B \equiv A \wedge \neg B$). La dérivation des règles d'introduction et d'élimination de la soustraction nous donne alors les macro-définitions suivantes :

- **make-coroutine** $t\ \alpha \equiv \langle t, \lambda x.\textbf{set-context}\ \alpha\ x\rangle$
- **resume** c **with** $x \mapsto u \equiv$ **match** c **with** $\langle x, k\rangle \mapsto (k\ u)$

À nouveau, la terminologie provient de l'interprétation en termes de coroutines : une coroutine de première classe (typée par la soustraction) contient son environnement local, fourni lors de sa création. Inversement, lorsqu'on reprend l'exécution d'une coroutine, celle-ci va commencer par restaurer son environnement local : une seconde contrainte, duale de la précédente est donc nécessaire pour garantir que seules ces variables locales soient utilisées. Cette seconde contrainte permet de caractériser le $\lambda\mu^{\rightarrow+\times-}$-calcul « sûr ».

Définition 1. *On dit qu'un $\lambda\mu^{+\times-}$-terme t est « sûr » si et seulement si:*

1. *pour tout sous-terme de t de la forme $\lambda x.u$, et pour toute μ-variable δ libre dans u, on a $x \notin \mathcal{S}_\delta(u)$;*

2. *pour tout sous-terme de t de la forme* **resume** c **with** $x \mapsto u$, *on a* $\mathcal{S}_{[]}(u) \subseteq \{x\}$.

Remarque. Si la coroutine a été réifiée en objet de première classe, l'environnement de déclaration n'est plus connu, par conséquent il est nécessaire de restreindre l'ensemble des variables visibles à la variable x. Cette variable sera liée, lors de l'exécution, au terme t (donné comme argument à **make-coroutine**) pour fabriquer le nouvel environnement local. Nous reviendrons sur cette interprétation calculatoire en section 4.

1.2 À propos des systèmes avec relations de dépendance

D'autres systèmes basés sur les relations de dépendance ont été découvert indépendemment par plusieurs auteurs à la même période, en particulier dans [Kashima, 1991], [Shimura and Kashima, 1994], [Hyland and de Paiva, 1993] et [Crolard, 1996]. Plus de détails sur l'histoire de ces systèmes sont présentés dans [de Paiva and Pereira, 2005]. En particulier, le système défini par Kashima est le premier système de déduction pour CDL admettant l'élimination des coupures. On peut noter que la preuve de normalisation de $\mathrm{SND}_{\to\vee\wedge}$ (au premier ordre) donne une nouvelle preuve d'élimination des coupures pour CDL.

Par ailleurs, d'autres types de systèmes de déduction pour la logique soustractive, orientés recherche de preuve (ou de contre-exemple) automatique, ont été développés depuis (cf. [Uustalu and Pinto, 2006] et [Buisman and Goré, 2007]).

Remarque. Dans [Rauszer, 1974a], un calcul des séquents pour la logique soustractive est défini et C. Rauszer annonce la propriété d'élimination des coupures et la propriété de la sous-formule pour ce calcul. Toutefois, Uustalu a récemment exhibé un contre-exemple (présenté dans [Buisman and Goré, 2007]) : le séquent suivant n'admet pas de dérivation sans coupure dans le calcul de [Rauszer, 1974a] :

$$p \vdash q, (r \to ((p - q) \wedge r))$$

En revanche, en voici une dérivation sans coupure dans le système $\mathrm{SND}_{\to\vee\wedge-}$:

$$\dfrac{\dfrac{p \vdash q, (p - q) \qquad r \vdash r}{p,\ r \vdash q, ((p - q) \wedge r)}}{p \vdash q, (r \to ((p - q) \wedge r))}$$

1.3 Logique à domaine constant et arithmétique

Il est connu depuis [Grzegorczyk, 1964] que le calcul des prédicats étendu par le schéma d'axiomes DIS ci-dessous (où x n'apparaît pas libre dans B) axiomatise les modèles de Kripke à domaine constant [Görnemann, 1971] :

$$\forall x (A \vee B) \vdash \forall x\, A \vee B \qquad \text{(DIS)}$$

En revanche, ajouter DIS à l'arithmétique de Heyting donne l'arithmétique de Peano. Ce résultat, déjà mentionné dans [Troelstra, 1973] page 92, découle de la décidabilité des formules atomiques dans HA. En effet, on peut dériver $\vdash (n=m) \vee \neg(n=m)$ dans HA en utilisant deux récurrences imbriquées sur n et m. On peut ensuite dériver $\vdash A \vee \neg A$ dans HA par récurrence sur A pour n'importe quelle formule A sans quantificateur. Il est alors possible de dériver le tiers-exclu pour des formules contenant aussi des quantificateurs en utilisant DIS (où les règles (a), (b), (c) et (d) sont facilement dérivables en logique intuitionniste).

$$\dfrac{\dfrac{\dfrac{\vdash A(x) \vee \neg A(x)}{\vdash A(x) \vee \exists x.\neg A(x)}(a)}{\vdash \forall x(A(x) \vee \exists x.\neg A(x))}}{\dfrac{\vdash \forall x.A(x) \vee \exists x.\neg A(x)}{\vdash \forall x.A(x) \vee \neg \forall x.A(x)}(b)}(\text{DIS}) \qquad \dfrac{\dfrac{\dfrac{\vdash A(x) \vee \neg A(x)}{\vdash \exists x.A(x) \vee \neg A(x)}(c)}{\vdash \forall x(\exists x.A(x) \vee \neg A(x))}}{\dfrac{\vdash \exists x.A(x) \vee \forall x.\neg A(x)}{\vdash \exists x.A(x) \vee \neg \exists x.A(x)}(d)}(\text{DIS})$$

Il semble donc désespéré de chercher à étendre l'arithmétique de Heyting avec DIS en restant constructif. Le même problème se pose avec la logique soustractive, puisque DIS est alors dérivable. En réalité, il suffit de prendre une autre formulation de l'arithmétique intuitionniste, appelée $\mathbf{IT}(\mathbb{N})$ dans [Leivant, 2002], pour contourner ce problème.

Arithmétique soustractive

Rappelons tout d'abord la sémantique de Kripke de la logique soustractive au premier ordre. Un modèle de Kripke est donné par un pré-ordre qui engendre une topologie dont les ouverts sont les sections finales du pré-ordre (c'est la topologie d'Alexandroff [Erné et al., 2007]). Cette topologie possède la particularité suivante : l'ensemble des ouverts est clos aussi par intersection quelconque. Il est donc possible de définir un opérateur de couverture *couv* qui associe à chaque ensemble le plus petit ouvert qui le contient.

Étant donné un modèle de Kripke avec domaine constant $\mathcal{D}$ et une interprétation des symboles de fonctions et de prédicats ε, la sémantique d'une formule du premier ordre est la suivante :

- $[\![p(t_1,...,t_n)]\!]_\varepsilon = \varepsilon(p)([\![t_1]\!]_\varepsilon,...,[\![t_n]\!]_\varepsilon)$
- $[\![A \vee B]\!]_\varepsilon = [\![A]\!]_\varepsilon \cup [\![B]\!]_\varepsilon$
- $[\![A \wedge B]\!]_\varepsilon = [\![A]\!]_\varepsilon \cap [\![B]\!]_\varepsilon$
- $[\![A \Rightarrow B]\!]_\varepsilon = int([\![A]\!]^c_\varepsilon \cup [\![B]\!]_\varepsilon)$
- $[\![A - B]\!]_\varepsilon = couv([\![A]\!]_\varepsilon \cap [\![B]\!]^c_\varepsilon)$
- $[\![\forall x.A]\!]_\varepsilon = \bigcap_{a \in \mathcal{D}} [\![A]\!]_{\varepsilon, x=a}$
- $[\![\exists x.A]\!]_\varepsilon = \bigcup_{a \in \mathcal{D}} [\![A]\!]_{\varepsilon, x=a}$

Remarquons maintenant que les modèles à domaines variants peuvent être simulés par des modèles à domaine constant de la manière suivante [Fitting, 2004]. Nous ajoutons au langage un symbole de prédicat unaire, noté E, qui représente un prédicat d'existence primitif. Les quantificateurs relativisés sont aussi introduits, $\forall^E x.\varphi$ pour $\forall x(E(x) \Rightarrow \varphi)$ et $\exists^E x.\varphi$ pour $\exists x(E(x) \wedge \varphi)$.

Nous pouvons alors encoder un modèle à domaine variant $\mathcal{V}$ comme un modèle à domaine constant $\mathcal{M}$ (basé sur le même modèle de Kripke), dont le domaine est l'union de tous les domaines de $\mathcal{V}$, et en interprétant E dans chaque monde m de $\mathcal{M}$ comme l'appartenance au domaine de m dans $\mathcal{V}$. On obtient clairement que la sémantique de $\forall x.\varphi$ dans $\mathcal{V}$ est la même de celle de $\forall^E x.\varphi$ dans $\mathcal{M}$ (et de même pour $\exists$).

Inversement, partant d'un modèle à domaine constant $\mathcal{M}$ et d'une interprétation de E, nous pouvons construire le modèle à domaines variants $\mathcal{V}$ où le domaine de chaque monde est obtenu en prenant la restriction du domaine de $\mathcal{M}$ par E. À nouveau, la sémantique de $\forall^E x.\varphi$ dans $\mathcal{M}$ est alors la même de celle de $\forall x.\varphi$ dans $\mathcal{V}$ (et de même pour $\exists$).

Remarquez que l'argument ci-dessus se généralise à la logique du premier ordre multi-sortée. En effet, l'encodage standard [Enderton, 1972, Gallier, 2003] de la logique multi-sortée dans la logique mono-sortée consiste à se donner un symbole de prédicat unaire pour chaque sorte et à relativiser les quantifications. Dans le cas de la sémantique de Kripke, on obtient alors naturellement un encodage de la logique *intuitionniste* multi-sortée dans la *logique à domaine constant* (mono-sortée).

Ce résultat s'étend directement à la logique soustractive puisque la soustraction est interprétable dans les modèles à domaine constant. Dans le cas particulier de l'arithmétique, nous noterons **nat** le prédicat d'existence E, et la formulation obtenue correspond à la version intuitionniste de $\mathbf{IT}(\mathbb{N})$ de [Leivant, 2002] (cf. article 3). La conséquence de la remarque ci-dessus est que l'on peut alors ajouter DIS à cette formulation de l'arithmétique de Heyting et rester conservatif. La conservativité s'exprime ainsi : une formule relativisée est prouvable dans $\mathbf{IT}(\mathbb{N})$ minimal si et seulement si elle est prouvable dans $\mathbf{IT}(\mathbb{N})$ minimal + DIS. Et le même résultat de conservativité est valable pour l'arithmétique soustractive (cf. [Wehmeier, 1996] pour une étude des modèles de Kripke de HA).

La preuve syntaxique de conservativité de la logique soustractive sur la logique intuitionniste (cf. article 1) s'étend au premier ordre en une preuve de conservativité sur la logique à domaine constant [Crolard, 1996]. Toutefois, dans le cas des formules relativisées, on peut supposer sans perte de généralité que la règle d'introduction du quantificateur universel est couplée avec la règle d'introduction de l'implication. Nous allons donc étendre (dans un travail futur) la preuve donnée dans [Crolard, 1996] aux quantificateurs relativisés en appliquant la même technique, mais où l'implication est généralisée en un produit dépendant.

Arithmétique, dualité et logique du second ordre

Dans cette formulation de l'arithmétique, la dualité est perdue puisque il n'y pas de dual défini pour le prédicat **nat**. Tentons de rajouter un nouveaux prédicat $\mathbf{nat}^\perp$, avec les axiomes pour 0 et S et un principe de « récurrence duale » relativisée :

$$\begin{array}{rcl@{\qquad}rcl}
\top & \vdash & \mathbf{nat}(0) & \mathbf{nat}^\perp(0) & \vdash & \perp \\
\mathbf{nat}(n) & \vdash & \mathbf{nat}(S(n)) & \mathbf{nat}^\perp(S(n)) & \vdash & \mathbf{nat}^\perp(n) \\
\varphi[0/n] \wedge \forall n(\varphi \Rightarrow \varphi[Sn/n]) & \vdash & \forall n(\mathbf{nat}(n) \Rightarrow \varphi) & \exists n(\varphi - \mathbf{nat}^\perp(n)) & \vdash & \varphi[0/n] \vee \exists n(\varphi[Sn/n] - \varphi)
\end{array}$$

A partir de $\vdash \neg\mathbf{nat}^\perp(0)$ et $\vdash \forall n(\neg\mathbf{nat}^\perp(n) \Rightarrow \neg\mathbf{nat}^\perp(S(n)))$ où $\neg$ est la négation intuitionniste, on obtient par récurrence $\vdash \forall n(\mathbf{nat}(n) \Rightarrow \neg\mathbf{nat}^\perp(n))$ et par conséquent $\mathbf{nat}(n) \wedge \mathbf{nat}^\perp(n) \vdash \perp$. Par dualité, en utilisant le principe de récurrence duale, on dérive alors $\vdash \mathbf{nat}^\perp(n) \vee \mathbf{nat}(n)$ et ensuite $\vdash \neg\mathbf{nat}(n) \vee \mathbf{nat}(n)$. Le prédicat **nat** est donc décidable. Par conséquent, toutes les formules relativisées sont à nouveau décidables (en utilisant DIS) et on peut donc prouver tous les théorèmes de l'arithmétique de Peano.

Les axiomes ci-dessus pour **nat** sont en particulier dérivables au second ordre pour la définition habituelle des entiers naturels $Nat(m) = \forall X(X(0) \Rightarrow \forall n(X(n) \Rightarrow X(S(n))) \Rightarrow X(m))$. Par dualité, les axiomes pour $\mathbf{nat}^\perp$ sont dérivables pour la définition duale suivante : $Nat^\perp(m) = \exists X(X(m) - \exists n(X(S(n)) - X(n)) - X(0))$. En conséquence, si l'on ajoute « naïvement » la soustraction à l'arithmétique intuitionniste du second ordre, on obtient l'arithmétique de Peano comme fragment du premier ordre. Le résultat de conservativité au premier ordre décrit ci-dessus ne se généralise donc pas au second ordre (puisque $\vdash \forall n(\neg Nat(n) \vee Nat(n))$ n'est pas dérivable au second ordre intuitionniste).

La technique pour contourner ce problème, et rester conservatif sur la logique intuitionniste au second ordre, est à nouveau la même : il suffit de considérer l'encodage habituel de la logique du second-ordre comme la théorie du premier ordre 2-sortée (une sorte pour les individus et une sorte pour les ensembles) axiomatisée par le schéma de compréhension [Krivine, 1993]. On obtient alors en particulier le théorème de complétude comme corollaire de la complétude pour la logique soustractive au premier ordre [Rauszer, 1976] (et la conservativité en découle).

1.4 Logique de la pragmatique

La « logique de la pragmatique » a été développée à l'origine par le philosophe Carlo Dalla Pozza et le physicien Claudio Garola [Dalla Pozza and Garola, 1995] pour modéliser les « actes de langage » (la *pragmatique* est le champ de la linguistique qui couvre ce thème de recherche), et en particulier les notions d'*assertion* et de *conjecture*. Ce projet de recherche a été repris par Gianluigi Bellin et ses élèves qui se sont alors intéressés aux aspects sémantiques [Bellin and Biasi, 2004, Bellin and Ranalter, 2003] et catégoriques [Ranalter, 2008], à théorie de la démonstration et, plus récemment, au contenu calculatoire de cette logique [Biasi and Aschieri, 2008].

La logique de la pragmatique a été formalisée au départ, comme c'est l'usage pour les logiques modales, en se basant sur la sémantique de Kripke. Un premier point important de cette modélisation vient du constat que les assertions doivent être *justifiées*, et on rejoint là naturellement les sémantiques constructivistes. Le fragment de la logique de la pragmatique restreint aux assertions correspond donc à la logique intuitionniste.

La seconde particularité de cette modélisation provient de la remarque suivante : la notion de conjecture est naturellement duale de celle d'assertion. Le fragment de la logique de la pragmatique restreint aux conjectures correspond alors à une logique co-intuitionniste, qui contient donc un opérateur de soustraction.

L'ensemble nous donne une logique polarisée, et le passage d'un fragment à l'autre peut être rendu explicite dans la syntaxe à l'aide d'un opérateur de dualité $^\perp$, qui correspond à une négation classique. Pour être plus précis, rappelons la définition (tirée de [Bellin, 2005]) des ensembles de formules $\mathcal{L}_\vartheta$ (langage des assertions) et $\mathcal{L}_\upsilon$ (langages des conjectures) par les grammaires suivantes :

$$\begin{array}{rcl} \alpha & ::= & p \mid p^\perp \\ \vartheta & ::= & \vdash\alpha \mid \top \mid \vartheta \Rightarrow \vartheta \mid \vartheta \wedge \vartheta \\ \upsilon & ::= & \mathcal{H}\,\alpha \mid \bot \mid \upsilon - \upsilon \mid \upsilon \vee \upsilon \end{array}$$

La dualité $^\perp$ entre $\mathcal{L}_\vartheta$ et $\mathcal{L}_\upsilon$ est alors définie inductivement ainsi :

$$\begin{array}{rclcrcl} (\vdash p)^\perp & = & \mathcal{H}\,p^\perp & \qquad & (\mathcal{H}\,p)^\perp & = & \vdash p^\perp \\ (\vdash p^\perp)^\perp & = & \mathcal{H}\,p & & (\mathcal{H}\,p^\perp)^\perp & = & \vdash p \\ (\top)^\perp & = & \bot & & (\bot)^\perp & = & \top \\ (\vartheta_0 \Rightarrow \vartheta_1)^\perp & = & (\vartheta_1^\perp - \vartheta_0^\perp) & & (\upsilon_0 - \upsilon_1)^\perp & = & (\upsilon_1^\perp \Rightarrow \upsilon_0^\perp) \\ (\vartheta_0 \wedge \vartheta_1)^\perp & = & (\vartheta_1^\perp \vee \vartheta_0^\perp) & & (\upsilon_0 \vee \upsilon_1)^\perp & = & (\upsilon_1^\perp \wedge \upsilon_0^\perp) \end{array}$$

Une sémantique topologique simple, qui induit une sémantique de Kripke dans le cas particulier des topologies d'Alexandroff, est obtenue en interprétant les assertions par des ouverts et les conjectures par des fermés. La dualité est interprétée par le complément ensembliste, les modalités $\vdash$ et $\mathcal{H}$ par les opérateurs d'intérieur et de clôture respectivement, l'implication par sa sémantique intuitionniste et la conjonction par l'intersection. Les autres connecteurs sont obtenus par dualité.

La logique des pragmatiques diffère donc de manière significative de la logique soustractive de [Rauszer, 1974b] qui n'est pas polarisée, et où les formules peuvent mixer arbitrairement tous les connecteurs et sont toutes interprétées par des ouverts de la topologie d'Alexandroff. Toutefois, les deux fragments, intuitionniste et co-intuitionniste, sont inclus dans la logique soustractive. Par conséquent, l'interprétation calculatoire du fragment co-intuitionniste [Bellin, 2005] devrait apporter un éclairage nouveau sur l'interprétation calculatoire de la logique soustractive de [Crolard, 2004].

2 Procédures d'ordre supérieur et variables procédurales

2.1 Présentation du second article

Dans le deuxième article, nous définissons le langage Loop$^\omega$, qui est une extension (typée statiquement) du langage Loop avec des variables procédurales d'ordre supérieur. Le langage Loop [Meyer and Ritchie, 1976] est un langage impératif simple contenant uniquement l'affectation, la séquence et la boucle **for**. Dans [Meyer and Ritchie, 1976], il est prouvé que les fonctions calculables par des programmes Loop sont exactement les fonctions primitives récursives. Nous prouvons ici que les fonctions calculables par des programmes Loop$^\omega$ sont exactement les fonctionnelles primitives récursives de type fini (les fonctionnelles du Système T).

Rappelons que le système T a été introduit par Gödel dans son étude de la consistance de l'arithmétique de Peano [Gödel, 1958]. Ce système peut aussi être formulé comme un λ-calcul simplement typé étendu avec un type atomique pour les entiers et la récurrence pour tous les types. Il est bien connu que les fonctionnelles ainsi représentables permettent de donner une sémantique dénotationnelle à certaines constructions présentes dans les langages de programmation d'ordre supérieurs [Girard et al., 1989]. Nous verrons que cette expressivité peut être aussi capturée par une syntaxe impérative et nous démontrerons que Loop$^\omega$ est un candidat naturel comme contrepartie impérative du système T.

Il est important de noter que Loop$^\omega$ est un authentique langage impératif, avec des procédures (éventuellement d'ordre supérieur) et des variables mutables (pouvant contenir des procédures). En particulier, les exemples donnés dans cet article peuvent être réécrits en utilisant la syntaxe de $C\#$ (où les procédures de première classe sont appelées « délégués » [ISO, 2003]) et ensuite compilés avec un compilateur $C\#$.

Toutefois, c'est un langage impératif « pur » dans le sens suivant : son système de types interdit les effets de bord et « l'aliasing » des paramètres effectifs lors d'un appel procédure. Ces contraintes sont celles habituellement imposées pour rendre la sémantique du langage non ambiguë [Gellerich and Plödereder, 2001]. En effet, en présence d'alias la sémantique de l'appel de procédure dépend du mode de passage des paramètres, par adresse ou par copie (et dans le second cas, de l'ordre dans lequel les paramètres sont copiés). L'utilisation de variables globales (non-locales) peut engendrer d'autres formes d'alias plus difficiles à détecter, et peut en général être simulée par un passage explicite de paramètre (en transformant la procédure en *state-passing style*).

De manière moins habituelle, le système de types de Loop$^\omega$ interdit aussi l'accès en lecture des variables mutables non-locales. L'objectif de cette contrainte supplémentaire est d'empêcher l'encodage bien connu d'un point fixe à l'aide des variables procédurales en appliquant la technique du *back-patching* [Landin, 1964]. Voici un exemple simple de bloc, interdit par le système de types, qui implante un point-fixe :

$$
\begin{array}{l}
\{ \\
\quad \textbf{var}\ f\colon \textbf{proc}\ (\textbf{out}\ int) := \textbf{proc}\ (\textbf{out}\ x\colon int)\{\ \mathit{fix}(x);\ \}_x; \\
\quad \mathit{fix} := f; \\
\}_{\mathit{fix}}
\end{array}
$$

Une conséquence importante de ces choix de conception est de rendre possible des sémantiques remarquablement simples pour un langage impératif puisqu'il n'est alors pas nécessaire de modéliser les adresses (*locations*) : à une variable (mutable ou non) est directement associée une valeur. Ce type de sémantique particulièrement élégant semble avoir été oublié depuis [Donahue, 1977]. Nous présentons deux sémantiques opérationnelles pour Loop$^\omega$, une sémantique naturelle [Kahn, 1987] et une sémantique transitionnelle [Plotkin, 1981]. Nous montrons en particulier que la sémantique transitionnelle, qui raffine la sémantique naturelle, préserve bien le typage.

Une troisième sémantique est donnée par la traduction d'un programme impératif en programme fonctionnel du système T. Ce type de sémantique entre généralement dans le cadre des sémantiques dénotationnelles, mais nous montrons qu'en fixant la stratégie d'évaluation par valeur, on obtient une nouvelle sémantique opérationnelle pour Loop$^\omega$.

Plus précisément, la traduction $(s)^\star_{\vec{x}}$ d'un séquence s contenant les variables mutables $\vec{x}$, qui est définie par récurrence sur la syntaxe, est rappelée ci-dessous :

- $\bar{n}^\star = S^n(0)$
- $y^\star = y$
- $*^\star = ()$
- $(\textbf{proc}\ (\textbf{in}\ \vec{y}\,;\textbf{out}\ \vec{z}\,)\,\{s\}_{\vec{z}})^\star = \lambda\vec{y}\,.(s)^\star_{\vec{z}}$
- $(\varepsilon)^\star_{\vec{x}} = \vec{x}$
- $(\textbf{var}\ y := e;\ s)^\star_{\vec{x}} = (s)^\star_{\vec{x}}[e^\star/y]$
- $(\textbf{cst}\ y = e;\ s)^\star_{\vec{x}} = \textbf{let}\ y = e^\star\ \textbf{in}\ (s)^\star_{\vec{x}}$
- $(y{:=}e;\ s)^\star_{\vec{x}} = \textbf{let}\ y = e^\star\ \textbf{in}\ (s)^\star_{\vec{x}}$
- $(\textbf{inc}(y);\ s)^\star_{\vec{x}} = \textbf{let}\ y = \textbf{succ}(y)\ \textbf{in}\ (s)^\star_{\vec{x}}$
- $(\textbf{dec}(y);\ s)^\star_{\vec{x}} = \textbf{let}\ y = \textbf{pred}(y)\ \textbf{in}\ (s)^\star_{\vec{x}}$
- $(p(\vec{e}\,;\vec{z}\,);\ s)^\star_{\vec{x}} = \textbf{let}\ \vec{z} = p^\star\ \vec{e}^{\,\star}\ \textbf{in}\ (s)^\star_{\vec{x}}$
- $(\{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \textbf{let}\ \vec{z} = (s_1)^\star_{\vec{z}}\ \textbf{in}\ (s_2)^\star_{\vec{x}}$
- $(\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \textbf{let}\ \vec{z} = \textbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda\vec{z}\,.(s_1)^\star_{\vec{z}})\ \textbf{in}\ (s_2)^\star_{\vec{x}}$

Le résultat principal de cet article est un théorème de simulation : l'exécution d'un programme impératif (par le système de transition) est simulée pas-à-pas par les étapes de réécriture de son image fonctionnelle. L'énoncé formel est le suivant (où une mémoire μ associe une valeur à chaque variable mutable) :

Théorème 2. *Pour n'importe quel état* (s, μ)*, si* $\vec{x} = dom(\mu)$ *on a :*

$$(s, \mu) \mapsto (s', \mu') \text{ implique } (s)^{\star}_{\vec{x}}[\mu(\vec{x})^{\star}/\vec{x}] \rightsquigarrow (s')^{\star}_{\vec{x}}[\mu'(\vec{x})^{\star}/\vec{x}]$$

Il est important de remarquer que la mémoire mutable est directement simulée par la méta-substitution. Par conséquent, le langage fonctionnel cible ne nécessite aucune extension qui ne soit interprétable en termes de normalisation de preuve.

Comme corollaire de ce théorème, nous pouvons déduire la terminaison de l'évaluation des programmes de Loop$^{\omega}$. En fait, nous pouvons aller plus loin et montrer que la hiérarchie syntaxique des fragments T_n du Système T correspond à une hiérarchie de fragments Loop^n de Loop$^{\omega}$. Dans T_n, l'indice n correspond à l'ordre du type le plus élevé utilisé par un récurseur. Dans Loop^n, n correspond au niveau du type le plus élevé des variables mutables apparaissant dans le corps d'une boucle. Ce résultat généralise donc celui de [Crolard et al., 2006] où nous montrons comment traduire le langage Loop de Meyer et Ritchie dans T_0. De même, il est possible de programmer la fonction d'Ackermann, qui est représentable dans T_1 (mais pas dans T_0), dans Loop^1.

Voici un exemple simple qui montre l'utilisation des variables procédurales (mutables) pour implanter un itérateur d'ordre n :

```
cst iterator = proc(in n, p; out q) {
    q := proc(in x; out y) {
        y := x;
        for i := 0 until n {
            p(y; y);
        };
    };
};
```

Cette procédure *iterator* est typée statiquement **proc**(**in** *int*, **proc**(**in** σ; **out** σ); **out proc**(**in** σ; **out** σ)), pour un σ donné. Voici son image fonctionnelle exprimée dans la syntaxe de Standard ML :

```
val iterator = fn (n, p) =>
    let val q = fn x =>
            let val y = x
                val y = Rec(n, y, fn i => fn y =>
                    let val y = p y
                    in y end)
            in y end
    in q end
```

La version fonctionnelle de *iterator* est alors typée $int \times (\sigma^{\star} \to \sigma^{\star}) \to (\sigma^{\star} \to \sigma^{\star})$ où $\sigma^{\star}$ est le type fonctionnel correspondant à σ. Remarquez que c'est bien le type de y dans la boucle **for** qui détermine le type du récurseur dans la version fonctionnelle.

Dans cet article, nous montrons un résultat de complétude, à savoir que tout terme du système T peut être représenté par un programme de Loop$^{\omega}$. Plus précisément, nous définissons une traduction inverse $^{\diamond}$ telle que le couple $\langle ^{\diamond}, ^{\star} \rangle$ forme une rétraction (i.e. $^{\star} \circ ^{\diamond}$ est l'identité modulo convertibilité). Cette traduction inverse sera raffinée dans le troisième article.

Deux applications à la complexité, qui découlent de la simulation pas-à-pas, sont présentées dans cet article. La première est une caractérisation syntaxique de la classe des fonctions élémentaires de Csillag-Kalmar dérivée de celle décrite dans [Beckmann and Weiermann, 2000] pour le système T. La seconde est une preuve de la propriété *d'ultime obstination* pour Loop$^\omega$ dérivée de la même propriété pour le système T en appel par valeur de [Colson and Fredholm, 1998]. Cette propriété a pour conséquence en particulier l'absence de programme calculant le minimum de deux entiers n et m en temps $O(min(n,m))$.

Remarque. Les modèles de complexité associés aux langages fonctionnels (nombre d'étapes de réduction) ne sont en général pas réalistes, en particulier à cause de la présence requise du « ramasse-miette » (*garbage collector*). Il s'agit là d'un problème de fond, à savoir l'adéquation des langages fonctionnels aux machines de type Von Neumann. Il est possible de concevoir et d'implanter un tel ramasse-miettes qui n'utilise à tout moment qu'un multiple de l'espace maximum théoriquement nécessaire (j'ai présenté un tel algorithme dans [Crolard and Durand, 2000]). La question naturelle est alors de savoir s'il est possible d'implanter un ramasse-miettes qui libère la mémoire inutilisée *dès que possible*. Nous avons répondu à cette question négativement, sous la forme d'un résultat théorique sur l'impossibilité d'un ramasse-miette idéal (i.e. temps-réel à la fois en temps et en espace) dans [Beauquier et al., 2001].

2.2 Spécification formelle exécutable de Loop$^\omega$

Le système de types et la sémantique transitionnelle du langage Loop$^\omega$ ont depuis été spécifiés formellement avec Ott [Sewell et al., 2007] et Isabelle/HOL [Nipkow et al., 2002] (ce travail est décrit dans le rapport technique [Crolard and Polonowski, 2009]). En particulier, le système de types et la sémantique transitionnelle ont tous les deux été testés en utilisant l'extraction de programme dans Isabelle/HOL à partir de définition inductives [Berghofer and Nipkow, 2002]. Le programme qui calcule la fonction d'Ackermann donné dans [Crolard et al., 2009] est bien typé et se comporte comme attendu. La seule différence majeure entre la sémantique décrite dans l'article et celle spécifiée formellement concerne le traitement du passage de paramètres en **out**.

En effet, Ott ne gère pas encore l'α-conversion et nous avons donc reformulé le passage de paramètre en **out** en utilisant des alias explicites (afin de modéliser le passage par adresse) au lieu de la méta-substitution. Par contre, le passage de paramètres en **in** est implanté exactement comme décrit dans [Crolard et al., 2009] et s'appuie sur la substitution générée par Ott. Dans un travail futur, nous avons l'intention de développer la méta-théorie de Loop$^\omega$ dans Coq [Bertot and Castéran, 2004] en nous basant sur l'encodage *locally nameless* récemment ajouté à Ott.

3 Logique de programmes et mécanismes de contrôle

3.1 Présentation du troisième article

La traduction présentée dans la section précédente peut être vue comme une variante adaptée pour Loop$^\omega$ de la traduction définie de Landin pour un Algol idéalisé [Landin, 1964, Landin, 1965a, Landin, 1965b]. Pour prendre en compte les sauts non-locaux de Algol, Landin a été amené à étendre le λ-calcul avec un nouvel opérateur **J**. Cet opérateur, qui est décrit en détail dans [Landin, 1965c] (cf. aussi [Thielecke, 1998]), est en fait le père des opérateurs de contrôle que l'on trouve dans les langages fonctionnels (comme le fameux **call/cc** de Scheme [Kelsey et al., 1998] ou sa version typée présente dans l'extension SML/NJ de Standard ML [Harper et al., 1993]).

Il est donc tentant de revisiter le travail de Landin à la lumière de l'interprétation des preuves en logique classique comme des programmes fonctionnels avec opérateurs de contrôle. En effet, en composant cette interprétation avec la traduction de la section précédente, il est possible de dériver une « logique de programmes » pour une extension de Loop$^\omega$ avec des sauts non-locaux. Au-delà de l'exercice de style, la motivation est claire : il est notoirement difficile de définir une logique de programme cohérente pour un langage impératif avec procédures et sauts non-locaux [O'Donnell, 1982]. En revanche, une approche plus abordable consiste à ajouter des types dépendants à un langage impératif, puis traduire les dérivations de typage en dérivations de preuves. C'est l'objet du troisième article présenté dans ce mémoire.

Nous nous limitons dans un premier temps à l'arithmétique. Dans le cadre intuitionniste de l'arithmétique de Heyting, le langage fonctionnel correspondant est le Système T de Gödel. Dans le cadre classique de l'arithmétique de Peano, il faut étendre le langage avec des opérateurs de contrôle [Murthy, 1991]. Nous utiliserons ici une formulation différente de l'arithmétique décrite dans [Leivant, 1990, Leivant, 2002] et, dans le cadre de la logique de second ordre, dans [Krivine and Parigot, 1990]. L'avantage de cette formulation est en particulier de nous dispenser des codages dans les formules pour raisonner sur les programmes fonctionnels.

Nous ne considérons pas non plus de sémantique opérationnelle directe des opérateurs de contrôle, mais uniquement une sémantique indirecte par traduction en « style passage de continuation » (CPS). Comme il est remarqué dans [Murthy, 1991], cette CPS simule un appel par valeur et correspond à la traduction de Kuroda au niveau des types dépendants [Kuroda, 1951]. Toutefois, nous profitons du fait que les termes à traduire proviennent des programmes impératifs, et qu'ils sont déjà en forme normale monadique [Hatcliff and Danvy, 1994] (aussi appelée *A-normal form* [Flanagan et al., 1993]).

Du côté impératif, nous étendons le langage Loop$^\omega$ avec des types dépendants du premier ordre. Une des conséquences de cette extension a été de nous amener à assouplir les systèmes de types statiques non-dépendants sous-jacents. En effet, après une affectation $x := 0$, le type de x doit être $\mathbf{nat}(0)$, donc le type de x est changé par l'affectation si x était non nul avant. En fait, le type de x avant l'affectation n'a aucune importance : il n'est même pas nécessaire que x soit de type entier. En généralisant cette idée, nous avons obtenu un système de types non-dépendants, que nous avons appelé « pseudo-dynamique », où le type de chaque variable peut être changé par une affectation (ou un appel de procédure). Bien que cela ressemble à une caractéristique des langages dynamiques, ce système de types reste complètement statique.

Pour établir une comparaison avec les types dépendants fonctionnels, rappelons qu'un état mutable peut être simulé par une transformation en « style passage d'état » (*state passing style*). Cette transformation peut être factorisée en une traduction en style monadique [Liang et al., 1995], puis en instanciant la monade par la monade d'état habituelle $\tau\ ST = \sigma \to (\tau \times \sigma)$, où σ est le type fixé de l'état mutable. Si l'on autorise des mutations du type de l'état, on obtient une monade d'état paramétrée [Atkey, 2006], $(\sigma, \tau, \sigma')\ ST = \sigma \to (\tau \times \sigma')$ où σ est le type de l'état global en entrée et σ' est son type en sortie.

Cette comparaison montre que le système de types pseudo-dynamique est très expressif et qu'il permet en fait de typer des programmes impératifs qui nécessitent habituellement des systèmes à effets [Talpin and Jouvelot, 1994]. De plus, l'intuition logique de ce système, proche de la logique de Floyd-Hoare, est parfaitement claire (nous reviendrons sur cette question en section 3.3). Un autre avantage de ce système est qu'il peut inclure naturellement la vérification de l'initialisation des variables. Cette vérification est généralement réalisée dans les langages impératifs par un algorithme ad-hoc, indépendant du système de types. Nous pouvons initialiser toutes les variables à l'unique valeur de type **unit** (ou $\top$ dans la version avec types dépendants). Remarquez que dans un système de types dépendants, il n'existe pas forcément de valeur (i.e. preuve) pour un type dépendant donné (i.e. une formule).

Voici un résumé des différents résultats présentés dans cet article, qui trouvent leur place dans le cadre formel décrit au début de cette introduction.

- Le langage fonctionnel **F**, qui correspond à notre formulation du système T de Gödel, est muni de deux systèmes de types, un système de types simple **FS** et un système de types dépendants **FD** similaire au système **M1LP** [Leivant, 1990]. En particulier, les types dépendants incluent les formules de l'arithmétique du premier ordre.
- Le langage impératif **I** est essentiellement le langage $\textsc{Loop}^\omega$. Ce langage **I** est aussi muni de deux systèmes de types inhabituels : un système de types pseudo-dynamique **IS** et un système de types dépendants **ID**.
- Nous montrons que la traduction * de **I** vers **F** préserve le typage dans les cadres dépendants et non-dépendants. Nous remarquons aussi que les images des programmes impératifs sont en forme monadique, sur lesquels il est possible de redéfinir la traduction inverse $^\diamond$. Le couple $\langle {}^\diamond, {}^\star \rangle$ forme toujours une rétraction (et non un isomorphisme), mais la traduction inverse est plus fine que celle de la section précédente. En composant la mise en forme normale monadique et la traduction inverse $^\diamond$, il est alors possible de générer un programme impératif à partir d'une preuve quelconque d'une formule de l'arithmétique.
- $\mathbf{F}^c$ est défini comme l'extension de **F** avec les opérateurs de contrôle **callcc** et **throw** (tirés de [Harper et al., 1993]). Le sémantique de $\mathbf{F}^c$ est donnée par une transformation CPS par valeur vers **F**. Cette transformation est factorisée en utilisant le méta-langage de Moggi [Moggi, 1990, Moggi, 1991]. Le fait que l'image d'un programme impératif soit déjà en forme normale monadique nous épargne alors la première étape de la transformation.
- Nous définissons alors $\mathbf{I}^c$ comme l'extension **I** obtenue en se donnant deux *procédures* primitives **callcc** et **throw**. Bien que ces primitives soient inhabituelles dans un langage impératif, elles permettent d'encoder des mécanismes de contrôle plus conventionnels. Remarquez toutefois qu'il n'est bien sûr pas possible d'encoder un **goto** général puisque notre langage est total.

3.2 Continuations délimitées

Comme exemple générique, nous montrons comment encoder les opérateurs **shift** et **reset** [Danvy and Filinski, 1989] qui permettent de manipuler des continuations délimitées. Ces opérateurs permettent à leur tour d'encoder n'importe quelle monade représentable [Filinski, 1994]. Nous utilisons ici l'implémentation de **shift** et **reset** utilisant un état global et **callcc** et **throw** décrite dans [Filinski, 1994]. Nous obtenons donc indirectement un système de types dépendants pour les continuation délimitées.

Avant de présenter les encodages de **shift** et **reset** impératifs, il peut être utile d'en donner une intuition au niveau du typage fonctionnel [Wadler, 1994]. La sémantique de ces opérateurs est donnée par une double transformation en CPS [Danvy and Filinski, 1989]. En effet, une simple transformation en CPS ne suffit pas à obtenir un terme dont l'évaluation soit indépendante de la stratégie. La première des deux transformation correspond à une monade de continuation paramétrée [Atkey, 2008] :

$$M(\alpha, \beta, \gamma) = (\gamma \to \beta) \to \alpha$$

La seconde transformation correspond à la monade de continuation habituelle, pour un type *output* o fixé :

$$\nabla\sigma = (\sigma \to o) \to o$$

La composition des deux transformations [Liang et al., 1995] nous donne donc la monade paramétrée suivante :

$$\begin{array}{rcl}(\gamma \to \nabla\beta) \to \nabla\alpha & \cong & ((\gamma \times (\beta \to o)) \to o) \to ((\alpha \to o) \to o) \\ & \cong & (\alpha \to o) \to (((\gamma \times (\beta \to o)) \to o) \to o) \\ & = & (\alpha \to o) \to \nabla(\gamma \times (\beta \to o))\end{array}$$

On reconnaît ici le transformateur de monade d'état paramétrée instancié par la monade de continuation. Cette monade correspond exactement à la transformation obtenue en composant la transformation par passage d'état, où l'état global est une continuation, avec une transformation en CPS. En d'autres termes, il est équivalent (modulo isomorphisme de type) de faire une transformation par passage d'état (où l'état global est une continuation) ou une transformation en CPS (paramétrée). C'est cette équivalence qui est exploitée dans [Filinski, 1994] pour encoder **shift** et **reset** avec un état global (contenant une continuation) et **callcc**/**throw**.

Pour traduire cet encodage en impératif, rappelons tout d'abord que les systèmes de types **IS** et **ID** interdisent l'accès aux variables mutables non-locales. La technique habituelle pour contourner cette restriction consiste à passer l'état global explicitement en paramètre **in out** à chaque appel de procédure. Un paramètre en **in out** peut lui même être simulé par un paramètre en **in** et un paramètre en **out** (initialisé à la valeur du paramètre en **in**). Pour alléger la syntaxe, nous introduisons donc les abréviations suivantes, où $\vec{z}$ est une liste de variables mutables :

$$\begin{array}{rcl}\textbf{proc}(\textbf{in}\ \vec{x}; \textbf{out}\ \vec{y})_{\vec{z}}\{s\}_{\vec{y},\vec{z}} & = & \textbf{proc}(\textbf{in}\ \vec{x}, \vec{z}\,'; \textbf{out}\ \vec{y}, \vec{z})\{\vec{z} := \vec{z}\,'; s\}_{\vec{y},\vec{z}} \\ p(\vec{e}; \vec{y})_{\vec{z}} & = & p(\vec{e}, \vec{z}; \vec{y}, \vec{z})\end{array}$$

L'implantation de **shift** et **reset** utilisant un état global et **callcc**/**throw** peut alors être traduite presque mécaniquement en impératif et on obtient :

$$\begin{array}{rcl}\textbf{reset} & : & \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \neg\alpha; \textbf{out}\ \beta, \neg\beta), \neg\gamma; \textbf{out}\ \alpha, \neg\gamma) \\ \textbf{reset} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{ \\ & & \quad k\colon\{ \\ & & \qquad \textbf{cst}\ m = mk; \\ & & \qquad mk := \textbf{proc}(\textbf{in}\ r; \textbf{out}\ z)\{\textbf{jump}\ (k, r, m)_z;\}_z; \\ & & \qquad \textbf{var}\ y;\ p(; y)_{mk}; \\ & & \qquad \textbf{jump}\ (mk, y)_{r,mk}; \\ & & \quad \}_{r,mk}; \\ & & \}_{r,mk};\end{array}$$

$$\begin{array}{rcl}\textbf{shift} & : & \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \alpha, \neg\beta; \textbf{out}\ \gamma, \neg\beta), \neg\delta; \textbf{out}\ \epsilon, \neg\epsilon), \neg\delta; \textbf{out}\ \alpha, \neg\gamma) \\ \textbf{shift} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{ \\ & & \quad k\colon\{ \\ & & \qquad \textbf{proc}\ q(\textbf{in}\ v; \textbf{out}\ r)_{mk}\{ \\ & & \qquad\quad \textbf{reset}\ (\textbf{proc}(\textbf{out}\ z)_{mk}\{\textbf{jump}\ (k, v, mk)_{z,mk};\}_{z,mk}; r); \\ & & \qquad \}_{r,mk}; \\ & & \qquad \textbf{var}\ y;\ p(q; y)_{mk}; \\ & & \qquad \textbf{jump}\ (mk, y)_{r,mk}; \\ & & \quad \}_{r,mk}; \\ & & \}_{r,mk}\end{array}$$

Bien entendu, les images de ces procédures par la traduction * produit des termes fonctionnels typables dans $\mathbf{FD}^c$. Leur types fonctionnels (propositionnels) sont les suivants:

$$\begin{aligned} \mathit{reset} &: (\neg\alpha \Rightarrow \beta \wedge \neg\beta) \wedge \neg\gamma \Rightarrow \alpha \wedge \neg\gamma \\ \mathit{shift} &: ((\alpha \wedge \neg\beta \Rightarrow \gamma \wedge \neg\beta) \wedge \neg\delta \Rightarrow \epsilon \wedge \neg\epsilon) \wedge \neg\delta \Rightarrow \alpha \wedge \neg\gamma \end{aligned}$$

Nous reconnaissons bien les types de ces opérateurs donnés dans [Danvy and Filinski, 1989] où $(\alpha \wedge \neg\sigma) \Rightarrow (\beta \wedge \neg\tau)$ est noté sous la forme $\alpha/\tau \to \beta/\sigma$. Une étude détaillée de ces opérateurs du point de vue de la théorie des types non-dépendants est présentée dans [Ariola et al., 2007].

Voici un exemple élémentaire d'utilisation de **shift** et **reset** présenté dans [Wadler, 1994].

$$1 + (\mathbf{reset}\,(10 + (\mathbf{shift}\,f\,.\,(f\,(f\;100)))))$$

Informellement, ce terme vaut 121 puisque f est associée à la continuation, délimitée par le **reset**, qui ajoute 10 à son argument. Nous pourrions évaluer ce terme en utilisant les règles de calcul de la sémantique contextuelle décrite dans [Danvy and Filinski, 1989]. Nous allons plutôt ici montrer statiquement cette propriété en utilisant le système de types dépendants.

Le terme ci-dessus mis en forme monadique, puis traduit dans la syntaxe impérative, s'écrit alors :

```
var R := *;
cst Add = proc(in X, Y; out Z)_mk {
    Z := X;
    for I := 0 until Y {
        inc(Z);
    };
};
cst P = proc(in F; out Y)_mk {
    F(100; Y)_mk;
    F(Y; Y)_mk;
};
cst Q = proc(; out R)_mk {
    var Z := *;
    shift(P; Z)_mk;
    Add(Z, 10; R)_mk;
};
k: {
    var mk := k;
    reset(Q; R)_mk;
};
inc(R);
```

Les types dépendants des procédures intermédiaires du programme ci-dessus ainsi que des occurrences de *shift* et *reset* sont les suivants :

$$
\begin{array}{rcl}
\mathit{Add} & : & \forall m,x,y(\mathbf{nat}(x)\wedge\mathbf{nat}(y)\wedge\neg\mathbf{nat}(m)\Rightarrow(\mathbf{nat}(x{+}y)\wedge\neg\mathbf{nat}(m)))\\
P & : & \forall n(\mathbf{nat}(n)\wedge\neg\mathbf{nat}(120)\Rightarrow(\mathbf{nat}(n{+}10)\wedge\neg\mathbf{nat}(120)))\wedge\neg\mathbf{nat}(120)\Rightarrow(\mathbf{nat}(120)\wedge\neg\mathbf{nat}(120))\\
Q & : & \neg\mathbf{nat}(120)\Rightarrow\exists n(\mathbf{nat}(n{+}10)\wedge\neg\mathbf{nat}(n{+}10))\\
\mathit{shift} & : & \forall n(\mathbf{nat}(n)\wedge\neg\mathbf{nat}(120)\Rightarrow(\mathbf{nat}(n{+}10)\wedge\neg\mathbf{nat}(120)))\wedge\neg\mathbf{nat}(120)\\
 & & \quad\Rightarrow(\mathbf{nat}(120)\wedge\neg\mathbf{nat}(120)))\wedge\neg\mathbf{nat}(120)\Rightarrow\exists n(\mathbf{nat}(n)\wedge\neg\mathbf{nat}(n{+}10))\\
\mathit{reset} & : & (\neg\mathbf{nat}(120)\Rightarrow\exists n(\mathbf{nat}(n{+}10)\wedge\neg\mathbf{nat}(n{+}10)))\wedge\neg\mathbf{nat}(120)\Rightarrow(\mathbf{nat}(120)\wedge\neg\mathbf{nat}(120))
\end{array}
$$

Le résultat final R est bien typable de type $\mathbf{nat}(121)$. Remarquez aussi que le type (dépendant) de **shift** attribue bien le type $\forall n(\mathbf{nat}(n)\Rightarrow\mathbf{nat}(n\,+\,10))$ à f si l'on ignore le type de la variable globale *mk*. Toutefois, cet exemple n'exploite pas toute l'expressivité du système de types, et le type de *mk* est presque toujours identique. Il sera intéressant d'étendre ce cadre formel en particulier aux listes pour prouver, par exemple, la correction des fonctions qui calculent les préfixes d'une liste [Danvy, 1989].

Remarque. Les systèmes de types dépendants impératif et fonctionnel, ainsi que la traduction, ont été implantés (sous forme de spécifications exécutables) et les exemples ci-dessus ont été vérifiés mécaniquement avec Ott [Sewell et al., 2007] et Twelf [Pfenning and Schürmann, 1999]. Ces spécifications formelles exécutables sont décrites complètement dans le rapport technique [Crolard, 2009].

3.3 Logique de Floyd-Hoare

Les systèmes de types dépendants sont inhabituels pour les langages impératifs pour lesquels les systèmes de preuve de programmes prennent généralement la forme d'une logique de Floyd-Hoare. Afin de fixer les idées, un exemple concret de langage impératif dont il est question ici, et qui est utilisé dans le monde industriel, est Spark Ada [Barnes, 2003] (il s'agit d'un sous-ensemble de Ada dont la syntaxe a été restreinte pour rendre la sémantique non-ambiguë et permettre ainsi la preuve de programme).

Nous montrons ici comment reformuler de manière systématique le système de types dépendants en une logique qui manipule des triplets de Hoare. L'objectif est à la fois théorique et pratique. Du point de vue théorique, il est nécessaire pour permettre une comparaison avec les nombreuses logiques de Floyd-Hoare existant dans la littérature. Du point de vue pratique, il permet d'étudier la possibilité d'ajouter une « règle de la conséquence » au système de types dépendants. C'est cette règle qui permet de justifier les techniques de « génération d'obligations de preuves » qui rendent les logiques de Floyd-Hoare confortables à utiliser, puisqu'elle permet de dissocier la partie preuve de programme de la partie purement logique.

Passer à un système de types dépendants qui manipule des triplets de Hoare est relativement direct. En effet, il suffit de se donner une variable globale *assert* et de considérer que le programme à prouver a été transformé en *state-passing style*. Par conséquent, toute séquence s sera typée par un séquent de la forme $\Gamma;\Omega,\mathit{assert}\colon\varphi^{\diamond}\vdash s\rhd\exists\vec{\jmath}\,.\Omega',\mathit{assert}\colon\psi^{\diamond}$ et toute expression e sera typée par un séquent de la forme $\Gamma;\Omega,\mathit{assert}\colon\varphi^{\diamond}\vdash e\colon\psi$. En introduisant la notation habituelle (qui masque ici le nom *assert* de la variable globale), on obtient les jugements $\Gamma;\Omega\vdash\{\varphi\}s\rhd\exists\vec{\jmath}\,.\Omega'\{\psi\}$ pour typer une séquence et $\Gamma;\Omega\vdash\{\varphi\}e\colon\psi$ pour typer une expression et on peut alors dériver le système de déduction présenté en Figure 2.

Remarque. La boucle **for** $y:=0$ **until** e $\{\varphi\}\{s\}_{\vec{x}}$ mentionne maintenant naturellement un *invariant de boucle* φ qui correspond au type de l qui annote le corps de la boucle dans la version en state-passing style : **for** $y:=0$ **until** e $\{s\}_{\vec{x},l:\varphi}$. De même, les pré-conditions et post-conditions d'un prototype expriment simplement le type de l en entrée et en sortie de la procédure.

$$\frac{x{:}\,\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash \{\varphi\}x{:}\,\tau} \quad (\textsc{t.env})$$

$$\frac{}{\Gamma;\Omega \vdash \{\varphi\}\bar{q}{:}\,\mathbf{nat}(\mathbf{s}^q(\mathbf{0}))} \quad (\textsc{t.num})$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma;\Omega \vdash \{\varphi\}*{:}\,n = m} \quad (\textsc{t.equal})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}{:}\,\vec{\sigma}; \vec{z}{:}\,\vec{\top} \vdash \{\varphi'\}s \rhd \exists \vec{\jmath}\,.\vec{z}{:}\,\vec{\tau}\,\{\psi'\}}{\Gamma;\Omega \vdash \{\varphi\}\mathbf{proc}\ (\mathbf{in}\ \vec{y};\ \mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}}{:}\,\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,\{\varphi'\}; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,\{\psi'\})} \quad (\textsc{t.proc})^*$$

$$\frac{\Gamma;\Omega \vdash \{\varphi\}e'{:}\,\tau[n/i] \qquad \Gamma;\Omega \vdash \{\varphi\}e{:}\,n = m}{\Gamma;\Omega \vdash \{\varphi\}e'{:}\,\tau[m/i]} \quad (\textsc{t.subst-i})$$

$$\frac{\Gamma;\Omega \vdash \{\varphi\}s \rhd \exists \vec{\kappa}\,.\Omega'[n/i] \qquad \Gamma;\Omega \vdash \{\varphi\}e{:}\,n = m}{\Gamma;\Omega \vdash \{\varphi\}s \rhd \exists \vec{\kappa}\,.\Omega'[m/i]\{\varphi\}[m/i]} \quad (\textsc{t.subst-ii})$$

$$\frac{}{\Gamma;\Omega, \Omega'[\vec{n}\,/\vec{\kappa}] \vdash \{\varphi\}\varepsilon \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}} \quad (\textsc{t.empty})$$

$$\frac{\Gamma;\Omega, \vec{x}{:}\,\vec{\sigma} \vdash \{\varphi\}c \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\tau}\,\{\psi\} \qquad \Gamma;\Omega, \vec{x}{:}\,\vec{\tau} \vdash \{\psi\}s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}}{\Gamma;\Omega, \vec{x}{:}\,\vec{\sigma} \vdash \{\varphi\}c;\ s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}} \quad (\textsc{t.seq})^*$$

$$\frac{\Gamma;\Omega \vdash \{\varphi\}e{:}\,\tau \qquad \Gamma, y{:}\,\tau;\Omega\{\varphi\} \vdash s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}}{\Gamma;\Omega \vdash \{\varphi\}\mathbf{cst}\ y = e;\ s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}} \quad (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma;\Omega, y{:}\,\tau \vdash \{\varphi\}s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\} \qquad y \notin \Omega'}{\Gamma;\Omega \vdash \{\varphi\}\mathbf{var}\ y := e;\ s \rhd \exists \vec{\kappa}\,.\Omega'\{\varphi'\}} \quad (\textsc{t.var})$$

$$\frac{\Gamma;\vec{x}{:}\,\vec{\tau} \vdash \{\varphi\}s \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\sigma}\,\{\varphi'\}}{\Gamma;\Omega, \vec{x}{:}\,\vec{\tau} \vdash \{\varphi\}\,\{s\}_{\vec{x}} \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\sigma}\,\{\varphi'\}} \quad (\textsc{t.block})$$

$$\frac{}{\Gamma;\Omega, y{:}\,\mathbf{nat}(n) \vdash \{\varphi\}\mathbf{inc}(y) \rhd y{:}\,\mathbf{nat}(\mathbf{s}(n))\{\varphi\}} \quad (\textsc{t.inc})$$

$$\frac{}{\Gamma;\Omega, y{:}\,\mathbf{nat}(n) \vdash \{\varphi\}\mathbf{dec}(y) \rhd y{:}\,\mathbf{nat}(\mathbf{p}(n))\{\varphi\}} \quad (\textsc{t.dec})$$

$$\frac{\Gamma;\Omega, y{:}\,\sigma \vdash \{\varphi\}e{:}\,\tau}{\Gamma;\Omega, y{:}\,\sigma \vdash \{\varphi\}y := e \rhd \Omega, y{:}\,\tau\{\varphi\}} \quad (\textsc{t.assign})$$

$$\frac{\Gamma;\Omega, \vec{x}{:}\,\vec{\sigma}\,[0/i] \vdash \{\varphi[0/i]\}e{:}\,\mathbf{nat}(n) \qquad \Gamma, y{:}\,\mathbf{nat}(i); \vec{x}{:}\,\vec{\sigma} \vdash \{\varphi\}s \rhd \vec{x}{:}\,\vec{\sigma}\,[\mathbf{s}(i)/i]\{\varphi[\mathbf{s}(i)/i]\}}{\Gamma;\Omega, \vec{x}{:}\,\vec{\sigma}\,[0/i] \vdash \{\varphi[0/i]\}\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{\varphi\}\ \{s\}_{\vec{x}} \rhd \vec{x}{:}\,\vec{\sigma}\,[n/i]\{\varphi[n/i]\}} \quad (\textsc{t.for})^*$$

$$\frac{\Gamma;\Omega, \vec{r}{:}\,\vec{\omega} \vdash \{\varphi[\vec{u}\,/\vec{\imath}\,]\}p{:}\,\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}\,\{\varphi\}; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,\{\varphi'\}) \qquad \Gamma;\Omega, \vec{r}{:}\,\vec{\omega} \vdash \{\varphi[\vec{u}\,/\vec{\imath}\,]\}\vec{e}{:}\,\vec{\sigma}\,[\vec{u}\,/\vec{\imath}\,]}{\Gamma;\Omega, \vec{r}{:}\,\vec{\omega} \vdash \{\varphi[\vec{u}\,/\vec{\imath}\,]\}p(\vec{e}\,;\vec{r}\,) \rhd \exists \vec{\jmath}\,.\vec{r}{:}\,\vec{\tau}\,[\vec{u}\,/\vec{\imath}\,]\{\varphi'[\vec{u}\,/\vec{\imath}\,]\}} \quad (\textsc{t.call})$$

*where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$ in (T.PROC) and $i \notin \mathcal{FV}(\Gamma)$ in (T.FOR)
and $\vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega')$ in (T.SEQ)

Figure 2. Système de types dépendants avec triplets de Hoare

D'autres transformations syntaxiques sont envisageables pour isoler la partie logique présente dans les types et se rapprocher d'une formulation plus traditionnelle de la logique de Hoare. Toutefois, une question plus délicate concerne la dérivation de la fameuse *règle de la conséquence* :

$$\frac{\Gamma, \Omega \vdash \varphi' \Rightarrow \varphi \qquad \Gamma;\Omega \vdash \{\varphi\}s \rhd \exists \vec{\jmath}\,.\Omega'\{\psi\} \qquad \Gamma, \Omega \vdash \forall \vec{\jmath}\,(\psi \Rightarrow \psi')}{\Gamma;\Omega \vdash \{\varphi'\}s \rhd \exists \vec{\jmath}\,.\Omega'\{\psi'\}}$$

La difficulté ici est de déterminer si le contenu calculatoire des preuves des séquents $\Gamma, \Omega \vdash \varphi' \Rightarrow \varphi$ et $\Gamma, \Omega \vdash \forall \vec{\jmath}\,(\psi \Rightarrow \psi')$ risque d'interférer avec le programme. La situation dépend de la présence ou non de sauts non locaux et donc de la logique utilisée pour valider le programme.

En logique intuitionniste

Si l'on suppose que les formules utilisées pour les annotations sont sans contenu calculatoire (le type non-dépendent sous-jacent à la formule est isomorphe à **unit**) alors l'image de la règle de la conséquence par la traduction * est dérivable et le contenu calculatoire (trivial) de $\varphi' \Rightarrow \varphi$ et $\psi \Rightarrow \psi'$ peut-être synthétisé.

En logique classique

Du point de vue de la logique classique la situation est complètement différente : il n'est plus possible de s'appuyer sur la forme syntaxique d'une formule pour déterminer si elle possède un contenu calculatoire. Cela se comprend en particulier en examinant la forme de la $\neg_o\neg_o$-traduction d'une formule quelconque : $(\varphi^\circ \Rightarrow o) \Rightarrow o$ possède toujours un contenu calculatoire (puisque o, le type en sortie du programme, possède un contenu calculatoire).

Voici un exemple particulièrement contre-intuitif de programme impératif avec saut non-local pour lequel tout le contenu calculatoire se trouve dans la « partie logique » que l'on efface habituellement en logique intuitionniste.

```
cst Axiom = proc(in N, C; out Z) {
    var P := proc(in C; out Z) {
        Z := ∗;
    };
    for i := 0 until N {
        P := proc(in C; out Z) {
            cst Q = proc(; out Z) {
                Z := ∗;
            };
            jump(C, Q);
        };
    };
    P(C; Z);
};
cst Sign0 = proc(in N, C; out Z) {
    var A := ∗;
    Axiom(N, C; A);
    Z := 0;
};
cst SignS = proc(in N, C; out Z) {
    var A := ∗;
    C(; A);
    Z := 1;
};
cst TryCatch = proc(in T, U; out R) {
    R := proc(in N; out Z) {
        W: {
```

```
            cst F = proc(in A; out B) {
                var Aux := ∗;
                U(N, A; Aux);
                W(Aux; B);
            };
            T(N, F; Z);
        };
    };
};
var Sign := ∗;
TryCatch(Sign0, SignS; Sign);
var X := ∗;
Sign(0; X);
```

Ce programme détermine en fait simplement le « signe » d'un entier naturel où la définition équationnelle de la fonction *sign* est la suivante :

$$\begin{aligned} \mathit{sign}(0) &= 0 \\ \mathit{sign}(S(n)) &= 1 \end{aligned}$$

Les procédures auxiliaires ci-dessus sont typables avec les types dépendants suivants, où *TryCatch* implante une variante dépendante du tiers-exclu :

$$\begin{aligned} \mathit{Axiom} &: \forall n(\mathbf{nat}(n) \wedge \neg\exists m(n = s(m)) \Rightarrow (n = 0))) \\ \mathit{Sign0} &: \forall n(\mathbf{nat}(n) \wedge \neg\exists m(n = s(m)) \Rightarrow \mathbf{nat}(\mathit{sign}(n)))) \\ \mathit{SignS} &: \forall n(\mathbf{nat}(n) \wedge \exists m(n = s(m)) \Rightarrow \mathbf{nat}(\mathit{sign}(n)))) \\ \mathit{TryCatch} &: \forall n(\mathbf{nat}(n) \wedge \varphi \Rightarrow \psi) \wedge \forall n(\mathbf{nat}(n) \wedge \neg\varphi \Rightarrow \psi)) \Rightarrow \forall n(\mathbf{nat}(n) \Rightarrow \psi) \\ \mathit{Sign} &: \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathit{sign}(n)) \end{aligned}$$

Remarquez que le seul test à zéro réalisé (encodé par la boucle **for**) dans le programme ci-dessus se trouve dans la procédure *Axiom*, que l'on considère habituellement sans contenu calculatoire en logique intuitionniste. Pourtant si l'on supprime le contenu calculatoire de *Axiom*, le programme ci-dessus devient incorrect. La règle « coupable » dans le système de types fonctionnel dépendant, valide en logique intuitionniste mais pas en logique classique, est la suivante :

$$\frac{\Gamma \vdash t : (n = m)}{\Gamma \vdash () : (n = m)}$$

Cette règle est toutefois valide en logique classique si t est une valeur [Makarov, 2006]. Cette propriété est vérifiée implicitement dans notre système de types dépendants impératif puisque les seules expressions impératives sont des valeurs.

Une solution élégante au problème consistant à discriminer les formules sans contenu calculatoire en logique classique a été proposée récemment dans [Thielecke, 2008] sous la forme d'une logique modale classique. La modalité est interprétée calculatoirement en termes de masquage des effets. Nous travaillons actuellement sur une extension de ce système avec des types dépendants afin d'introduire la règle de la conséquence dans notre système.

4 Coroutines et logique soustractive

Dans cette section, nous revenons sur l'interprétation calculatoire de la logique soustractive en termes de coroutines. La notion de coroutine est habituellement attribuée à Conway [Conway, 1963] qui l'a introduite dans le cadre de l'analyse lexicale et syntaxique pour simplifier la coopération entre l'analyse lexicale et syntaxique réalisée par un compilateur. Elles sont aussi utilisées par Knuth dans [Knuth, 1973] qui les considère alors comme un mécanisme qui généralise les sous-routines (procédures sans paramètres). Les coroutines sont d'abord apparues dans le langage Simula-67 [Dahl and Nygaard, 1966]. Un cadre formel pour prouver la correction de programmes contenant des coroutines simples à même été développé dans [Clint, 1973] et [Dahl, 1975]. Les coroutines ont ensuite ont été reprises, entre autres, dans Modula-2 [Wirth and Mincer-Daszkiewicz, 1980], Icon [Griswold and Griswold, 1983] et plus récemment dans le langage fonctionnel Lua [de Moura et al., 2004].

Dans sa thèse, Marlin [Marlin, 1980] résume les caractéristiques d'une coroutine comme suit :

1. *the values of data local to a coroutine persist between successive occasions on which control enters it (that is, between successive calls), and*

2. *the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.*

Autrement dit, une coroutine est une sous-routine *avec un état local* dont on peut suspendre et reprendre l'exécution. Cette définition informelle ne suffit bien entendu pas à capturer les diverses implantations qui ont été réalisées en pratique. En particulier, dans [de Moura and Ierusalimschy, 2004] les différences principales entre les mécanismes de coroutines sont résumées ainsi :

- *the control-transfer mechanism, which can provide symmetric or asymmetric coroutines.*

- *whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs;*

- *whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from with nested calls.*

Des coroutines symétriques proposent en général une seule opération de transfert de contrôle qui permet aux coroutines de se passer le contrôle entre elles. Un mécanisme de contrôle asymétrique, aussi appelées semi-coroutines [Dahl et al., 1972], comporte deux primitives de transfert de contrôle : la première pour invoquer une coroutine, la seconde pour la suspendre et rendre le contrôle à l'appelant. Alors que les coroutines symétriques s'exécutent au même niveau hiérarchique, une coroutine asymétrique est en quelque sorte subordonnée à son appelant, de la même manière que pour un appel de sous-routine.

Un exemple célèbre qui illustre le troisième point ci-dessus, appelé « the same fringe problem », consiste à déterminer si deux arbres possèdent la même séquence de feuilles en utilisant deux coroutines, où chaque coroutine parcourt un arbre récursivement et passe la main à l'autre coroutine lorsqu'elle rencontre une feuille. L'élégance de cet algorithme vient du fait que chaque coroutine possède sa propre pile, ce qui autorise des parcours d'arbre récursifs.

Les coroutines asymétriques correspondent souvent au mécanisme de coroutines rendu directement accessible au programmeur (comme dans Simula ou Lua), parfois sous la forme restreinte de générateurs (comme dans C#). Par contre, les coroutines symétriques sont généralement choisies comme primitives pour implanter des mécanismes de concurrence plus évolués (comme dans Modula). C'est le cas en particulier dans le standard Unix [The Open Group, 1997] où les primitives fournies pour implanter des bibliothèques de processus légers (*threads*) utilisateurs sont **getcontext**, **setcontext**, **swapcontext** et **makecontext**. C'est la terminologie que nous avons adopté dans l'article 1, et voici l'extrait de la spécification qui les décrit :

- *int* **getcontext**(*ucontext_t *ucp*);

  The **getcontext**() function saves the current thread's execution context in the structure pointed to by ucp. This saved context may then later be restored by calling **setcontext**().

- *int* **setcontext**(const *ucontext_t *ucp*);

  The **setcontext**() function makes a previously saved thread context the current thread context, i.e., the current context is lost and **setcontext**() does not return. Instead, execution continues in the context specified by *ucp*, which must have been previously initialized by a call to **getcontext**(), **makecontext** [...] If *ucp* was initialized by **getcontext**(), then execution continues as if the original getcontext() call had just returned (again). If *ucp* was initialized by **makecontext**, execution continues with the invocation of the function specified to **makecontext**. [...]

- *int* **swapcontext**(*ucontext_t *oucp*, const *ucontext_t *ucp*);

  The **swapcontext**() function saves the current thread context in **oucp* and makes **ucp* the currently active context.

- *void* **makecontext**(*ucontext_t *ucp*, *void* (**func*)(), *int argc*, ...);

  The **makecontext**() function shall modify the context specified by *ucp*, which has been initialized using **getcontext**(). When this context is resumed using **swapcontext**() or **setcontext**(), program execution shall continue by calling *func*, passing it the arguments that follow *argc* in the **makecontext**() call. Before a call is made to **makecontext**(), the application shall ensure that the context being modified has a stack allocated for it. [...]

Remarque. La sémantique des trois premières primitives est relativement claire. Une utilisation typique de **makecontext** pour créer une coroutine est décrite dans [Engelschall, 2000] : il faut créer d'abord une nouvelle pile (environnement) sur laquelle sont placés les arguments qui seront donnés en argument à *func* lorsque la nouvelle coroutine sera invoquée par **setcontext**.

4.1 Coroutines fonctionnelles

Revenons sur l'interprétation calculatoire du $\lambda\mu$-calcul sûr et rappelons tout d'abord que les opérateurs de contrôle comme **call/cc** permettent en particulier de simuler un mécanisme de coroutines. Une implantation en Scheme, utilisant le **call/cc**, peut être trouvée dans [Friedman et al., 1984, Friedman et al., 1986]. Cette approche a été étendue dans le Standard ML du New Jersey (SML/NJ) pour fournir une implantation simple et élégante des processus légers (*threads*). Remarquez que les coroutines correspondent à un mécanisme de concurrence coopérative, où chaque *thread* (coroutine) demande explicitement à être suspendu [Reppy, 1988, Reppy, 1989, Reppy, 1995, Ramsey, 1990] (pour des implantations complètes du temps partagé, comprenant donc un ordonnanceur préemptif, cf. [Cooper and Morrisett, 1990, Reppy, 1990]).

Ce qu'il est important de noter concernant cette implantation des coroutines, c'est que des opérateurs de contrôle comme le **call/cc** de Scheme et sa variante typée **callcc** (et **throw**) de SML/NJ permettent d'échanger les contextes de coroutines, où le contexte d'une coroutine est exactement sa continuation. Dans ces implantations, et à la différence des mécanismes de coroutines décrits ci-dessus, l'environnement (qui correspond à la pile dans les langages à piles) est partagé : chaque coroutine peut accéder librement aux variables locales des autres coroutines. La notion de $\lambda\mu$-calcul sûr de l'article 1, qui est basé sur cette notion de visibilité des variables locales, modélise exactement cette distinction. Dans un programme sûr, une coroutine n'accède pas à une variable locale d'une autre coroutine. Cette variable peut apparaître dans la portée statique, mais elle ne sera pas disponible dans l'environnement à l'exécution.

Les remarques de la section précédente suggèrent que dans un langage à pile, un contexte de coroutine n'est pas simplement une continuation, mais un couple *environnement + continuation*. Donc le $\lambda\mu$-calcul sûr apparaît comme un moyen statique de garantir la correction de l'exécution, pour une implantation des coroutines où le contexte intègre l'environnement (comme dans les langages à piles). Afin de préciser ces idées, dans le cadre fonctionnel, nous allons adapter la machine abstraite de Krivine au $\lambda\mu$-calcul sûr.

Une machine abstraite pour le $\lambda\mu$-calcul

Nous rappelons tout d'abord la machine de Krivine (qui a été étudiée en détail ces dernières années, cf. le recueil d'articles [Danvy, 2007]) qui implante la réduction faible de tête. Cette machine a été étendue au $\lambda\mu$-calcul dans [Streicher and Reus, 1998] et [de Groote, 1998], puis généralisée à μ-PCF, en évaluation par valeur et par nom, dans [Bierman, 1998a, Bierman, 1998b]. Les états de la machine sont des triplets $[t, \mathcal{E}, \mathcal{S}]$ où t est un terme, $\mathcal{E}$ est un environnement et $\mathcal{S}$ est une pile de clôtures de contextes élémentaires (*evaluation frames*). Un environnement $\mathcal{E}$ est en fait une paire $(\mathcal{E}_\lambda, \mathcal{E}_\mu)$ où $\mathcal{E}_\lambda$ associe des variables à des clôtures et $\mathcal{E}_\mu$ associe des μ-variables à des piles. Formellement, les ensembles **term**, **env**, **frame**, **clos**, **stack** et **state** sont définis inductivement comme suit :

$$
\begin{array}{lrcl}
\textbf{(term)} & t & ::= & x \quad | \quad \lambda x.t \quad | \quad (t_1\ t_2) \quad | \quad \mu\alpha[\beta]t \\
\textbf{(env)} & \mathcal{E} & \in & (\textbf{var} \to \textbf{clos}) \times (\mu\textbf{-var} \to \textbf{stack}) \\
\textbf{(clos)} & c & ::= & \langle t, \mathcal{E} \rangle \\
\textbf{(frame)} & f & ::= & ([\,]\ c) \\
\textbf{(stack)} & \mathcal{S} & ::= & () \quad | \quad f :: \mathcal{S} \\
\textbf{(state)} & \sigma & ::= & [t, \mathcal{E}, \mathcal{S}]
\end{array}
$$

La relation de transition $\to$ entre états est donnée par les règles suivantes :

$$
\begin{array}{lrcll}
\textbf{(var)} & [x, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}', S] & \text{où } \langle t, \mathcal{E}' \rangle = \mathcal{E}(x) \\
\textbf{(fun)} & [\lambda x.t, \mathcal{E}, ([\,]\ c) :: \mathcal{S}] & \to & [\lambda x.t, (\{\mathcal{E}_\lambda, x = c\}, \mathcal{E}_\mu), \mathcal{S}] & \\
\textbf{(app)} & [(t\ u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, ([\,]\ \langle u, \mathcal{E} \rangle) :: \mathcal{S}] & \\
\textbf{(swap)} & [\mu\alpha[\beta]t, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, \mathcal{S}'] & \text{où } \mathcal{S}' = \{\mathcal{E}_\mu, \alpha = \mathcal{S}\}(\beta)
\end{array}
$$

Remarque. La machine ci-dessus peut être prouvée correcte pour l'évaluation des $\lambda\mu$-termes [de Groote, 1998]. Le cœur de la preuve consiste, comme toujours, à associer un $\lambda\mu$-terme s^* à chaque état de la machine et à montrer que la machine implémente la réduction du $\lambda\mu$-calcul, c'est-à-dire que si $s \to s'$ alors $s^* \to^\star s^{*\prime}$. Le terme s^* est obtenu en appliquant récursivement les environnements (vus comme des substitutions) et en reconstruisant les contextes. La preuve de correction peut être simplifiée en utilisant un calcul intermédiaire avec substitutions explicites. Par ailleurs, cette machine peut être systématiquement dérivée (et elle est dans ce cas correcte par construction) à partir de la sémantique contextuelle en appliquant les techniques développées dans [Biernacka and Danvy, 2007].

Une machine abstraite pour les coroutines

Nous adaptons maintenant la machine précédente pour permettre le changement de contexte. Un contexte de coroutine consiste en une paire composée de l'environnement de la coroutine et d'une continuation. La seconde composante $\mathcal{E}_\mu$ d'un environnement $\mathcal{E}$ associe maintenant un contexte de coroutine à chaque μ-variable. Pour refléter ce changement, il suffit de modifier la définition de la catégorie **env** :

$$\textbf{(env)} \quad \mathcal{E} \in (\textbf{var} \to \textbf{clos}) \times (\mu\textbf{-var} \to ((\textbf{var} \to \textbf{clos}) \times \textbf{stack}))$$

La nouvelle machine est censée implanter le changement de contexte, et c'est le rôle de la règle **swap**. Nous modifions donc cette règle en conséquence :

$$\textbf{(swap)} \quad [\mu\alpha[\beta]t, (\mathcal{E}_\lambda, \mathcal{E}_\mu), \mathcal{S}] \;\to\; [t, (\mathcal{E}'_\lambda, \mathcal{E}_\mu), \mathcal{S}'] \quad \text{où } (\mathcal{E}'_\lambda, \mathcal{S}') = \{\mathcal{E}_\mu, \alpha = (\mathcal{E}_\lambda, \mathcal{S})\}(\beta)$$

Comme cas particuliers, nous obtenons les règles pour les abréviations **get-context** et **set-context** à partir de la règle **swap** :

$$\begin{array}{lcll} [\textbf{get-context}\ \alpha\ t, (\mathcal{E}_\lambda, \mathcal{E}_\mu), \mathcal{S}] & \to & [t, (\mathcal{E}_\lambda, \mathcal{E}'_\mu), \mathcal{S}] & \text{où } \mathcal{E}'_\mu = \{\mathcal{E}_\mu, \alpha = (\mathcal{E}_\lambda, \mathcal{S})\} \\ [\textbf{set-context}\ \beta\ t, (\mathcal{E}_\lambda, \mathcal{E}_\mu), \mathcal{S}] & \to & [t, (\mathcal{E}'_\lambda, \mathcal{E}_\mu), \mathcal{S}'] & \text{où } (\mathcal{E}'_\lambda, \mathcal{S}') = \mathcal{E}_\mu(\beta) \end{array}$$

Remarquez que l'environnement courant est sauvegardé (avec la continuation courante) au moment où **get-context** est exécuté. Inversement, quand **set-context** est exécuté, la continuation donnée est re-installée et l'environnement correspondant est restauré (alors que l'environnement courant et la continuation courante sont oubliés).

Remarque. La machine ci-dessus est correcte pour les $\lambda\mu$-termes sûrs. L'idée de la preuve consiste, à nouveau, à associer un $\lambda\mu$-terme s^* à chaque état de la machine et à montrer que la machine implémente la réduction du $\lambda\mu$-calcul, c'est-à-dire que si $s \to s'$ alors $s^* \to^\star s^{*\prime}$. La seule chose qui change par rapport à la machine précédente est la forme des environnements, et l'application récursive des substitutions permettant de construire s^* n'est donc plus triviale. On peut toutefois montrer que les états de la nouvelle machine représentent les mêmes $\lambda\mu$-termes, à condition qu'ils soient sûrs.

Coroutines de première classe

Pour étendre la machine aux coroutines de première classe (typées par la soustraction) nous complétons la syntaxe de termes et des contextes d'évaluation ainsi :

$$\begin{array}{lrcl} \textbf{(term)} & t, u & ::= & \ldots \mid \textbf{make-coroutine}\ \alpha\ t \mid \textbf{resume}\ t\ \textbf{with}\ x \mapsto u \\ \textbf{(frame)} & f & ::= & \ldots \mid \textbf{resume}\ [\,]\ \textbf{with}\ x \mapsto c \end{array}$$

Les nouvelles règles de réduction sont alors les suivantes :

$$\begin{array}{rcl} [(\textbf{resume}\ t\ \textbf{with}\ x \mapsto u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, (\textbf{resume}\ [\,]\ \textbf{with}\ x \mapsto \langle u, \mathcal{E}\rangle) :: \mathcal{S}] \\ [(\textbf{make-coroutine}\ \alpha\ t), \mathcal{E}, (\textbf{resume}\ [\,]\ \textbf{with}\ x \mapsto \langle u, \mathcal{E}'\rangle) :: \mathcal{S}] & \to & [u, (\{x = \langle t, \mathcal{E}\rangle\}, \mathcal{E}'_\mu), \mathcal{E}_\mu(\alpha)] \end{array}$$

Remarque. À nouveau, la correction de cette machine découle du fait que les règles sont dérivées de l'encodage des primitives à partir des paires et des continuations de première classe :

- **make-coroutine** $t\ \alpha \equiv \langle t, \lambda x.\textbf{set-context}\ \alpha\ x\rangle$
- **resume** c **with** $x \mapsto u \equiv$ **match** c **with** $\langle x, k\rangle \mapsto (k\ u)$

La différence principale par rapport l'exécution directe de ces macro-définitions provient de la l'environnement $\{x = \langle t, \mathcal{E}\rangle\}$ au lieu de $\{\mathcal{E}_\lambda, x = \langle t, \mathcal{E}\rangle\}$ dans la seconde règle, et cette optimisation est justifiée par le fait que le $\lambda\mu$-terme exécuté est sûr et que ces deux environnements produisent donc le même terme lorsqu'ils sont appliqués à u.

4.2 Coroutines impératives

En appliquant les techniques développées dans les articles 2 et 3, il est possible de faire basculer les opérateurs de manipulation de coroutines dans le monde impératif. Les opérateurs deviennent des instructions primitives **get-context** $\alpha\ \{s\}$ et **set-context** $\alpha\ \{s\}$ où α est un label et s est une séquence et la traduction vers le fonctionnel est celle attendue :

$$\begin{aligned}(\textbf{get-context}\ \alpha\ \{s\}_{\vec{z}})^\star &= \textbf{get-context}\ \alpha\ (s)^\star_{\vec{z}}\\ (\textbf{set-context}\ \alpha\ \{s\}_{\vec{z}})^\star &= \textbf{set-context}\ \alpha\ (s)^\star_{\vec{z}}\end{aligned}$$

De même, **make-coroutine** devient une procédure primitive qui prend en entrée un label et une expression et produit une coroutine de première classe. Sa traduction est la même que pour un appel de procédure non-primitive :

$$(\textbf{make-coroutine}(t, \alpha; c);\ s)^\star_{\vec{z}} = \textbf{let}\ c = \textbf{make-coroutine}\ t^\star\ \alpha\ \textbf{in}\ s^\star_{\vec{z}}$$

Finalement, l'opérateur **resume** devient une instruction primitive définie par la traduction suivante vers le fonctionnel (où x est une variable immutable) :

$$(\textbf{resume}\ c\ \textbf{with}\ x \mapsto \{s\}_{\vec{z}})^\star = \textbf{resume}\ c^\star\ \textbf{with}\ x \mapsto (s)^\star_{\vec{z}}$$

Remarque. La machine abstraite pour les coroutines fonctionnelles décrite précédemment implante la stratégie d'appel par nom décrite dans l'article 1. Il faut adapter cette machine au Système T avec opérateurs de contrôle, en appel par valeur, avant de pouvoir exploiter les résultats des articles 2 et 3. On pourra pour cela s'appuyer sur la machine de Krivine développée pour μ-PCF dans [Bierman, 1998a, Bierman, 1998b].

Système de types dépendants et programmes impératifs « sûr »

Il est bien entendu possible de définir des notions de typage et de sûreté pour les programme impératif indirectement en s'appuyant sur la traduction : un programme est bien typé (resp. sûr) si et seulement si son image fonctionnelle est bien typée (resp. sûre). Il serait toutefois intéressant d'expliciter ce système de types dépendants pour les coroutines impératives en particulier pour permettre une comparaison avec les sémantiques axiomatiques décrites dans [Clint, 1973] et [Dahl, 1975].

De façon similaire, même s'il est possible de définir la sémantique opérationnelle des coroutines impératives par traduction dans le fonctionnel et l'exécution par la machine abstraite décrite ci-dessus, l'objectif principal est de se rapprocher d'une machine à pile usuelle pour un langage impératif, pour laquelle la notion de sûreté est naturelle. En effet, dans un langage fonctionnel, le programmeur est habitué à ce que toutes les variables soient visibles et qu'une clôture soit créée implicitement lorsque cela est nécessaire (ce qui est le cas par exemple pour les coroutines de Lua [Ierusalimschy et al., 2005]). La notion de sûreté peut donc sembler artificielle. À l'inverse, dans un langage à pile, pour une instruction comme celle-ci :

$$\textbf{resume}\ c\ \textbf{with}\ (\vec{x}) \to \{s\}$$

le programmeur s'attend à ce que l'instruction **resume** restaure la pile et que par conséquent, il soit limité dans s à l'utilisation de certaines variables (celles de $\vec{x}$). Les contraintes de ce type dans les langages impératifs sont généralement empiriques, et il serait clairement préférable de les spécifier formellement en les intégrant au système de types.

5 Conclusion et perspectives

Pour conclure, nous énumérons quelques tâches accomplies et celles qui restent à accomplir pour compléter le cadre formel décrit dans l'introduction, dans le contexte de l'étude formelle des coroutines de première classe :

- Nous avons défini un système de types fonctionnel pour un $\lambda\mu$-calcul avec coroutines qui correspond à l'arithmétique soustractive. Il reste à définir un système de types dépendants pour le langage impératif qui lui correspond (par traduction dans le fonctionnel).
- Nous avons défini un système de types dépendants impératifs pour les sauts non-locaux. Il reste à étudier plus précisément comment plonger une logique de Floyd-Hoare, et en particulier la règle de la conséquence, dans ce système de types dépendants impératif.
- Nous avons prouvé un théorème de simulation des programmes Loop$^\omega$ par des termes du système T, et nous avons étendu ce travail aux opérateurs de contrôle (par traduction CPS). Il reste à définir une machine abstraite impérative pour les sauts non-locaux et montrer un théorème de simulation similaire à celui de l'article 2, mais cette fois-ci directement au niveau des machines abstraites.
- Nous pourrons alors adapter cette machine impérative aux coroutines de première classe, en se basant sur les contraintes naturellement imposées par une discipline de pile. Cette machine devra être prouvée correcte pour les programmes impératifs typables dans l'arithmétique soustractive.
- Nous avons étudié l'encodage des opérateurs de continuations délimitées **shift** et **reset** dans le cadre impératif. Ces opérateurs sont souvent ceux utilisés en pratique pour simuler des coroutines fonctionnelles. Le lien entre ces opérateurs et les coroutines (et la logique soustractive) mériterait d'être approfondi d'un point de vue formel.
- Nous avons spécifié formellement le système de types et la sémantique transitionnelle de Loop$^\omega$ avec Ott et Isabelle/HOL. Nous avons l'intention de développer prochainement la méta-théorie de Loop$^\omega$ dans Coq en nous basant sur l'encodage *locally nameless* récemment ajouté à Ott.
- Finalement, il faudrait aussi étendre le cadre formel à d'autres structures de données comme les arbres binaires et prouver la correction d'exemples non triviaux, comme par exemple la fameuse fonction *same-fringe*.

Bibliographie

[Apt, 1981] Apt, K. R. (1981). Ten years of hoare's logic: A survey – part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483.

[Ariola et al., 2007] Ariola, Z. M., Herbelin, H., and Sabry, A. (2007). A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*.

[Asperti and Longo, 1991] Asperti, A. and Longo, G. (1991). *Categories, Types and Structures*. MIT Press.

[Atkey, 2006] Atkey, R. (2006). Parameterised notions of computation. In McBride, C. and Uustalu, T., editors, *MSFP 2006: Workshop on mathematically structured functional programming*. Electronic Workshops in Computing, British Computer Society.

[Atkey, 2008] Atkey, R. (2008). Parameterised notions of computation. *Journal of Functional Programming*.

[Barbanera and Berardi, 1994a] Barbanera, F. and Berardi, S. (1994a). Extracting constructive content from classical logic via control-like reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag.

[Barbanera and Berardi, 1994b] Barbanera, F. and Berardi, S. (1994b). A symmetric lambda calculus for "classical" program extraction. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 495–515. Springer-Verlag.

[Barnes, 2003] Barnes, J. (2003). *High integrity software: the SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Beauquier et al., 2001] Beauquier, D., Crolard, T., Durand, A., and Slissenko, A. (2001). Impossibility of essential real-time garbage collection in the general case. In *CSIT'01*, pages 17–20, Yerevan (Armenia).

[Beckmann and Weiermann, 2000] Beckmann, A. and Weiermann, A. (2000). Characterizing the elementary recursive functions by a fragment of Gödel's T. *Archive for Mathematical Logic*, V39:475–491.

[Bellin, 2005] Bellin, G. (2005). A term assignment for dual intuitionistic logic. In *Proceedings of the LICS'05-IMLA'05 Workshop*.

[Bellin and Biasi, 2004] Bellin, G. and Biasi, C. (2004). Towards a logic for pragmatics. assertions and conjectures. *Journal of Logic and Computation*, 14(4):473.

[Bellin and Ranalter, 2003] Bellin, G. and Ranalter, K. (2003). A kripke-style semantics for the intuitionistic logic of pragmatics ilp. *Journal of Logic and Computation*, 13(5):755.

[Berghofer and Nipkow, 2002] Berghofer, S. and Nipkow, T. (2002). Executing higher order logic. In *In Proc. TYPES Working Group Annual Meeting 2000, LNCS*, pages 24–40. Springer-Verlag.

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag.

[Biasi and Aschieri, 2008] Biasi, C. and Aschieri, F. (2008). A term assignment for polarized bi-intuitionistic logic and its strong normalization. *Fundam. Inf.*, 84(2):185–205.

[Bierman, 1998a] Bierman, G. M. (1998a). A computational interpretation of the lambda-μ-calculus. In Brim, L., Gruska, J., and Zlatuska, J., editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345. Springer.

[Bierman, 1998b] Bierman, G. M. (1998b). The $\lambda\mu$-calculus: Function and control. Technical Report 448, University of Cambridge Computer Laboratory.

[Biernacka and Danvy, 2007] Biernacka, M. and Danvy, O. (2007). A syntactic correspondence between context-sensitive calculi and abstract machines. *TCS: Theoretical Computer Science*, 375.

[Buisman and Goré, 2007] Buisman, L. and Goré, R. (2007). A cut-free sequent calculus for bi-intuitionistic logic: Extended version.

[Cardelli and Leroy, 1990] Cardelli, L. and Leroy, X. (1990). Abstract types and the dot notation. In Broy, M. and Jones, C. B., editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland.

[Clint, 1973] Clint, M. (1973). Program proving: Coroutines. *Acta Informatica*, 2(1):50–63.

[Colson and Fredholm, 1998] Colson, L. and Fredholm, D. (1998). System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315.

[Conway, 1963] Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408.

[Cooper and Morrisett, 1990] Cooper, E. C. and Morrisett, J. G. (1990). Adding threads to standard ML. Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Cousot, 1990] Cousot, P. (1990). Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–994. Elsevier Science Publishers B.V. (North Holland).

[Crolard, 1996] Crolard, T. (1996). Extension de l'isomorphisme de Curry-Howard au traitement des exceptions (application d'une étude de la dualité en logique intuitionniste). Thèse de Doctorat. Université Paris 7.

[Crolard, 1999a] Crolard, T. (1999a). A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming*, 9(6):625–647.

[Crolard, 1999b] Crolard, T. (1999b). Typage des coroutines en logique soustractive. In *Journées Francophones des Langages Applicatifs*, Collection Didactique de l'INRIA, pages 73–92.

[Crolard, 2001] Crolard, T. (2001). Subtractive Logic. *Theoretical Computer Science*, 254(1–2):151–185.

[Crolard, 2002] Crolard, T. (2002). A Constructive Restriction of the $\lambda\mu$-calculus. Technical Report 02, LACL - Université Paris Est.

[Crolard, 2004] Crolard, T. (2004). A Formulæ-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14(4):529–570.

[Crolard, 2009] Crolard, T. (2009). A formally specified program logic for higher-order procedural variables and non-local jumps. Technical Report 10, LACL - Université Paris-Est.

[Crolard and Durand, 2000] Crolard, T. and Durand, A. (2000). Real-time garbage collection and complexity issues. In *Workshop on Specification and Verification of Real-time Systems.* LACL - Université Paris 12.

[Crolard et al., 2006] Crolard, T., Lacas, S., and Valarcher, P. (2006). On the Expressive Power of the Loop Language. *Nordic Journal of Computing*, 13(1-2):46–57.

[Crolard and Polonowski, 2009] Crolard, T. and Polonowski, E. (2009). A formally specified type system and operational semantics for higher-order procedural variables. Technical Report 03, LACL - Université Paris Est.

[Crolard and Polonowski, 2010] Crolard, T. and Polonowski, E. (2010). A program logic for higher-order procedural variables and non-local jumps. Submitted.

[Crolard et al., 2009] Crolard, T., Polonowski, E., and Valarcher, P. (2009). Extending the loop language with higher-order procedural variables. *Special issue of ACM TOCL on Implicit Computational Complexity*, 10(4):1–37.

[Curien and Herbelin, 2000] Curien, P.-L. and Herbelin, H. (2000). The duality of computation. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243, N.Y. ACM Press.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic*. North-Holland.

[Dahl, 1975] Dahl, O.-J. (1975). An approach to correctness proofs of semicoroutines. *Mathematical Foundations of Computer Science*, pages 157–174.

[Dahl et al., 1972] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured programming*. Academic Press.

[Dahl and Nygaard, 1966] Dahl, O.-J. and Nygaard, K. (1966). Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678.

[Dalla Pozza and Garola, 1995] Dalla Pozza, C. and Garola, C. (1995). A pragmatic interpretation of intuitionistic propositional logic. *Erkenntnis*, 43:81–109.

[Danvy, 1989] Danvy, O. (1989). On listing list prefixes. *SIGPLAN Lisp Pointers*, 2(3-4):42–47.

[Danvy, 2007] Danvy, O., editor (2007). *Special Issue on the Krivine Machine*, volume 20.

[Danvy and Filinski, 1989] Danvy, O. and Filinski, A. (1989). A functional abstraction of typed contexts. Technical report, Copenhagen University.

[de Groote, 1995] de Groote, P. (1995). A simple calculus of exception handling. In *Second International Conference on Typed Lambda Calculi and Applications*, LNCS, pages 201–215, Edinburgh, United Kingdom.

[de Groote, 1998] de Groote, P. (1998). An environment machine for the lambda-mu-calculus. *Mathematical Structure in Computer Science*, 8:637–669.

[de Groote, 2001] de Groote, P. (2001). Strong normalization of classical natural deduction with disjunction. In Calculi, T. L. and Applications, editors, *LNCS*, pages LNCS 2044, p. 182 ff.1–13. Springer-Verlag.

[de Moura and Ierusalimschy, 2004] de Moura, A. L. and Ierusalimschy, R. (2004). Revisiting coroutines. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ.

[de Moura et al., 2004] de Moura, A. L., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925.

[de Paiva and Pereira, 2005] de Paiva, V. and Pereira, L. C. (2005). A short note on intuitionistic propositional logic with multiple conclusions. In Campinas, R. I. F., editor, *Manuscrito*, volume 28, pages 317–329.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A discipline of programming*. Prentice Hall.

[Donahue, 1977] Donahue, J. E. (1977). Locations considered unnecessary. *Acta Inf.*, 8:221–242.

[Enderton, 1972] Enderton, H. B. (1972). *A mathematical introduction to logic*. Academic press New York.

[Engelschall, 2000] Engelschall, R. S. (2000). Portable multithreading: the signal stack trick for user-space thread creation. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 239–250, Berkeley, CA, USA. USENIX Association.

[Erné et al., 2007] Erné, M., Gehrke, M., and Pultr, A. (2007). Complete congruences on topologies and down-set lattices. *Applied Categorical Structures*, 15(1):163–184.

[Filinski, 1989] Filinski, A. (1989). Declarative Continuations: An Investigation of Duality in Programming Language Semantics. In *Category Theory and Comp. Sci.*, volume 389 of *LNCS*, pages 224–249. Springer-Verlag.

[Filinski, 1994] Filinski, A. (1994). Representing monads. In *Conference Record of the Twenty-First Annual Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon.

[Fitting, 2004] Fitting, M. (2004). First-order intensional logic. *Annals of Pure and Applied Logic*, 127(1-3):171–193.

[Flanagan et al., 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1.

[Friedman et al., 1984] Friedman, D. P., Haynes, C. T., and Wand, M. (1984). Continuations and coroutines. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298.

[Friedman et al., 1986] Friedman, D. P., Haynes, C. T., and Wand, M. (1986). Obtaining coroutines with continuations. *Journal of Computer Languages*, 11(3/4):143–153.

[Gallier, 2003] Gallier, J. (2003). *Logic for computer science: foundations of automatic theorem proving*. Wiley, New York.

[Gellerich and Plödereder, 2001] Gellerich, W. and Plödereder, E. (2001). Parameter-induced aliasing in ada. In Craeynest, D. and Strohmeier, A., editors, *Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference Leuven, Belgium, May 14-18, 2001, Proceedings*, volume 2043 of *Lecture Notes in Computer Science*, pages 88–99. Springer.

[Girard et al., 1989] Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*, volume 7. Cambridge Tracts in Theorical Comp. Sci.

[Gödel, 1958] Gödel, K. (1958). Über eine bisher noch nicht benütz teerweiterung des finiten standpunktes. *Dialectica*, 12:280–287.

[Gordon, 1988] Gordon, M. J. C. (1988). Specification and verification I. Lecture notes, University of Cambridge, Computer Laboratory.

[Görnemann, 1971] Görnemann, S. (1971). A logic stronger than intuitionism. *The Journal of Symbolic Logic*, 36:249–261.

[Griffin, 1990] Griffin, T. G. (1990). A formulæ-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Langages*, pages 47–58.

[Griswold and Griswold, 1983] Griswold, R. E. and Griswold, M. T. (1983). *The Icon programming language*. Prentice-Hall Englewood Cliffs, New Jersey.

[Grzegorczyk, 1964] Grzegorczyk, A. (1964). A philosophically plausible formal interpretation of intuitionistic logic. *Nederl. Akad. Wet., Proc., Ser. A*, 67:596–601.

[Harper et al., 1993] Harper, R., Duba, B. F., and MacQueen, D. (1993). Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484.

[Hatcliff and Danvy, 1994] Hatcliff, J. and Danvy, O. (1994). A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA. ACM.

[Herbelin, 1995] Herbelin, H. (1995). A λ-calculus structure isomorphic to gentzen-style sequent calculus structure. In Pacholski, L. and Tiuryn, J., editors, *Selected papers 8th Workshop Computer Science Logic, CSL'94, Kazimierz, Poland. 25–30 Sept 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, Berlin.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Howard, 1969] Howard, W. A. (1969). The formulæ-as-types notion of constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculs and Formalism*, pages 479–490. Academic Press.

[Hyland and de Paiva, 1993] Hyland, M. and de Paiva, V. (1993). Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64(3):273–291.

[Ierusalimschy et al., 2005] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2005). The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176.

[ISO, 2003] ISO (2003). C♯ language specification ISO/IEC 23270.

[Kahn, 1987] Kahn, G. (1987). Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag.

[Kashima, 1991] Kashima, R. (1991). Cut-elimination for the intermediate logic cd. Research Report on Information Sciences C100, Institute of Technology, Tokyo.

[Kelsey et al., 1998] Kelsey, R., Clinger, W., and Rees, J. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 2nd edition edition.

[Krivine, 1993] Krivine, J.-L. (1993). *Lambda-calculus, types and models*. Ellis Horwood.

[Krivine, 1994] Krivine, J.-L. (1994). Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78.

[Krivine and Parigot, 1990] Krivine, J.-L. and Parigot, M. (1990). Programming with proofs. *J. Inf. Process. Cybern. EIK*, 26(3):149–167.

[Kuroda, 1951] Kuroda, S. (1951). Intuitionistische untersuchungen der formalistischen logik. *Nagoya Math. J*, 2:35–47.

[Landin, 1964] Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

[Landin, 1965a] Landin, P. J. (1965a). A correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101.

[Landin, 1965b] Landin, P. J. (1965b). A correspondence between algol 60 and church's lambda-notations: Part ii. *Commun. ACM*, 8(3):158–167.

[Landin, 1965c] Landin, P. J. (1965c). A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research.

[Leivant, 1990] Leivant, D. (1990). Contracting proofs to programs. In *Logic and Computer Science*, pages 279–327. Academic Press.

[Leivant, 2002] Leivant, D. (2002). Intrinsic reasoning about functional programs i: first order theories. *Annals of Pure and Applied Logic*, 114(1-3):117–153.

[Liang et al., 1995] Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343.

[Makarov, 2006] Makarov, Y. (2006). Practical program extraction from classical proofs. *Electronic Notes in Theoretical Computer Science*, 155:521 – 542. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[Marlin, 1980] Marlin, C. D. (1980). *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Martin-Löf, 1975] Martin-Löf, P. (1975). An intuitionistic theory of types: predicative part. In Rose, H. E. and Stepherdson, J. C., editors, *Logic colloquium '73*. North-Holland.

[Meyer and Ritchie, 1976] Meyer, A. R. and Ritchie, D. M. (1976). The complexity of loop programs. In *Proc. ACM Nat. Meeting*.

[Mitchell and Plotkin, 1985] Mitchell, J. C. and Plotkin, G. D. (1985). Abstract types have existential type. In *12th Annual ACM symposium on Principles of Programming Languages*.

[Moggi, 1990] Moggi, E. (1990). *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1):55–92.

[Murthy, 1990] Murthy, C. R. (1990). *Extracting Constructive Content from Classical proofs*. PhD thesis, Cornell University, Department of Computer Science.

[Murthy, 1991] Murthy, C. R. (1991). Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

[O'Donnell, 1982] O'Donnell, M. J. (1982). A critique of the foundations of hoare style programming logics. *Commun. ACM*, 25(12):927–935.

[Parigot, 1992] Parigot, M. (1992). $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Logic Prog. and Autom. Reasoning*, volume 624 of *LNCS*, pages 190–201.

[Parigot, 1993a] Parigot, M. (1993a). Classical proofs as programs. In *Computationnal logic and theory*, volume 713 of *LNCS*, pages 263–276. Springer-Verlag.

[Parigot, 1993b] Parigot, M. (1993b). Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*.

[Pfenning and Schürmann, 1999] Pfenning, F. and Schürmann, C. (1999). System description: Twelf - a meta-logical framework for deductive systems. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, London, UK. Springer-Verlag.

[Plotkin, 1981] Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

[Ramsey, 1990] Ramsey, N. (1990). Concurrent programming in ML. Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, Princeton, NJ.

[Ranalter, 2008] Ranalter, K. (2008). A semantic analysis of a logic for pragmatics with assertions, obligations, and causal implication. *Fundam. Inf.*, 84(3,4):443–470.

[Rauszer, 1974a] Rauszer, C. (1974a). A formalization of the propositional calculus of H-B logic. *Studia Logica*, 33(1):23–34.

[Rauszer, 1974b] Rauszer, C. (1974b). Semi-boolean algebras and their applications to intuitionistic logic with dual operations. In *Fundamenta Mathematicae*, volume 83, pages 219–249.

[Rauszer, 1976] Rauszer, C. (1976). On the strong semantical completeness of any extension of the intuitionistic predicate calculus. In *Bulletin de l'Académie Polonaises des Sciences*, volume 24-2 of *Série des sciences math., astr. et phys.*, pages 81–87.

[Rehof and Sørensen, 1994] Rehof, N. J. and Sørensen, M. H. (1994). The λ_δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag.

[Reppy, 1988] Reppy, J. H. (1988). Synchronous operations as first-class values. In Wise, D. S., editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 250–259, Atlanta, GE, USA. ACM Press.

[Reppy, 1989] Reppy, J. H. (1989). First-class synchronous operations in standard ML. Technical Report TR89-1068, Cornell University, Computer Science Department.

[Reppy, 1990] Reppy, J. H. (1990). Asynchronous signals is standard ML. Technical Report TR90-1144, Cornell University, Computer Science Department.

[Reppy, 1995] Reppy, J. H. (1995). First-class synchronous operations. *Lecture Notes in Computer Science*, 907:235–252.

[Selinger, 2001] Selinger, P. (2001). Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260.

[Sewell et al., 2007] Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strniša, R. (2007). Ott: effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12.

[Shimura and Kashima, 1994] Shimura, T. and Kashima, R. (1994). Cut-elimination theorem for the logic of constant domains. *Math. Log. Q*, 40:153–172.

[Streicher and Reus, 1998] Streicher, T. and Reus, B. (1998). Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572.

[Talpin and Jouvelot, 1994] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

[The Open Group, 1997] The Open Group (1997). The Single UNIX Specification, Version 2. (http://www.UNIX-systems.org/online.html).

[Thielecke, 1998] Thielecke, H. (1998). An introduction to landin's "a generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–123.

[Thielecke, 2008] Thielecke, H. (2008). Control effects as a modality. *Journal of Functional Programming*, 19:17–26.

[Troelstra, 1973] Troelstra, A. S. (1973). *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin.

[Uustalu and Pinto, 2006] Uustalu, T. and Pinto, L. (2006). Proof search and countermodel construction in bi-intuitionistic propositional logic. In *Days in logic '06*.

[Wadler, 1994] Wadler, P. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55.

[Wadler, 2003] Wadler, P. (2003). Call-by-value is dual to call-by-name. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden. ACM Press.

[Wadler, 2005] Wadler, P. (2005). Call-by-value is dual to call-by-name - reloaded. In Giesl, J., editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer.

[Wehmeier, 1996] Wehmeier, K. F. (1996). Classical and intuitionistic models of arithmetic. *Notre Dame Journal of Formal Logic*, 37(3):452–461.

[Wirth and Mincer-Daszkiewicz, 1980] Wirth, N. and Mincer-Daszkiewicz, J. (1980). *Modula-2*. ETH Zurich, Schweiz.

Deuxième partie

Articles sélectionnés

Chapitre 1

Interprétation calculatoire de la logique soustractive

∗. Les résultats présentés dans ce chapitre ont été publiés dans le *Journal of Logic and Computation. Special issue on Intuitionistic Modal Logic and Application.* Volume 14, Issue 4, August 2004. pp 529-570.

A Formulae-as-Types Interpretation of Subtractive Logic

Abstract

We present a formulae-as-types interpretation of Subtractive Logic (*i.e.* bi-intuitionistic logic). This presentation is two-fold: we first define a very natural restriction of the $\lambda\mu$-calculus which is closed under reduction and whose type system is a constructive restriction of the Classical Natural Deduction. Then we extend this deduction system conservatively to Subtractive Logic. From a computational standpoint, the resulting calculus provides a type system for first-class coroutines (a restricted form of first-class continuations).

1 Introduction

Subtractive logic, also called "bi-intuitionistic logic", is an extension of intuitionistic logic with a new connector, the *subtraction* (called *pseudo-difference* in Rauszer's original work [Rauszer, 1974a, Rauszer, 1974b, Rauszer, 1980]), which is dual to implication. This duality has already been widely investigated from algebraic, relational, axiomatic and sequent perspectives by various authors [Rauszer, 1974a, Rauszer, 1974b, Rauszer, 1980, Restall, 1997, Goré, 2000]. In particular, a unified proof-theoretic approach can be found in Goré's recent paper [Goré, 2000], while a connection with category theory is presented in the author's previous work [Crolard, 1996, Crolard, 2001].

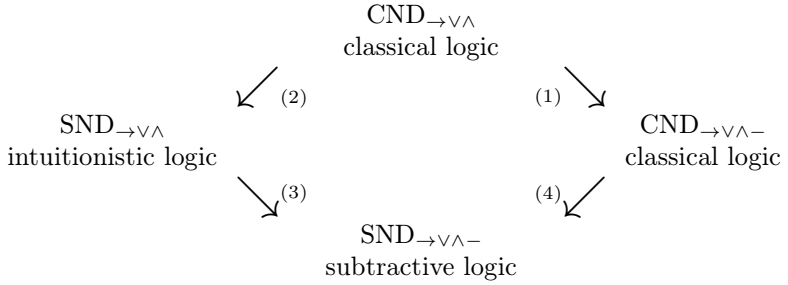
We present in this paper a Curry-Howard correspondence for this logic. In other words, we define an extension of the λ-calculus together with its type system such that the logical interpretation of this type system (as a natural deduction system) is exactly subtractive logic. This work is mainly motivated by the following two facts:

- subtractive logic is not a constructive logic (disjunction and existence properties do not hold in it). In fact, subtractive logic already combines many flavors of classical logic [Crolard, 2001]. However, subtractive logic is conservative over some constructive logic (see below). In other words, disjunction and existence properties hold for formulas which contain no subtraction. Therefore, it is likely that a faithful computational interpretation of subtractive logic would shed new light onto the boundary between constructive and classical logics.
- Curien and Herbelin demonstrated in [Curien and Herbelin, 2000] a striking result: duality in classical logic exchanges call-by-value with call-by-name (see also Wadler's recent paper [Wadler, 2003]). Actually, a first attempt at a categorical continuation semantics was Filinski's pioneering work [Filinski, 1989]. Then Selinger [Selinger, 2001] has investigated thoroughly this duality (but still in a categorical setting). On the other hand, Curien and Herbelin's work is based on Parigot's $\lambda\mu$-calculus and its type system, the Classical Natural Deduction (CND) [Parigot, 1992, Parigot, 1993a]. In order to complete the duality, they are led to extend CND with the subtraction (Walder does not consider the subtraction in [Wadler, 2003]). However, this new connective is presented in a purely formal way (without any computational meaning). We believe that an accurate computational interpretation of subtraction should arise from a formulas-as-types investigation of subtractive logic.

Subtractive Logic

The usual way to define a (sound and faithful) deduction system for subtractive logic is first to add rules for subtraction (which are symmetrical to the rules for implication) and then to restrict sequents to singletons on the left/right sides [Goré, 2000]. Unfortunately, such a calculus is not closed under Parigot's proof normalization process. We shall thus define another restriction which takes into account "dependencies" between hypotheses and conclusions of a derived sequent. Since these dependencies are defined in a symmetrical manner, our restriction can also be dualized to subtraction. We shall call Subtractive Natural Deduction (SND) the resulting system.

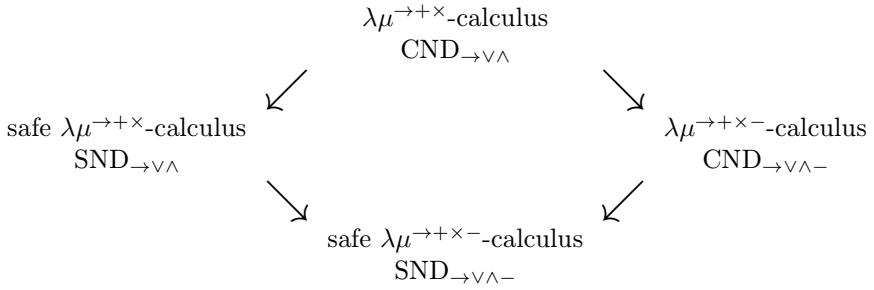
Our method is, on the one hand, to derive introduction and elimination rules for subtraction from its definition in classical logic, and on the other hand, first to restrict the Classical Natural Deduction to intuitionistic logic and then to extend it again to subtractive logic. This process is summarized in the following diagram:



1. Definition of subtraction in classical logic ($A - B \equiv A \wedge \neg B$).
2. Restriction of $\mathrm{CND}_{\to\vee\wedge}$ to intuitionistic logic (resp. CDL).
3. Extension of $\mathrm{SND}_{\to\vee\wedge}$ with restricted rules for subtraction.
4. Dualized restriction of $\mathrm{CND}_{\to\vee\wedge-}$ to subtractive logic.

Formulae-as-types

The restriction above can be rephrased for the pure (*i.e.* untyped) $\lambda\mu$-calculus, in such a way that the former restriction is exactly the latter when we consider typed terms. We shall call "safe $\lambda\mu^{\to+\times-}$-calculus" (resp. "safe $\lambda\mu^{\to+\times}$-calculus") the subset of restricted $\lambda\mu^{\to+\times-}$-terms (resp. $\lambda\mu^{\to+\times}$-terms). This terminology comes from the computational interpretation of this restriction (see below). The main result of this paper is the proof that these subsets are closed under reduction, which means that they actually define a calculus. The diagram above is thus completed accordingly:



Let us now rephrase precisely the relation between safe $\lambda\mu^{\to+\times-}$-terms (resp. safe $\lambda\mu^{\to+\times}$-terms) and proofs of $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$). This relation can be stated as follows: given the derivation of a typing judgement of a $\lambda\mu^{\to+\times-}$-term (resp. $\lambda\mu^{\to+\times}$-term) t by some sequent in $\mathrm{CND}_{\to\vee\wedge-}$ (resp. $\mathrm{CND}_{\to\vee\wedge}$), if t is safe then this derivation belongs to $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$). As a consequence, the subject reduction for $\mathrm{CND}_{\to\vee\wedge-}$ (resp. $\mathrm{CND}_{\to\vee\wedge}$) together with the closure under reduction of the safe $\lambda\mu^{\to+\times-}$-calculus (resp. $\lambda\mu^{\to+\times}$-calculus) provides the subject reduction for $\mathrm{SND}_{\to\vee\wedge-}$ (resp. $\mathrm{SND}_{\to\vee\wedge}$).

Strong normalization and the subformula property

Since our calculi are derived from de Groote's $\lambda\mu^{\to\wedge\vee\bot}$, they enjoy the properties listed in [de Groote, 2001], namely:

a) all connectives are taken as primitive;

b) normal deductions satisfy the subformula property;

c) the reduction relation is defined by means of local reduction steps;

d) the reduction relation is strongly normalizing ;

e) the reduction relation is Church-Rosser;

f) the reduction relation is defined at the untyped level;

g) the reduction relation satisfies the subject reduction property.

In [Rauszer, 1974a], C. Rauszer defined a sequent calculus for subtractive logic which enjoys cut-elimination (and the subformula property which follows). As a by-product of the previous properties, we obtain a new proof of this result (actually, we prove a stronger result, namely the *strong* normalization). As a corollary of the subformula property, we obtain that the normal form of the deduction of a sequent in $\mathrm{CND}_{\rightarrow\vee\wedge-}$ (resp. $\mathrm{SND}_{\rightarrow\vee\wedge-}$) which does not contain any occurrence of subtraction belongs to $\mathrm{CND}_{\rightarrow\vee\wedge}$ (resp. $\mathrm{SND}_{\rightarrow\vee\wedge}$).

Computational interpretation

Since Griffin's pioneering work [Griffin, 1990], the extension of the well-known formulas-as-types paradigm to classical logic has been widely investigated for instance by Murthy [Murthy, 1991], Barbanera and Berardi [Barbanera and Berardi, 1994], Rehof and Sørensen [Rehof and Sørensen, 1994], de Groote [de Groote, 1995, de Groote, 2001], Krivine [Krivine, 1994]. We shall consider here Parigot's $\lambda\mu$-calculus mainly because it is confluent and strongly normalizing in the second order framework [Parigot, 1993b]. However, Parigot's original CND is a second-order logic, in which $\vee, \wedge, \exists, \exists^2$ are definable from $\rightarrow, \forall, \forall^2$. Since we are also interested in the subformula property, we shall restrict ourselves to the propositional framework where any connective is taken as primitive and where the proof normalization process includes permutative conversions [de Groote, 2001]. Such an extension of CND with primitive conjunction and disjunction has already been investigated by Pym, Ritter and Wallen [Pym et al., 1996, Pym et al., 2000, Pym and Ritter, 2001] and de Groote [de Groote, 2001].

The computational interpretation of classical logic is usually given by a λ-calculus extended with some form of control (such as the famous **call/cc** of Scheme or the catch/throw mechanism of Lisp) or similar formulations of first-class continuation constructs. Continuations are used in denotational semantics to describe control commands such as jumps. They can also be used as a programming technique to simulate backtracking and coroutines. For instance, first-class continuations have been successfully used to implement Simula-like cooperative coroutines in Scheme [Wand, 1980, Friedman et al., 1984, Friedman et al., 1986]. This approach has been extended in the Standard ML of New Jersey (with the typed counterpart of Scheme's **call/cc** [Duba et al., 1991]) to provide simple and elegant implementations of light-weight processes (or threads), where concurrency is obtained by having individual threads voluntarily suspend themselves [Reppy, 1995, Ramsey, 1990] (providing time-sliced processes using pre-emptive scheduling requires additional run-time system support [Cooper and Morrisett, 1990, Reppy, 1990]). The key point in these implementations is that control operators make it possible to switch between coroutine contexts, where the context of a coroutine is encoded as its continuation.

The definition of a catch/throw mechanism is straightforward in the $\lambda\mu$-calculus: just set $\mathbf{catch}\,\alpha\,t \equiv \mu\alpha[\alpha]t$ and $\mathbf{throw}\,\alpha\,t \equiv \mu\delta[\alpha]t$ where δ is a name which does not occur in t (see [Crolard, 1999] for a study of the sublanguage obtained when we restrict the $\lambda\mu$-terms to these operators). Then a name α may be reified as the first-class continuation $\lambda x.\mathbf{throw}\,\alpha\,x$. However, the type of such a $\lambda\mu$-term is the excluded-middle $\vdash A^{\alpha}; \neg A$. Thus, a continuation cannot be a first-class citizen in a constructive logic. To figure out what kind of restricted use of continuations is allowed in $\mathrm{SND}_{\rightarrow\vee\wedge}$, we observe that in the restricted $\lambda\mu^{\rightarrow+\times}$-calculus, even if continuations are no longer first-class objects, the ability of context-switching remains (in fact, this observation is easier to make in the framework of abstract state machines). However, a context is now a pair $\langle \mathit{environment}, \mathit{continuation} \rangle$. Note that such a pair is exactly what we expect as the context of a coroutine, since a coroutine should not access the local environment (the part of the environment which is not shared) of another coroutine. Consequently, we say that a $\lambda\mu^{\rightarrow+\times}$-term t is "safe with respect to coroutine contexts" (or just "safe" for short) if no coroutines of t access the local environment of another coroutine.

The extension of this interpretation to $\mathrm{SND}_{\to\vee\wedge-}$ is now straightforward. We already know (from its definition in classical logic) that subtraction is a good candidate to type a pair $\langle \mathit{environment}, \mathit{continuation}\rangle$. $\mathrm{SND}_{\to\vee\wedge-}$ is thus interpreted as a type system for first-class coroutines. The dual restriction from the safe $\lambda\mu^{\to+\times-}$-calculus just ensures that one cannot obtain full-fledged first-class continuations back from first-class coroutines.

Related work

Nakano, Kameyama and Sato [Nakano, 1994b, Nakano, 1994a, Nakano, 1995, Kameyama, 1997, Kameyama and Sato, 1998, Kameyama and Sato, 2002] have proposed various logical frameworks that are intended to provide a type system for a lexical variant of the catch/throw mechanism used in functional languages such as Lisp. Moreover, Nakano has shown that it is possible to restrict the catch/throw mechanism in order to stay in an intuitionistic (propositional) framework. However, in their approach, a disjunction is used to type first-class exceptions. In this paper, we generalize these results in several ways:

- We use Parigot's $\lambda\mu$-calculus and its type system, the Classical Natural deduction which is confluent and strongly normalizing in the second order framework [Parigot, 1993b].
- We consider a type system *à la* Curry, which allows us to rephrase the above restriction on pure (*i.e.* untyped) $\lambda\mu$-terms, and not only on typed terms as in [Nakano, 1995].

Brauner and de Paiva [Brauner and de Paiva, 1997] have proposed a restriction of Classical Linear Logic (in order to obtain a sequent-style formulation of Full Intuitionistic Linear Logic defined by Hyland and de Paiva [Hyland and de Paiva, 1993]) very akin to the one presented in this paper. In a recent work [13], de Paiva and Ritter also consider a Parigot-style linear λ-calculus for this logic which is based on Pym and Ritter's $\lambda\mu\nu$-calculus [Pym et al., 2000]. If we forget about linearity (which has of course many consequences on the resulting system), the main advantages of our approach are the following ones:

- We propose a computational interpretation of our restriction (as "coroutines which do not access the local environment on another coroutine").
- Our restriction is symmetrical and thus easily extends to duality. This allows us to propose a computational interpretation of subtraction (as a type constructor for "first-class coroutines").

About first-order subtractive logic

Propositional subtractive logic is conservative over intuitionistic logic. However, in the first-order framework, subtractive logic is no longer conservative over intuitionistic logic but over Constant Domain Logic (CDL). This logic as been studied for instance in [Gabbay, 1981, Görnemann, 1971, Ono, 1983] and [Rauszer, 1980]. Although CDL is not conservative over intuitionistic logic (in the first-order framework), it is important to note that CDL is still a constructive logic (*i.e.* both disjunction and existence properties hold in it [Görnemann, 1971]). Moreover, CDL if fully axiomatized as the first order intuitionistic logic extended with the following axiom schema (called DIS in [Rauszer, 1980]) where x does not occur in B (see [Gabbay, 1981, Ono, 1983] for instance):

$$\forall x(A \vee B) \vdash \forall x\, A \vee B$$

Recall however that this paper is devoted to the computational interpretation of *propositional* subtractive logic.

Plan of the paper

The remainder of the paper is organized as follows. In section 2, we present $\mathrm{CND}_{\to\vee\wedge-}$ which is our extension of $\mathrm{CND}_{\to\vee\wedge}$ with the subtraction. In section 3, we present the $\mathrm{SND}_{\to\vee\wedge-}$ which is our restriction of $\mathrm{CND}_{\to\vee\wedge-}$ to subtractive logic. In section 4, we recall the $\lambda\mu^{\to+\times}$-calculus and we introduce the $\lambda\mu^{\to+\times-}$-calculus. In section 5, we define the safe $\lambda\mu^{\to+\times-}$-calculus and we show how it is related to $\mathrm{CND}_{\to\vee\wedge-}$. Eventually, in section 6, we prove that the safe $\lambda\mu^{\to+\times-}$-calculus is closed under reduction.

$$A \vdash A \quad (axiom)$$

$$\frac{\Gamma, A, A \vdash \Delta; B}{\Gamma, A \vdash \Delta; B}(C_L) \qquad \frac{\Gamma \vdash \Delta; B}{\Gamma, A \vdash \Delta; B}(W_L)$$

$$\frac{\Gamma \vdash \Delta(, A^\alpha); }{\Gamma \vdash \Delta; A}(activate) \qquad \frac{\Gamma \vdash \Delta(, A^\alpha); A}{\Gamma \vdash \Delta, A^\alpha; }(passivate)$$

$$\frac{\Gamma \vdash \Delta; A \qquad \Gamma, A \vdash \Delta; B}{\Gamma \vdash \Delta; B}(cut)$$

$$\frac{\Gamma, A \vdash \Delta; B}{\Gamma \vdash \Delta; A \to B}(I_\to) \qquad \frac{\Gamma \vdash \Delta; A \to B \qquad \Gamma \vdash \Delta; A}{\Gamma \vdash \Delta; B}(E_\to)$$

$$\frac{\Gamma \vdash \Delta; A \qquad \Gamma \vdash \Delta; B}{\Gamma \vdash \Delta; A \wedge B}(I_\wedge)$$

$$\frac{\Gamma \vdash \Delta; A \wedge B}{\Gamma \vdash \Delta; A}(E^1_\wedge) \qquad \frac{\Gamma \vdash \Delta; A \wedge B}{\Gamma \vdash \Delta; B}(E^2_\wedge)$$

$$\frac{\Gamma \vdash \Delta; A}{\Gamma \vdash \Delta; A \vee B}(I^1_\vee) \qquad \frac{\Gamma \vdash \Delta; B}{\Gamma \vdash \Delta; A \vee B}(I^2_\vee)$$

$$\frac{\Gamma \vdash \Delta; A \vee B \qquad \Gamma, A \vdash \Delta; C \qquad \Gamma, B \vdash \Delta; C}{\Gamma \vdash \Delta; C}(E_\vee)$$

$$\frac{\Gamma \vdash \Delta(, C^\gamma); A \qquad \Gamma, B \vdash \Delta(, C^\gamma); C}{\Gamma \vdash \Delta, C^\gamma; A - B}(I_-)$$

$$\frac{\Gamma \vdash \Delta; A - B \qquad \Gamma, A \vdash \Delta; B}{\Gamma \vdash \Delta; C}(E_-)$$

Figure 2.1. System $\mathrm{CND}_{\to\vee\wedge-}$

2 The system $\mathrm{CND}_{\to\vee\wedge-}$

We consider sequents with the form $\Gamma \vdash \Delta; A$ where Γ, Δ are sets of *named* propositional formulas. As usual, the semi-colon serves to single out one distinguished (or "active") conclusion of a sequent, which clearly amounts to naming this occurrence with a special name (when needed, we shall use the name "[]" for this "unnamed" occurrence).

The deductions rules for $\mathrm{CND}_{\to\vee\wedge-}$ (CND with conjunction, disjunction, subtraction and an explicit cut rule) are given in figure 2.1. We write "$\Delta(, A^\alpha)$" in the premise of the *activate/passivate* rules to outline the fact that A^α is allowed but not required in the succedent of the sequent. Note that if A does not occur in Δ in the premise of the *activate* rule the we obtain a weakening rule. Similarly, if A^α already occurs in Δ in the conclusion of the *passivate* rule then we obtain a contraction rule. Moreover, the *activate/passivate* rules allow us to derive the following weakening, contraction and exchange rules (on the right-hand side):

$$\frac{\Gamma \vdash \Delta(, A^\alpha); A}{\Gamma \vdash \Delta; A}(C_R) \qquad \frac{\Gamma \vdash \Delta(, A^\alpha); A}{\Gamma \vdash \Delta, A^\alpha; B}(W_R) \qquad \frac{\Gamma \vdash \Delta(, B^\beta)(, A^\alpha); B}{\Gamma \vdash \Delta, B^\beta; A}(Ex_R)$$

Note that we may replace the semi-colon by a comma in the rules whenever we want to allow implicit exchange (and consider named conclusions up to permutation).

Let us now comment the introduction/elimination rules for subtraction. The introduction rule is dual to the left-hand side introduction rule for implication (in LK), while the elimination rule is reminiscent of the elimination rule for disjunction. A simpler introduction rule for the subtraction would be the following one:

$$\frac{\Gamma \vdash \Delta(, B^\beta); A}{\Gamma \vdash \Delta, B^\beta; A - B}(I'_-)$$

This rule is equivalent to (I_{-}). Indeed, (I'_{-}) is derivable from (I_{-}) as follows (for sake of simplicity, we omit Γ,Δ in axioms):

$$\frac{\Gamma \vdash \Delta(, B^{\beta}); A \qquad B \vdash ; B}{\Gamma \vdash \Delta, B^{\beta}; A - B}\,(I_{-})$$

Conversely, (I_{-}) is derivable from (I'_{-}) using the *cut* rule (we omit the structural rules):

$$\frac{\dfrac{\Gamma \vdash \Delta(, C^{\gamma}), A}{\Gamma \vdash \Delta(, C^{\gamma}), B^{\beta}, A - B}\,(I'_{-}) \qquad \Gamma, B \vdash \Delta(, C^{\gamma}), C}{\Gamma \vdash \Delta, C^{\gamma}, A - B}\,(cut)$$

We shall give more details about the differences between (I_{-}) and (I'_{-}) in section 4.

2.1 Defining $A - B$ as $A \wedge \neg B$

Since $A - B$ is definable in classical logic as $A \wedge \neg B$, let us prove that the introduction/elimination rules for subtraction are derivable. In order to deal with the negation, we add the propositional constant for *falsum* $\bot$. We use the same name ε for occurrences of $\bot$ in all sequents. Moreover, we do not represent $\bot^{\varepsilon}$ in the conclusions of sequents (see [1] for more details about the various treatments of $\bot$ in CND). Now the elimination rule for $\bot$ is just an instance of (W_R) (take $\bot^{\varepsilon}$ as A^{α}):

$$\frac{\Gamma \vdash \Delta; \bot}{\Gamma \vdash \Delta; B}\,(E_{\bot})$$

As usual, we also define $\neg A$ as $(A \to \bot)$.

Derivation of the introduction rule (I'_{-})

$$\frac{\Gamma \vdash \Delta; A \qquad \dfrac{\dfrac{\Gamma, B \vdash \Delta; B}{\Gamma \vdash \Delta, B; \bot}\,(W_R)}{\Gamma \vdash \Delta, B; \neg B}\,(I_{\to})}{\Gamma \vdash \Delta, B; A \wedge \neg B}\,(I_{\wedge})$$

Derivation of the elimination rule (E_{-})

$$\frac{\Gamma \vdash \Delta; A \wedge \neg B \qquad \dfrac{\dfrac{\neg B \vdash \neg B \qquad \Gamma, A \vdash \Delta; B}{\Gamma, A, \neg B \vdash \Delta; \bot}\,(E_{\to})}{\Gamma, A, \neg B \vdash \Delta; C}\,(E_{\bot})}{\Gamma \vdash \Delta; C}\,(E_{\wedge})$$

where the last rule $(E_{\wedge})$ is easily derivable from $(E^1_{\wedge})$, $(E^2_{\wedge})$ and (cut).

2.2 About the duality

The rules chosen for disjunction (resp. subtraction) in $\mathrm{CND}_{\to\vee\wedge-}$ are clearly not dual to the rules for conjunction (resp. implication). As expected, such rules are left-hand side introduction/elimination rules. For instance, the rules for disjunction which are dual to the rules for conjunction are the following ones:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,(LI_{\vee}) \qquad \frac{\Gamma, A \vee B \vdash \Delta}{\Gamma, A \vdash \Delta}\,(LE^1_{\vee}) \qquad \frac{\Gamma, A \vee B \vdash \Delta}{\Gamma, B \vdash \Delta}\,(LE^2_{\vee})$$

It is easy to show that these rules are equivalent to the (right-hand side) introduction/elimination for disjunction. Similarly, we could define by duality left-hand side introduction/elimination rules for subtraction:

$$\frac{\Gamma, A \vdash \Delta, B}{\Gamma, A - B \vdash \Delta}\,(LI_{-}) \qquad \frac{\Gamma, A - B \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vdash \Delta}\,(LE_{-})$$

Let us prove that these rules are indeed equivalent to the (right-hand side) introduction/elimination rules:

Derivation of the left-hand side introduction rule (LI_{-})

$$\dfrac{\dfrac{A-B\vdash A-B \qquad \Gamma, A\vdash \Delta, B}{\Gamma, A-B\vdash \Delta, \bot}(E_{-})}{\Gamma, A-B\vdash \Delta}(C_R)$$

Derivation of the left-hand side elimination rule (LE_{-})

$$\dfrac{\dfrac{A\vdash A \qquad \Gamma, B\vdash \Delta}{\Gamma, A\vdash \Delta, A-B}(I_{-}) \qquad \Gamma, A-B\vdash \Delta}{\Gamma, A\vdash \Delta}(cut)$$

Conversely, the (right-hand side) introduction/elimination rules are derivable from the left-hand side rules. Indeed:

Derivation of the introduction rule (I_{-})

$$\dfrac{\Gamma\vdash \Delta, A \qquad \dfrac{A-B\vdash A-B \qquad \Gamma, B\vdash \Delta, C}{\Gamma, A\vdash \Delta, C, A-B}(LE_{-})}{\Gamma\vdash \Delta, C, A-B}(cut)$$

Derivation of the elimination rule (E_{-})

$$\dfrac{\dfrac{\Gamma\vdash \Delta, A-B \qquad \dfrac{\Gamma, A\vdash \Delta, B}{\Gamma, A-B\vdash \Delta}(LE_{-})}{\Gamma\vdash \Delta}(cut)}{\Gamma\vdash \Delta, C}(W_R)$$

Definition 2.1. *We call Symmetric* $\mathrm{CND}_{\to\vee\wedge-}$ *the system* $\mathrm{CND}_{\to\vee\wedge-}$ *where the rules for disjunction and subtraction are replaced by* $(\mathrm{LI}_{\vee})$, $(\mathrm{LE}^1_{\vee})$, $(\mathrm{LE}^2_{\vee})$, *and* (LI_{-}), (LE_{-}).

3 Constructive restrictions of CND

It is well-known that if we restrict the classical sequent calculus LK [Szabo, 1969] to sequents with at most one conclusion we obtain the intuitionistic sequent calculus LJ [Szabo, 1969]. As for natural deduction, it was originally presented for sequents having one conclusion and formalized intuitionistic logic. Parigot's CND may be seen as an extension of natural deduction to sequents with several conclusions. As expected, this extension leads to classical logic. In order to stay in a constructive framework, several authors (for instance [Kleene, 1952] p. 481 and also [Beth, 1956, Dragalin, 1988, Dyckhoff, 1992]) have suggested to restrict only the introduction rule of implication of LK to sequents with at most one conclusion. The same restriction can be applied to CND and by duality it can be generalized to Symmetric $\mathrm{CND}_{\to\vee\wedge-}$ (definition 2.1) as follows:

Definition 3.1. *We call Symmetric* $\mathrm{SND}^1_{\to\vee\wedge-}$ *the system Symmetric* $\mathrm{CND}_{\to\vee\wedge-}$ *where:*

1. *the rule* $(I_{\to})$ *is restricted to sequents with at most one conclusion,*
2. *the rule* (LI_{-}) *is restricted to sequents with at most one hypothesis.*

Proposition 3.2. *Symmetric* $\mathrm{SND}^1_{\to\vee\wedge-}$ *is sound and complete for Subtractive Logic.*

Proof. By induction on the derivation, we prove that a sequent $\Gamma_1, ..., \Gamma_m\vdash \Delta_1, ..., \Delta_n$ is derivable in Symmetric $\mathrm{SND}^1_{\to\vee\wedge-}$ iff $\Gamma_1\wedge ...\wedge\Gamma_m\vdash \Delta_1\vee ...\vee\Delta_n$ is derivable in the symmetrical propositional categorical calculus defined in [Crolard, 2001]. □

Remark 3.3. The restriction on the rule (LI_{-}) is required, otherwise we are still in classical logic (even if the introduction rule for implication is restricted). See [Crolard, 2001] for more details.

However, this restriction of the introduction rule for implication is not closed under Parigot's proof normalization process. Let us consider for instance the following proof in CND where $(I_\to)$ is applied only to sequents with at most one conclusion:

$$\dfrac{\dfrac{\dfrac{\dfrac{B\vdash;B}{A,B\vdash;B}(W_L)}{B\vdash;A\to B}(I_\to)}{\vdash;B\to(A\to B)}(I_\to)\qquad\dfrac{\vdots}{\Gamma\vdash\Delta;B}}{\Gamma\vdash\Delta;A\to B}(E_\to)$$

Since $(I_\to)$ is immediately followed by the last $(E_\to)$, this proof may be replaced by:

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash\Delta;B}(W_L)}{\Gamma,A\vdash\Delta;B}(I_\to)$$

And in this proof, the rule $(I_\to)$ is applied to a sequent with multiple conclusions. Note that we could try to detect when multiple conclusions are harmless (from the constructive standpoint) in an occurrence of $(I_\to)$. This was also observed by Ritter, Pym and Wallen in [Pym et al., 2000], where a notion of "superfluous subderivations" is defined. Since their definition relies on weakening occurrences in proof terms, we shall come back to this question in section 5 (remark 5.13). We shall here consider another restriction of CND which enjoys the following properties: it is closed under proof normalization, it is expressed independently of proof terms and it is designed to easily extend by symmetry to SND.

Our restriction is based on dependencies between occurrences of hypotheses and occurrences of conclusions in a derived sequent, where dependencies come from axioms and are propagated by inference rules. Then some hypothesis may be discharged onto some conclusion if and only if no other conclusion depends on it.

A very similar notion of dependency, and the same restriction was discovered independently by Brauner and de Paiva [Brauner and de Paiva, 1997, Hyland and de Paiva, 1993], and applied to the definition of Full Intuitionistic Linear Logic. In a recent work [13], de Paiva and Ritter also present a Parigot-style linear λ-calculus for this logic which is based on Pym and Ritter's $\lambda\mu\nu$-calculus.

Note however that linearity has many consequences on the resulting system. For instance, the properties of their multiplicative "disjunction" (the connective $\square$) are very different from the properties of our (usual) intuitionistic disjunction. Moreover we do not need any specific typed λ-calculus in order to keep track how dependencies are propagated (as in [13]). Our annotations are just sets of names.

Eventually, a major difference (if we forget about linearity) with their work in [Brauner and de Paiva, 1997, Hyland and de Paiva, 1993] is that we emphasize here the symmetry of our definition of dependency relations in order to take the subtraction into account.

3.1 A restriction of CND based on dependency relations

We use undirected links to make explicit all *dependencies* between occurrences of hypotheses and occurrences of conclusions in any derived sequent. A link between some occurrence of an hypothesis Γ_i and some occurrence of a conclusion Δ_j may be represented as follows:

$$\Gamma_1\,,\ldots\,\overset{\ulcorner\!\!\!\!\!\!\urcorner}{\Gamma_i}\,,\ldots\,\Gamma_n\ \vdash\ \Delta_1\,,\ldots\,\overset{\ulcorner\!\!\!\!\!\!\urcorner}{\Delta_j}\,,\ldots\,\Delta_m$$

In order to annotate sequents with such links, we name any *occurrence* of hypotheses (x, y, z, ...) and any *occurrence* of conclusions (α, β, γ, ...) in a sequent. We assume that the name of an hypothesis (resp. a conclusion) never occurs twice in a sequent. The links annotating some sequent $\Gamma_1^{x_1},...,\Gamma_n^{x_n}\vdash\Delta_1^{\alpha_1},...,\Delta_m^{\alpha_m}$ provide a subset of $\{x_1,...,x_n\}\times\{\alpha_1,...,\alpha_m\}$ which can be represented by annotating either each conclusion by a set of hypotheses or each hypothesis by a set of conclusions.

$$A^x \vdash \{x\}\colon A \quad (ax)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A^x \vdash \Delta}\,(W_L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \{\}\colon A}\,(W_R)$$

$$\frac{\Gamma, U\colon A, V\colon A \vdash \Delta}{\Gamma, U \cup V\colon A \vdash \Delta}\,(C_L) \qquad \frac{\Gamma \vdash \Delta, U\colon A, V\colon A}{\Gamma \vdash \Delta, U \cup V\colon A}\,(C_R)$$

$$\frac{\Gamma \vdash \Delta, S\colon A \qquad \Gamma', A^x \vdash S'_1\colon \Delta'_1, ..., S'_p\colon \Delta'_p}{\Gamma, \Gamma' \vdash \Delta, S'_1[S/x]\colon \Delta'_1, ..., S'_p[S/x]\colon \Delta'_p}\,(cut)$$

$$\frac{\Gamma, A^x \vdash S_1\colon \Delta_1, ..., S_n\colon \Delta_n, S\colon B}{\Gamma \vdash S_1 \setminus \{x\}\colon \Delta_1, ..., S_n \setminus \{x\}\colon \Delta_n, S \setminus \{x\}\colon (A \to B)}\,(I_\to)$$

$$\frac{\Gamma \vdash \Delta, U\colon (A \to B) \qquad \Gamma' \vdash \Delta', V\colon A}{\Gamma, \Gamma' \vdash \Delta, \Delta', U \cup V\colon B}\,(E_\to)$$

$$\frac{\Gamma \vdash \Delta, S\colon A \wedge B}{\Gamma \vdash \Delta, S\colon A}\,(E^1_\wedge) \qquad \frac{\Gamma \vdash \Delta, S\colon A \wedge B}{\Gamma \vdash \Delta, S\colon B}\,(E^2_\wedge)$$

$$\frac{\Gamma \vdash \Delta, U\colon A \qquad \Gamma' \vdash \Delta', V\colon B}{\Gamma, \Gamma' \vdash \Delta, \Delta', U \cup V\colon A \wedge B}\,(I_\wedge)$$

$$\frac{\Gamma, S\colon A \vee B \vdash \Delta}{\Gamma, S\colon A \vdash \Delta}\,(LE^1_\vee) \qquad \frac{\Gamma, S\colon A \vee B \vdash \Delta}{\Gamma, S\colon B \vdash \Delta}\,(LE^2_\vee)$$

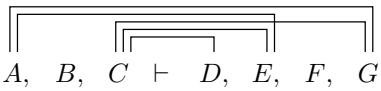
$$\frac{\Gamma, U\colon A \vdash \Delta \qquad \Gamma', V\colon B \vdash \Delta'}{\Gamma, \Gamma', U \cup V\colon A \vee B \vdash \Delta, \Delta'}\,(LI_\vee)$$

$$\frac{\Gamma, U\colon A - B \vdash \Delta \qquad \Gamma', V\colon B \vdash \Delta'}{\Gamma, \Gamma', U \cup V\colon A \vdash \Delta, \Delta'}\,(LE_-)$$

$$\frac{S_1\colon \Gamma_1, ..., S_m\colon \Gamma_m, S\colon A \vdash \Delta, B^\beta}{S_1 \setminus \{\beta\}\colon \Gamma_1, ..., S_m \setminus \{\beta\}\colon \Gamma_m, S \setminus \{\beta\}\colon A - B \vdash \Delta}\,(LI_-)$$

Figure 3.1. Annotations for Symmetric $\mathrm{CND}_{\to\vee\wedge-}$

Example 3.4. Consider the sequent $A^x, B^y, C^z \vdash D^\alpha, E^\beta, F^\gamma, G^\delta$ together with the dependencies $\{(x,\beta),(x,\delta),(z,\alpha),(z,\beta),(z,\delta)\}$:



This annotated sequent can be represented in both forms:

- each conclusion is annotated by a set of hypotheses

$$A^x, B^y, C^z \vdash \{z\}\colon D, \{x,z\}\colon E, \{\}\colon F, \{x,z\}\colon G$$

- each hypothesis is annotated by a set of conclusions

$$\{\beta,\delta\}\colon A, \{\}\colon B, \{\alpha,\beta,\delta\}\colon C \vdash D^\alpha, E^\beta, F^\gamma, G^\delta$$

3.1.1 Annotations for Symmetric $\mathrm{CND}_{\to\vee\wedge-}$

We first present the annotations for the multiplicative context variant of Symmetric $\mathrm{CND}_{\to\vee\wedge-}$. Multiplicative rules are easier to annotate since the dependencies in the context are not modified by the rule. Thus, in these multiplicative rules, we assume that a formula does not occur with the same name in both Γ and Γ' (resp. Δ and Δ'). Note that we use here both representations of dependencies (*i.e.* in some rules conclusions are annotated while in dual rules, hypotheses are annotated) in order to emphasize the symmetry of the definition. The annotated rules of Symmetric $\mathrm{CND}_{\to\vee\wedge-}$ are summarized in figure 3.1.

Note 3.5. We shall use the following abbreviation:

$$U[V/x] \equiv \begin{cases} U \setminus \{x\} \cup V \text{ if } x \in U \\ U \text{ otherwise} \end{cases}$$

We also omit annotations which are not modified by a rule.

3.1.2 Restriction of Symmetric CND$_{\to\vee\wedge-}$ to subtractive logic

Definition 3.6. *In an annotated proof of Symmetric* CND$_{\to\vee\wedge-}$*, we say that an occurrence of the rule* $(I_{\to})$ *is constructive iff* x *does not occur in any* S_i*. We obtain then:*

$$\frac{\Gamma, A^x \vdash S_1 \colon \Delta_1, ..., S_n \colon \Delta_n, V \colon B}{\Gamma \vdash S_1 \colon \Delta_1, ..., S_n \colon \Delta_n, V \setminus \{x\} \colon (A \to B)} \qquad \text{where } x \notin S_1 \cup ... \cup S_n$$

In other words, to stay in a constructive framework, an hypothesis may be discharged onto some conclusion if and only if no other conclusion depends on it. By duality, we obtain the following restriction on (LI_{-}):

Definition 3.7. *In an annotated proof of Symmetric* CND$_{\to\vee\wedge-}$*, we say that an occurrence of the rule* (LI$_{-}$) *is constructive iff* β *does not occur in any* S_i*. We obtain then:*

$$\frac{S_1 \colon \Gamma_1, ..., S_m \colon \Gamma_m, S \colon A \vdash \Delta, B^{\beta}}{S_1 \colon \Gamma_1, ..., S_m \colon \Gamma_m, S \setminus \{\beta\} \colon A - B \vdash \Delta} \qquad \text{where } \beta \notin S_1 \cup ... \cup S_m$$

Definition 3.8. *We say that a proof of Symmetric* CND$_{\to\vee\wedge-}$ *is constructive iff:*

1. *any occurrence* $(I_{\to})$ *is constructive,*
2. *any occurrence* (LI$_{-}$) *is constructive.*

Definition 3.9. *We call Symmetric* SND$_{\to\vee\wedge-}$ *the restriction of Symmetric* CND$_{\to\vee\wedge-}$ *to constructive proofs.*

Theorem 3.10. *Symmetric* SND$_{\to\vee\wedge-}$ *is sound and complete for Subtractive Logic.*

Proof. (sketch) It is clear that Symmetric SND$_{\to\vee\wedge-}$ is a generalization of Symmetric SND$^1_{\to\vee\wedge-}$, thus it is sufficient to prove that any sequent derivable in Symmetric SND$_{\to\vee\wedge-}$ is also derivable in Symmetric SND$^1_{\to\vee\wedge-}$. The idea of the proof is the following one: any derivation of a sequent in Symmetric SND$_{\to\vee\wedge-}$ can be translated into a derivation which does not contain any introduction rule of implication nor any left-hand side introduction rule of subtraction, but which depends only on axioms valid in Symmetric SND$^1_{\to\vee\wedge-}$. For that purpose, we show that (a generalized version of) constructive $(I_{\to})$ "commutes" with every other rule of Symmetric SND$_{\to\vee\wedge-}$. By duality, we know that (a generalized version of) constructive (LI_{-}) also "commutes" with every other rule of Symmetric SND$_{\to\vee\wedge-}$. There is thus a procedure which removes occurrences of constructive $(I_{\to})$ and constructive (LI_{-}) in the original proof and yields a proof of Symmetric SND$^1_{\to\vee\wedge-}$. The details are given in appendix B. □

3.2 Annotations for CND$_{\to\vee\wedge-}$

We derive here the annotations for CND$_{\to\vee\wedge-}$ from the annotations of Symmetric CND$_{\to\vee\wedge-}$. There are several steps to perform:

- derive rules with additive contexts (using the weakening and contraction rules),
- move all the annotations on the right-hand side (each conclusion is annotated by a set of hypotheses),
- derive right-hand side introduction/elimination rules for disjunction and subtraction (using the cut rule).

We shall just give two examples below, the full system of annotations for $\mathrm{CND}_{\to\vee\wedge-}$ is summarized in figure 3.2. Due to lack of space, we shorten $S_1\colon\Delta_1, \ldots S_n\colon\Delta_n$ (with $n \geqslant 0$) simply as $\ldots, S_i\colon\Delta_i \ldots$ in the elimination rule for disjunction.

Example 3.11. Let us derive the annotations for $(E_\vee)$. We first annotate the additive variant of the left-hand side introduction rule for disjunction (where hypotheses are still annotated):

$$\frac{T'_1\colon\Gamma_1, \ldots, T'_p\colon\Gamma_p, U\colon A \vdash \Delta \qquad T''_1\colon\Gamma_1, \ldots, T''_p\colon\Gamma_p, V\colon B \vdash \Delta}{T'_1 \cup T''_1\colon\Gamma_1, \ldots, T'_p \cup T''_p\colon\Gamma_p, U \cup V\colon A \vee B \vdash \Delta}$$

Let us now annotate this rule on the right-hand side:

$$\frac{\Gamma, A^x \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n \qquad \Gamma, B^y \vdash S''_1\colon\Delta_1, \ldots, S''_n\colon\Delta_n}{\Gamma, A \vee B^z \vdash S'_1[\{z\}/x] \cup S''_1[\{z\}/y]\colon\Delta_1, \ldots, S'_n[\{z\}/x] \cup S''_n[\{z\}/y]\colon\Delta_n}$$

Using the (additive variant of the) cut rule, we eventually obtain the right-hand side annotations for the usual (in natural deduction style) elimination rule for disjunction.

$$\frac{\Gamma \vdash \ldots, S_i\colon\Delta_i \ldots, S\colon A \vee B \quad \Gamma, A^x \vdash \ldots, S'_i\colon\Delta_i \ldots, S'\colon C \quad \Gamma, B^y \vdash \ldots, S''_i\colon\Delta_i \ldots, S''\colon C}{\Gamma \vdash \ldots, S_i \cup S'_i[S/x] \cup S''_i[S/y]\colon\Delta_i \ldots, S'[S/x] \cup S''[S/y]\colon C}\,(E_\vee)$$

Note that the apparent inhomogeneous treatment of Δ_i and C comes from the fact that C cannot occur (with the same name) in the leftmost sequent of the premise.

$$A^x \vdash; \{x\}\colon A \quad (ax)$$

$$\frac{\Gamma, U\colon A, V\colon A \vdash \Delta; B}{\Gamma, U \cup V\colon A \vdash \Delta; B}\,(C_L) \qquad \frac{\Gamma \vdash \Delta; B}{\Gamma, A^x \vdash \Delta; B}\,(W_L)$$

$$\frac{\Gamma \vdash \Delta(, U\colon A^\alpha);}{\Gamma \vdash \Delta; U\colon A}\,(activate) \qquad \frac{\Gamma \vdash \Delta(, U\colon A^\alpha); V\colon A}{\Gamma \vdash \Delta, U \cup V\colon A^\alpha;}\,(passivate)$$

$$\frac{\Gamma \vdash S_1\colon\Delta_1, \ldots, S_n\colon\Delta_n; S\colon A \qquad \Gamma, A^x \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n; S'\colon B}{\Gamma \vdash S_1 \cup S'_1[S/x]\colon\Delta_1, \ldots, S_n \cup S'_n[S/x]\colon\Delta_n; S'[S/x]\colon B}\,(cut)$$

$$\frac{\Gamma, A^x \vdash S_1\colon\Delta_1, \ldots, S_n\colon\Delta_n; S\colon B}{\Gamma \vdash S_1 \setminus \{x\}\colon\Delta_1, \ldots, S_n \setminus \{x\}\colon\Delta_n; S \setminus \{x\}\colon(A \to B)}\,(I_\to)$$

$$\frac{\Gamma \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n; U\colon(A \to B) \qquad \Gamma \vdash S''_1\colon\Delta_1, \ldots, S''_n\colon\Delta_n; V\colon A}{\Gamma \vdash S'_1 \cup S''_1\colon\Delta_1, \ldots, S'_n \cup S''_n\colon\Delta_n; U \cup V\colon B}\,(E_\to)$$

$$\frac{\Gamma \vdash \Delta; S\colon A \wedge B}{\Gamma \vdash \Delta; S\colon A}\,(E^1_\wedge) \qquad \frac{\Gamma \vdash \Delta; S\colon A \wedge B}{\Gamma \vdash \Delta; S\colon B}\,(E^2_\wedge)$$

$$\frac{\Gamma \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n; U\colon A \qquad \Gamma \vdash S''_1\colon\Delta_1, \ldots, S''_n\colon\Delta_n; V\colon B}{\Gamma \vdash S'_1 \cup S''_1\colon\Delta_1, \ldots, S'_n \cup S''_n\colon\Delta_n; U \cup V\colon A \wedge B}$$

$$\frac{\Gamma \vdash \Delta; S\colon A}{\Gamma \vdash \Delta; S\colon A \vee B}\,(I^1_\vee) \qquad \frac{\Gamma \vdash \Delta; S\colon B}{\Gamma \vdash \Delta; S\colon A \vee B}\,(I^2_\vee)$$

$$\frac{\Gamma \vdash \ldots, S_i\colon\Delta_i \ldots; S\colon A \vee B \quad \Gamma, A^x \vdash \ldots, S'_i\colon\Delta_i \ldots; S'\colon C \quad \Gamma, B^y \vdash \ldots, S''_i\colon\Delta_i \ldots; S''\colon C}{\Gamma \vdash \ldots, S_i \cup S'_i[S/x] \cup S''_i[S/y]\colon\Delta_i \ldots; S'[S/x] \cup S''[S/y]\colon C}\,(E_\vee)$$

$$\frac{\Gamma \vdash S_1\colon\Delta_1, \ldots, S_n\colon\Delta_n(, S_{n+1}\colon C); S\colon A \qquad \Gamma, B^y \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n(, S_{n+1}\colon C); S'\colon C}{\Gamma \vdash S_1 \cup S'_1[S/y]\colon\Delta_1, \ldots, S_n \cup S'_n[S/y]\colon\Delta_n, S_{n+1} \cup S'_{n+1} \cup S'[S/y]\colon C; S\colon A - B}\,(I_-)$$

$$\frac{\Gamma \vdash S_1\colon\Delta_1, \ldots, S_n\colon\Delta_n; S\colon A - B \qquad \Gamma, A^x \vdash S'_1\colon\Delta_1, \ldots, S'_n\colon\Delta_n; U\colon B}{\Gamma \vdash S_1 \cup S'_1[S/x]\colon\Delta_1, \ldots, S_n \cup S'_n[S/x]\colon\Delta_n; \{\}\colon C}(E_-)$$

Figure 3.2. Annotations for $\mathrm{CND}_{\to\vee\wedge-}$

Example 3.12. Let us derive the annotations for (E_{-}). We first annotate the left-hand side introduction rule for subtraction on the right-hand side:

$$\frac{\Gamma, A^x \vdash S'_1 : \Delta_1, ..., S'_n : \Delta_n, U : B}{\Gamma, A - B^z \vdash S'_1[\{z\}/x] : \Delta_1, ..., S'_n[\{z\}/x] : \Delta_n}\,(LI_{-})$$

By applying the cut rule and then the weakening rule, we obtain the right-hand side annotations for the right-hand side elimination rule for subtraction:

$$\frac{\dfrac{\Gamma \vdash S_1 : \Delta_1, ..., S_n : \Delta_n; S : A - B \qquad \dfrac{\Gamma, A^x \vdash S'_1 : \Delta_1, ..., S'_n : \Delta_n, U : B}{\Gamma, A - B^z \vdash S'_1[\{z\}/x] : \Delta_1, ..., S'_n[\{z\}/x] : \Delta_n}\,(LI_{-})}{\Gamma \vdash S_1 \cup S'_1[S/x] : \Delta_1, ..., S_n \cup S'_n[S/x] : \Delta_n}\,(cut)}{\Gamma \vdash S_1 \cup S'_1[S/x] : \Delta_1, ..., S_n \cup S'_n[S/x] : \Delta_n, \{\} : C}\,(W_R)$$

The technique used in the previous examples can be applied to show that the rules of Symmetric $\mathrm{CND}_{\to\vee\wedge-}$ and the rules of $\mathrm{CND}_{\to\vee\wedge-}$ are interderivable. Consequently, Symmetric $\mathrm{CND}_{\to\vee\wedge-}$ and $\mathrm{CND}_{\to\vee\wedge-}$ are equivalent as systems with annotations:

Proposition 3.13. *An annotated sequent is derivable in Symmetric* $\mathrm{CND}_{\to\vee\wedge-}$ *iff it is derivable in* $\mathrm{CND}_{\to\vee\wedge-}$ *(with the same annotations in both systems).*

3.2.1 Restriction of $\mathrm{CND}_{\to\vee\wedge-}$ to subtractive logic

In order to define the system $\mathrm{SND}_{\to\vee\wedge-}$ we still have to derive the restriction on (E_{-}) from the restriction on (LI_{-}). We obtain thus:

Definition 3.14. *In an annotated proof of* $\mathrm{CND}_{\to\vee\wedge-}$*, we say that an occurrence of the elimination rule for subtraction is constructive iff* $U \subseteq \{x\}$*. The rule becomes then:*

$$\frac{\Gamma \vdash S_1 : \Delta_1, ..., S_n : \Delta_n; S : A - B \qquad \Gamma, A^x \vdash S'_1 : \Delta_1, ..., S'_n : \Delta_n; U : B}{\Gamma \vdash S_1 \cup S'_1[S/x] : \Delta_1, ..., S_n \cup S'_n[S/x] : \Delta_n; \{\} : C}\,(E_{-}) \quad \textit{where } U \subseteq \{x\}$$

This definition extends to proofs:

Definition 3.15. *A proof of* $\mathrm{CND}_{\to\vee\wedge-}$ *is said to be constructive iff:*

1. *any occurrence of the introduction rule for implication is constructive,*
2. *any occurrence of the elimination rule for subtraction is constructive.*

Definition 3.16. *We call* $\mathrm{SND}_{\to\vee\wedge-}$ *the restriction of* $\mathrm{CND}_{\to\vee\wedge-}$ *to constructive proofs.*

By construction, constructive (E_{-}) and constructive (LI_{-}) are interderivable (with the same annotations), which gives us the following proposition:

Proposition 3.17. *An annotated sequent is derivable in Symmetric* $\mathrm{SND}_{\to\vee\wedge-}$ *iff it is derivable in* $\mathrm{SND}_{\to\vee\wedge-}$ *(with the same annotations in both systems).*

Theorem 3.18. $\mathrm{SND}_{\to\vee\wedge-}$ *is sound and complete for Subtractive Logic.*

Proof. By proposition 3.17 and theorem 3.10. □

Since subtractive logic is conservative over intuitionistic logic (see [Crolard, 2001] for instance), we also have the following corollary:

Corollary 3.19. *A sequent is derivable in* $\mathrm{SND}_{\to\vee\wedge}$ *iff it is valid in intuitionistic logic.*

Example 3.20. In the following example the (only) occurrence of $(I_\to)$ is constructive since there is no link between C and A in the premise, and the derived sequent is thus valid in intuitionistic logic.

$$\dfrac{\dfrac{\dfrac{A\vee B^z\vdash;A\vee B \qquad \dfrac{A^x\vdash;A}{A^x\vdash\{x\}\!:\!A;\{\}B}(W_R)}{A\vee B^z\vdash\{z\}\!:\!A;\{z\}\!:\!B} \qquad B^y\vdash;B}{A\vee B^z\vdash\{z\}\!:\!A;\{z\}\!:\!B}(E_\vee) \qquad C^w\vdash;\{w\}\!:\!C}{A\vee B^z,C^w\vdash\{z\}A;\{z,w\}\!:\!B\wedge C}(I_\wedge)}{A\vee B^z\vdash\{z\}\!:\!A;\{z\}\!:\!C\to(B\wedge C)}(I_\to)$$

4 The typed $\lambda\mu^{\to+\times-}$-calculus

In this section, we briefly present the $\lambda\mu^{\to+\times}$-calculus which is an extension of Parigot's $\lambda\mu$-calculus [Parigot, 1992] where disjunction and conjunction are taken as primitives. Our $\lambda\mu^{\to+\times}$-calculus corresponds to de Groote's $\lambda\mu^{\to\wedge\vee\bot}$-calculus [de Groote, 2001] up to minor differences (namely, we augment the syntax with a primitive **let** and we also consider simplification rules as in Parigot's original calculus). Then we extend this calculus with a primitive subtraction and thus obtain the $\lambda\mu^{\to+\times-}$-calculus. Note that in this "unsafe" calculus coroutines are exactly as expressive as first-class continuations (since a coroutine's context is just a pair $\langle \mathit{value}, \mathit{continuation}\rangle$).

4.1 The $\lambda\mu^{\to+\times}$-calculus

Raw $\lambda\mu^{\to+\times}$-terms $M, N, ...$ are constructed by the following grammar:

$$\begin{array}{rcl} M & ::= & x \mid (M\,N) \mid \lambda x.M \mid \mathbf{let}\, x = M\, \mathbf{in}\, N \mid \mu\alpha M \mid [\beta]M \\ & \mid & \mathbf{inl}\, M \mid \mathbf{inr}\, M \mid \mathbf{case}\, M\, \mathbf{of}\, (\mathbf{inl}\, x)\mapsto P \mid (\mathbf{inr}\, y)\mapsto Q \\ & \mid & \langle M, N\rangle \mid \mathbf{fst}\, M \mid \mathbf{snd}\, M \end{array}$$

where $x, y, z, ...$ range over variables and $\alpha, \beta, \gamma, ...$ range over names. Besides, we call *simple* $\lambda\mu^{\to+\times}$*-contexts* the contexts defined by the following grammar:

$$C \;::=\; ([\,]\; t) \mid \mathbf{fst}\,[\,] \mid \mathbf{snd}\,[\,] \mid \mathbf{case}\,[\,]\,\mathbf{of}\,(\mathbf{inl}\, x)\mapsto M \mid (\mathbf{inr}\, y)\mapsto N$$

4.1.1 Reduction rules

Detour-reduction

a) $(\lambda x.u\; t) \;\rightsquigarrow\; u\{t/x\}$

b) $\mathbf{fst}\,\langle t, u\rangle \;\rightsquigarrow\; t$

c) $\mathbf{snd}\,\langle t, u\rangle \;\rightsquigarrow\; u$

d) $\mathbf{case}\,(\mathbf{inl}\, t)\,\mathbf{of}\,(\mathbf{inl}\, x)\mapsto u \mid (\mathbf{inr}\, y)\mapsto v \;\rightsquigarrow\; u\{t/x\}$

e) $\mathbf{case}\,(\mathbf{inr}\, t)\,\mathbf{of}\,(\mathbf{inl}\, x)\mapsto u \mid (\mathbf{inr}\, y)\mapsto v \;\rightsquigarrow\; v\{t/y\}$

f) $\mathbf{let}\, x = t\, \mathbf{in}\, u \;\rightsquigarrow\; u\{t/x\}$

where $u\{t/x\}$ stands for the usual capture-avoiding substitution.

Structural reduction (μ-reduction and permutative rule)

a) $C[\mu\alpha.u] \;\rightsquigarrow\; \mu\alpha.u\{\alpha \hookleftarrow C[\,]\}$

b) $C[\mathbf{case}\; t\; \mathbf{of}\; (\mathbf{inl}\, x)\mapsto u \mid (\mathbf{inr}\, y)\mapsto v] \;\rightsquigarrow\; \mathbf{case}\; t\; \mathbf{of}\; (\mathbf{inl}\, x)\mapsto C[u] \mid (\mathbf{inr}\, y)\mapsto C[v]$

$$x\colon A^x \vdash ;\, A$$

$$\frac{t\colon \Gamma \vdash \Delta;\, B}{t\colon \Gamma, A^x \vdash \Delta;\, B}\,(W_L) \qquad \frac{t\colon \Gamma \vdash \Delta;\, B}{t\colon \Gamma \vdash \Delta, A^\alpha\,;\, B}\,(W_R)$$

$$\frac{t\colon \Gamma \vdash \Delta(, A^\alpha);}{\mu\alpha.t\colon \Gamma \vdash \Delta;\, A}\,(activate) \qquad \frac{t\colon \Gamma \vdash \Delta(, A^\alpha);\, A}{[\alpha]t\colon \Gamma \vdash \Delta, A^\alpha;}\,(passivate)$$

$$\frac{t\colon \Gamma, A^x \vdash \Delta;\, B}{\lambda x.t\colon \Gamma \vdash \Delta;\, A \to B}\,(I_\to) \qquad \frac{u\colon \Gamma \vdash \Delta;\, A \to B \qquad v\colon \Gamma \vdash \Delta;\, A}{(u\ v)\colon \Gamma \vdash \Delta;\, B}\,(E_\to)$$

$$\frac{t\colon \Gamma \vdash \Delta;\, A}{\mathbf{inl}\ t\colon \Gamma \vdash \Delta;\, A \vee B}\,(I^1_\vee) \qquad \frac{t\colon \Gamma \vdash \Delta;\, B}{\mathbf{inr}\ t\colon \Gamma \vdash \Delta;\, A \vee B}\,(I^2_\vee)$$

$$\frac{t\colon \Gamma \vdash \Delta;\, A \vee B \qquad u\colon \Gamma, A^x \vdash \Delta;\, C \qquad v\colon \Gamma, B^y \vdash \Delta;\, C}{\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\ x) \mapsto u \mid (\mathbf{inr}\ y) \mapsto v\colon \Gamma \vdash \Delta;\, C}\,(E_\vee)$$

$$\frac{t\colon \Gamma \vdash \Delta;\, A \wedge B}{\mathbf{fst}\ t\colon \Gamma \vdash \Delta;\, A}\,(E^1_\wedge) \qquad \frac{t\colon \Gamma \vdash \Delta;\, A \wedge B}{\mathbf{snd}\ t\colon \Gamma \vdash \Delta;\, B}\,(E^2_\wedge)$$

$$\frac{t\colon \Gamma \vdash \Delta;\, A \qquad u\colon \Gamma \vdash \Delta;\, B}{\langle t, u\rangle\colon \Gamma \vdash \Delta;\, A \wedge B}\,(I_\wedge)$$

$$\frac{u\colon \Gamma \vdash \Delta;\, A \qquad t\colon \Gamma, A^x \vdash \Delta;\, B}{\mathbf{let}\ x = u\ \mathbf{in}\ t\colon \Gamma \vdash \Delta;\, B}\,(cut)$$

Figure 4.1. The $\lambda\mu^{\to+\times}$-calculus

where $C[\,]$ ranges over simple $\lambda\mu^{\to+\times}$-contexts and $M\{\alpha \hookleftarrow C[\,]\}$ denotes the structural substitution which is inductively defined as follows, where M^* stands for $M\{\alpha \hookleftarrow C[\,]\}$:

$$\begin{aligned}
x^* &\equiv x\\
(\lambda x.t)^* &\equiv \lambda x.t^*\\
(t\ u)^* &\equiv (t^*\ u^*)\\
(\mathbf{fst}\ t)^* &\equiv \mathbf{fst}\ t^* \text{ and } (\mathbf{snd}\ t)^* \equiv \mathbf{snd}\ t^*\\
\langle t, u\rangle^* &\equiv \langle t^*, u^*\rangle\\
(\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\ x) \mapsto u \mid (\mathbf{inr}\ y) \mapsto v)^* &\equiv \mathbf{case}\ t^*\ \mathbf{of}\ (\mathbf{inl}\ x) \mapsto u^* \mid (\mathbf{inr}\ y) \mapsto v^*\\
(\mathbf{inl}\ t)^* &\equiv \mathbf{inl}\ t^* \text{ and } (\mathbf{inr}\ t)^* \equiv \mathbf{inr}\ t^*\\
(\mathbf{let}\ x = t\ \mathbf{in}\ u)^* &\equiv \mathbf{let}\ x = t^*\ \mathbf{in}\ u^*\\
(\mu\gamma.u)^* &\equiv \mu\gamma.u^*\\
([\alpha]t)^* &\equiv [\alpha]C[t^*]\\
([\gamma]t)^* &\equiv [\gamma]t^* \text{ if } \gamma \neq \alpha
\end{aligned}$$

Simplification

a) $\mu\alpha[\alpha]u \rightsquigarrow u$ if α does not occur in u

b) $[\beta]\mu\alpha.t \rightsquigarrow t\{\beta/\alpha\}$

Remark 4.1. We shall often refer to the following macro-definitions: **get-context** $\alpha\ t \equiv \mu\alpha[\alpha]t$ and **set-context** $\alpha\ t \equiv \mu\delta[\alpha]t$ where δ is a name which does not occur in t (see [Crolard, 1999] for a study of the sublanguage obtained when we restrict the $\lambda\mu$-terms to these operators, where they are called respectively **catch** and **throw**). Our terminology in this paper comes from [Group, 1997].

4.2 Typing rules for the $\lambda\mu^{\to+\times}$-calculus

The typing rules for the $\lambda\mu^{\to+\times}$-calculus are summarized in figure 4.1. Note that the typing rules for **get-context** and **set-context** are easily derivable. They correspond respectively to the right-hand side weakening

and contraction rules :

$$\frac{t{:}\,\Gamma \vdash \Delta; A}{\textbf{set-context}\ \alpha\ t{:}\,\Gamma \vdash \Delta, A^{\alpha}; B}\,(W_R) \qquad \frac{t{:}\,\Gamma \vdash \Delta, A^{\alpha}; A}{\textbf{get-context}\ \alpha\ t{:}\,\Gamma \vdash \Delta; A}\,(C_R)$$

Remark 4.2. In order to deal with the negation, we add a macro-definition called **abort** (following de Groote [de Groote, 1994]) defined as **set-context** $\varepsilon\ t$ where ε is a free name (the name of $\bot$ given in section 2.1) considered as a constant (see [1] for more details). Here is the typing rule for **abort**:

$$\frac{t{:}\,\Gamma \vdash \Delta; \bot}{\textbf{abort}\ t{:}\,\Gamma \vdash \Delta; C}\,(\bot_E)$$

4.3 The $\lambda\mu^{\rightarrow+\times-}$-calculus

4.3.1 Defining $A - B$ as $A \wedge \neg B$

We have shown in section 2.1 that the introduction/elimination rules for subtraction are derivable if we define $A - B$ as $A \wedge \neg B$. Since we are looking for a term calculus for $\mathrm{CND}_{\rightarrow\vee\wedge-}$, let us first annotate these proofs with $\lambda\mu^{\rightarrow+\times}$-terms and consider the macro-definitions we obtain.

Derivation of the introduction rule

$$\frac{t{:}\,\Gamma \vdash \Delta; A \qquad \dfrac{\dfrac{x{:}\,\Gamma, B^{x} \vdash \Delta; B}{\textbf{set-context}\ \beta\ x{:}\,\Gamma, B^{x} \vdash \Delta, B^{\beta}; \bot}\,(set)}{\lambda x.\textbf{set-context}\ \beta\ x{:}\,\Gamma \vdash \Delta, B^{\beta}; \neg B}\,(I_{\rightarrow})}{(t, \lambda x.\textbf{set-context}\ \beta\ x){:}\,\Gamma \vdash \Delta, B^{\beta}; A \wedge \neg B}\,(I_{\wedge})$$

Derivation of the elimination rule

$$\frac{t{:}\,\Gamma \vdash \Delta; A \wedge \neg B \qquad \dfrac{\dfrac{k{:}\,\neg B^{k} \vdash \neg B \qquad u{:}\,\Gamma, A^{x} \vdash \Delta; B}{(k\ u){:}\,\Gamma, A^{x}, \neg B^{k} \vdash \Delta; \bot}\,(E_{\rightarrow})}{\textbf{abort}\ (k\ u){:}\,\Gamma, A^{x}, \neg B^{k} \vdash \Delta; C}\,(E_{\bot})}{\textbf{match}\ t\ \textbf{with}\ (x, k) \mapsto \textbf{abort}\ (k\ u){:}\,\Gamma \vdash \Delta; C}\,(E_{\wedge})$$

Remark 4.3.

- It is easy to define the **match** instruction used in the last inference rule with the projections, for instance as follows:

  $$\textbf{match}\ t\ \textbf{with}\ (x, y) \mapsto u \equiv \textbf{let}\ x = (\textbf{fst}\ t)\ \textbf{in}\ (\textbf{let}\ y = (\textbf{snd}\ t)\ \textbf{in}\ u)$$

  However, since we also expect the subformula property for the $\lambda\mu^{\rightarrow+\times-}$-calculus, we need permutative reduction rules related to **match** (in order to derive the permutative reduction rules related to **resume**) and these rules are usually not derivable when pattern matching is defined using the projections. This is why we need to consider an extension of the $\lambda\mu^{\rightarrow+\times\bot}$-calculus with a new product $\otimes$ and built-in pattern matching. The resulting calculus (called the $\lambda\mu^{\rightarrow+\times\otimes\bot}$-calculus) is studied in appendix A.

- Informally, the introduction rules builds a pair (*value*, *continuation*) while the elimination rule opens a pair (*value*, *continuation*) and invokes the *continuation* on some $\lambda\mu$-term which may also use the *value*. We shall explain in section 5.8 why we interpret such a pair as a first-class coroutine, and the *value* as its local environment. For the time being, we just define the following macros:

  $$\begin{aligned} \textbf{make-coroutine}\ t\ \alpha &\equiv (t, \lambda z.\textbf{set-context}\ \alpha\ z) \\ \textbf{resume}\ t\ \textbf{with}\ x \mapsto u &\equiv \textbf{match}\ t\ \textbf{with}\ (x, k) \mapsto \textbf{abort}\ (k\ u) \end{aligned}$$

  The derived typing rules for these macro-definitions are then:

  $$\frac{t{:}\,\Gamma \vdash \Delta; A}{\textbf{make-coroutine}\ t\ \beta{:}\,\Gamma \vdash \Delta, B^{\beta}; A - B} \qquad \frac{t{:}\,\Gamma \vdash \Delta; A - B \quad u{:}\,\Gamma, A^{x} \vdash \Delta; B}{\textbf{resume}\ t\ \textbf{with}\ x \mapsto u{:}\,\Gamma \vdash \Delta; C}$$

- We said in the introduction that in the $\lambda\mu$-calculus, a name α may be reified as the first-class continuation $\lambda z.\mathbf{set\text{-}context}\ \alpha\ z$. However, this last syntactic form is not closed under μ-reduction.

  For instance, the term $((\mathbf{get\text{-}context}\ \alpha\ (\lambda z.\mathbf{set\text{-}context}\ \alpha\ z))\, u)$ is a μ-redex, and its contractum is $(\mathbf{get\text{-}context}\ \alpha\ ((\lambda z.\mathbf{set\text{-}context}\ \alpha\ (z\, u))\, u))$. We shall thus consider a more general form $\lambda z.\mathbf{set\text{-}context}\ \alpha\ C[z]$ where $C[\,]$ is a continuation context (see definition 4.4).

We are now ready to extend the $\lambda\mu^{\to+\times}$-calculus with first-class coroutines.

4.3.2 Syntax of the $\lambda\mu^{\to+\times-}$-calculus

We add two term constructors to the syntax of the raw $\lambda\mu^{\to+\times}$-calculus (where $C[\,]$ ranges over arbitrary contexts):

$$M\ ::=\ \ldots \mid \mathbf{make\text{-}coroutine}\, M\, (C[\,], \alpha) \mid \mathbf{resume}\ M\ \mathbf{with}\ x \mapsto N$$

We also define the *simple* $\lambda\mu^{\to+\times-}$*-contexts* by extending the grammar of simple $\lambda\mu^{\to+\times}$-contexts:

$$C\ ::=\ \ldots \mid \mathbf{resume}\ [\,]\ \mathbf{with}\ x \mapsto N$$

4.3.3 Continuation contexts

Definition 4.4. *Continuation contexts are defined by the following grammar:*

$$\begin{array}{rcl} C & ::= & [\,] \ \mid\ (C[\,]\ t) \\ & \mid & \mathbf{fst}\ C[\,] \ \mid\ \mathbf{snd}\ C[\,] \\ & \mid & \mathbf{case}\ C[\,]\ \mathbf{of}\ (\mathbf{inl}\ x) \mapsto M \mid (\mathbf{inr}\ y) \mapsto N \\ & \mid & \mathbf{resume}\ C[\,]\ \mathbf{with}\ x \mapsto N \end{array}$$

Remark 4.5. We shall abbreviate as C_α a pair $(C[\,], \alpha)$ where $C[\,]$ is a continuation context. Note that if $C[\,]$ is a simple $\lambda\mu^{\to+\times-}$-context and $K[\,]$ is a continuation context then $C[K[\,]]$ is also a continuation context, and thus $C[K]_\alpha$ is an abbreviation for $(C[K[\,]], \alpha)$.

4.3.4 Reduction rules

We extend the reduction rules of the $\lambda\mu^{\to+\times}$-calculus with this new detour-reduction rule:

g) $\mathbf{resume}\ (\mathbf{make\text{-}coroutine}\, t\, C_\alpha)\ \mathbf{with}\ x \mapsto u \rightsquigarrow \mathbf{set\text{-}context}\ \alpha\ C[u\{t/x\}]$

and the structural reduction is now defined by the following rules:

a) $C[\mu\alpha.u] \ \rightsquigarrow\ \mu\alpha.u\{\alpha \hookleftarrow C[\,]\}$

b) $C[\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v] \ \rightsquigarrow\ \mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto C[u] \mid (\mathbf{inr}\, y) \mapsto C[v]$

c) $C[\mathbf{resume}\ t\ \mathbf{with}\ x \mapsto u] \ \rightsquigarrow\ \mathbf{resume}\ t\ \mathbf{with}\ x \mapsto u$

where $C[\,]$ ranges over simple $\lambda\mu^{\to+\times-}$-contexts. We also extend the inductive definition of the structural substitution $t\{\alpha \hookleftarrow C[\,]\}$ to $\lambda\mu^{\to+\times-}$-terms and continuation contexts as follows, where again M^* stands for $M\{\alpha \hookleftarrow C[\,]\}$:

$$\begin{array}{rcl} [\,]^* & \equiv & [\,] \\ (\mathbf{make\text{-}coroutine}\, t\, K_\alpha)^* & \equiv & \mathbf{make\text{-}coroutine}\, t^*\, C[K^*]_\alpha \\ (\mathbf{make\text{-}coroutine}\, t\, K_\gamma)^* & \equiv & \mathbf{make\text{-}coroutine}\, t^*\, K^*_\gamma \text{ if } \gamma \neq \alpha \end{array}$$

$$(\textbf{resume}\ t\ \textbf{with}\ x \mapsto u)^* \ \equiv\ \textbf{resume}\ t^*\ \textbf{with}\ x \mapsto u^*$$

Remark 4.6. By definition of the structural substitution, the form $\textbf{make-coroutine}\, t\, C_\alpha$ where $C[\,]$ is a continuation context is preserved by the reduction rules of the $\lambda\mu^{\to+\times-}$-calculus. We shall thus restrict the set of raw $\lambda\mu^{\to+\times-}$-terms to those terms.

4.4 Typing rules for coroutines

The typing rules for **make-coroutine** and **resume** are respectively introduction and elimination rules for subtraction:

$$\frac{t\colon \Gamma \vdash \Delta; A \quad C[x]\colon \Gamma, D^x \vdash \Delta; B}{\textbf{make-coroutine}\, t\, C_\beta \colon \Gamma \vdash \Delta, B^\beta; A - D}\,(I_-) \qquad \frac{t\colon \Gamma \vdash \Delta; A - B \quad u\colon \Gamma, A^x \vdash \Delta; B}{\textbf{resume}\ t\ \textbf{with}\ x \mapsto u \colon \Gamma \vdash \Delta; C}(E_-)$$

Remark 4.7. The typing rule for **make-coroutine** given in the remark 4.3 corresponds to the particular case $C[\,] = [\,]$ (but we have proved in section 2 that from a logical standpoint these rules are equivalent).

4.5 Strong normalization and the Church-Rosser property

Theorem 4.8. *The typed $\lambda\mu^{\to+\times-}$-calculus is strongly normalizing and enjoys the Church-Rosser property.*

Proof. (sketch) Recall that the typed $\lambda\mu^{\to+\times\otimes\bot}$-calculus is an extension of the $\lambda\mu^{\to+\times\bot}$-calculus with pattern matching (for the tensor product$\otimes$) which is strongly normalizing and enjoys the Church-Rosser property (see appendix A). Let us denote by Φ the translation from the $\lambda\mu^{\to+\times-}$-calculus into the $\lambda\mu^{\to+\times\otimes\bot}$-calculus defined by the following macro-definitions:

$$\begin{array}{rcl} \textbf{make-coroutine}\, t\, C_\alpha & \equiv & (t, \lambda z.\textbf{set-context}\ \alpha\ C[z]) \\ \textbf{resume}\ t\ \textbf{with}\ x \mapsto u & \equiv & \textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto\ \textbf{abort}\,(k\ u) \end{array}$$

It is sufficient to check the following properties:

1. Φ is a morphism for the reduction: if $u \rightsquigarrow v$ then $\Phi(u) \rightsquigarrow^+ \Phi(v)$
2. Φ preserves normal forms: if u is a normal $\lambda\mu^{\to+\times-}$-term then $\Phi(u)$ is a normal $\lambda\mu^{\to+\times\otimes\bot}$-term.
3. Φ is injective on normal forms.

Indeed, from the first property, we can derive the strong normalization of the $\lambda\mu^{\to+\times-}$-calculus from the strong normalization of the $\lambda\mu^{\to+\times\otimes\bot}$-calculus. From the second and third properties, we obtain the uniqueness of the normal form (*i.e.* the Church-Rosser property). The proof that Φ is a morphism for the reduction is given in appendix A. □

4.6 Normal forms and the subformula property

We end this section by proving the subformula property (which is derived as usual from a characterization of the normal terms).

Proposition 4.9. *Given the derivation, in system* $\mathrm{CND}_{\to\vee\wedge-}$*, of some typing judgement $t : \Gamma \vdash \Delta; A$, if t is normal then every type occurring in the derivation is either a subformula of a type occurring in Γ or Δ, or a*

subformula of A.

Proof. Since the subformula property always holds for introduction rules (and weakening/contraction rules), we just have to check the property for occurrences of elimination rules in a normal proof. Let us then call *applicative contexts* the contexts defined by the following grammar:

$$A \ ::= [\,] \ \mid \ (A\ t) \ \mid \ \mathbf{fst}\ A \ \mid \ \mathbf{snd}\ A$$

In a well-typed normal $\lambda\mu^{\rightarrow+\times-}$-term, any occurrence of an application or a projection has the form $A[x]$ (where x is a variable). Indeed, in an application $(u\,v)$ and in a projection $\mathbf{fst}\ u$ or $\mathbf{snd}\ u$, the term u must be either a variable, or again a projection or an application (otherwise we obtain a detour-redex or a redex for one of the structural rules). It is easy to prove by induction that in the typing judgement of an applicative context $A[x]$: $\Gamma, F^x \vdash \Delta; D$, the formula D is a subformula of F.

Now, in a well-typed normal $\lambda\mu^{\rightarrow+\times-}$-term, in any subterm of the form $\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\,x) \mapsto u \mid (\mathbf{inr}\,y) \mapsto v$ or $\mathbf{resume}\ t\ \mathbf{with}\ x \mapsto u$, the term t has the form $A[z]$ (where z is a variable). Indeed, otherwise we obtain a detour-redex or redex for one of the structural rules. Consequently, any occurrence of an elimination rule for $\vee$ (resp. $-$) has the following form:

Elimination rule for $\vee$

$$\frac{A[z]\colon \Gamma, F^z \vdash \Delta; B \vee C \qquad u\colon \Gamma, B^x \vdash \Delta; D \qquad v\colon \Gamma; C^y \vdash \Delta; D}{\mathbf{case}\ A[z]\ \mathbf{of}\ \ (\mathbf{inl}\,x) \mapsto u \mid (\mathbf{inr}\,y) \mapsto v\colon \Gamma, F^z \vdash \Delta; D}$$

Elimination rule for $-$

$$\frac{A[z]\colon \Gamma, F^z \vdash \Delta; A - B \qquad u\colon \Gamma, A^x \vdash \Delta; B}{\mathbf{resume}\ A[z]\ \mathbf{with}\ x \mapsto u\colon \Gamma, F^z \vdash \Delta; C}$$

Then one can easily check that the subformula property holds for these rules. □

5 The safe $\lambda\mu$-calculus

From a computational standpoint, the restriction of the introduction rule for $\rightarrow$ to at most one conclusion amounts to requiring that in any subterm $\lambda x.u$ of a term t, u is μ-closed (there is no free name in u). This restriction is clearly not closed under reduction. For instance, if u contains a free name (take for instance $u = \mu\beta[\alpha]\lambda z.z$):

$$((\lambda y \lambda x.y)\ u) \rightsquigarrow \lambda x.u$$

then the contractum is not μ-closed. We define in this section a weaker restriction on $\lambda\mu^{\rightarrow+\times}$-terms which is closed under reduction. This restriction has actually been derived from the restriction on sequents given in section 3.1.2. The formal relation between the two definitions is stated by theorem 5.9.

5.1 The safe $\lambda\mu^{\rightarrow+\times-}$-calculus

In this section we introduce the concept of "scope of a name with respect to variables" which corresponds to the variables shared by a coroutine. This leads to the formal definition of a $\lambda\mu^{\rightarrow+\times}$-term in which a coroutine does not access the local environment of another coroutine (see remark 4.7). We shall say that such a $\lambda\mu^{\rightarrow+\times-}$-term is *safe with respect to coroutine contexts*. Then we give the formal definition of a $\lambda\mu^{\rightarrow+\times-}$-term *safe with respect to first-class coroutines*.

- $\mathcal{S}_{[]}(x) = \{x\}$
  $\mathcal{S}_{\delta}(x) = \emptyset$
- $\mathcal{S}_{[]}(\lambda x.u) = \mathcal{S}_{[]}(u) \backslash \{x\}$
  $\mathcal{S}_{\delta}(\lambda x.u) = \mathcal{S}_{\delta}(u) \backslash \{x\}$
- $\mathcal{S}_{[]}(u\, v) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_{[]}(v)$
  $\mathcal{S}_{\delta}(u\, v) = \mathcal{S}_{\delta}(u) \cup \mathcal{S}_{\delta}(v)$
- $\mathcal{S}_{[]}([\alpha]u) = \emptyset$
  $\mathcal{S}_{\delta}([\alpha]u) = \mathcal{S}_{\delta}(u)$ for any $\delta \neq \alpha$ and $\mathcal{S}_{\alpha}([\alpha]u) = \mathcal{S}_{\alpha}(u) \cup \mathcal{S}_{[]}(u)$
- $\mathcal{S}_{[]}(\mu\alpha.u) = \mathcal{S}_{\alpha}(u)$
  $\mathcal{S}_{\delta}(\mu\alpha.u) = \mathcal{S}_{\delta}(u)$
- $\mathcal{S}_{[]}(\mathbf{inl}\, u) = \mathcal{S}_{[]}(u)$ and $\mathcal{S}_{[]}(\mathbf{inr}\, u) = \mathcal{S}_{[]}(u)$
  $\mathcal{S}_{\delta}(\mathbf{inl}\, u) = \mathcal{S}_{\delta}(u)$ and $\mathcal{S}_{\delta}(\mathbf{inr}\, u) = \mathcal{S}_{\delta}(u)$
- $\mathcal{S}_{[]}(\mathbf{case}\, w\, \mathbf{of}\, (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v) = \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(w)/x] \cup \mathcal{S}_{[]}(v)[\mathcal{S}_{[]}(w)/y]$
  $\mathcal{S}_{\delta}(\mathbf{case}\, w\, \mathbf{of}\, (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v) = \mathcal{S}_{\delta}(u)[\mathcal{S}_{[]}(w)/x] \cup \mathcal{S}_{\delta}(v)[\mathcal{S}_{[]}(w)/y] \cup \mathcal{S}_{\delta}(w)$
- $\mathcal{S}_{[]}(\langle t, u\rangle) = \mathcal{S}_{[]}(t) \cup \mathcal{S}_{[]}(u)$
  $\mathcal{S}_{\delta}(\langle t, u\rangle) = \mathcal{S}_{\delta}(t) \cup \mathcal{S}_{\delta}(u)$
- $\mathcal{S}_{[]}(\mathbf{fst}\, u) = \mathcal{S}_{[]}(u)$ and $\mathcal{S}_{[]}(\mathbf{snd}\, u) = \mathcal{S}_{[]}(u)$
  $\mathcal{S}_{\delta}(\mathbf{fst}\, u) = \mathcal{S}_{\delta}(u)$ and $\mathcal{S}_{\delta}(\mathbf{snd}\, u) = \mathcal{S}_{\delta}(u)$
- $\mathcal{S}_{[]}(\mathbf{let}\, x = v\, \mathbf{in}\, u) = \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(v)/x]$
  $\mathcal{S}_{\delta}(\mathbf{let}\, x = v\, \mathbf{in}\, u) = \mathcal{S}_{\delta}(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_{\delta}(v)$
- $\mathcal{S}_{[]}([\,]) = \emptyset$
  $\mathcal{S}_{\delta}([\,]) = \emptyset$
- $\mathcal{S}_{[]}(\mathbf{make\text{-}coroutine}\, t\, C_{\alpha}) = \mathcal{S}_{[]}(t)$
  $\mathcal{S}_{\alpha}(\mathbf{make\text{-}coroutine}\, t\, C_{\alpha}) = \mathcal{S}_{\alpha}(t) \cup \mathcal{S}_{\alpha}(C[\,]) \cup \mathcal{S}_{[]}(t) \cup \mathcal{S}_{[]}(C[\,])$
  $\mathcal{S}_{\delta}(\mathbf{make\text{-}coroutine}\, t\, C_{\alpha}) = \mathcal{S}_{\delta}(t) \cup \mathcal{S}_{\delta}(C[\,])$ for any $\delta \neq \alpha$
- $\mathcal{S}_{[]}(\mathbf{resume}\, c\, \mathbf{with}\, x \mapsto u) = \mathcal{S}_{[]}(u) \backslash \{x\} \cup \mathcal{S}_{[]}(c)$
  $\mathcal{S}_{\delta}(\mathbf{resume}\, c\, \mathbf{with}\, x \mapsto u) = \mathcal{S}_{\delta}(u)[\mathcal{S}_{[]}(c)/x] \cup \mathcal{S}_{\delta}(c)$

Figure 5.1. Shared variables

We call $\mathcal{S}_{[]}(t)$ the set of free variables that occur in the scope of the current coroutine (which we dubbed $[]$) and $\mathcal{S}_{\delta}(t)$ the set of the variables that occur in the scope of a coroutine δ in t. It is easy to check by induction on t that $\mathcal{S}_{\delta}(t) = \emptyset$ if δ does not occur free in t.

Definition 5.1. *For any $\lambda\mu^{\rightarrow+\times-}$-term t, the inductive definition of the sets $\mathcal{S}_{[]}(t)$ and $\mathcal{S}_{\delta}(t)$ is given in figure •.*

Remark 5.2. In the particular case of **set-context** and **get-context**, we obtain the following definition:

- If the $\lambda\mu^{\rightarrow+\times-}$-term is **get-context** $\alpha\, u$ (*i.e.* $\mu\alpha[\alpha]u$) then:
  $\mathcal{S}_{[]}(\mu\alpha[\alpha]u) = \mathcal{S}_{\alpha}([\alpha]u) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_{\alpha}(u)$
  $\mathcal{S}_{\delta}(\mu\alpha[\alpha]u) = \mathcal{S}_{\delta}([\alpha]u) = \mathcal{S}_{\delta}(u)$
- If the $\lambda\mu^{\rightarrow+\times-}$-term t is **set-context** $\alpha\, u$ (*i.e.* $\mu\beta[\alpha]u$ where β does not occur in $[\alpha]u$) then:
  $\mathcal{S}_{[]}(\mu\beta[\alpha]u) = \mathcal{S}_{\beta}([\alpha]u) = \emptyset$
  $\mathcal{S}_{\delta}(\mu\beta[\alpha]u) = \mathcal{S}_{\delta}([\alpha]u) = \mathcal{S}_{\delta}(u)$ for any $\delta \neq \alpha$ and $\mathcal{S}_{\alpha}(\mu\beta[\alpha]u) = \mathcal{S}_{\alpha}(u) \cup \mathcal{S}_{[]}(u)$

Remark 5.3. Given a $\lambda\mu^{\rightarrow+\times-}$-term, a variable may occur in the scope of *several* coroutines (including the current coroutine). We shall say that such a variable is *shared* between several coroutines.

Definition 5.4. *A $\lambda\mu^{\to+\times-}$-term t is **safe with respect to coroutine contexts** iff for any subterm of t which has the form $\lambda x.u$, for any free name δ of u, $x \notin \mathcal{S}_\delta(u)$.*

Remark 5.5. In safe $\lambda\mu^{\to+\times}$-terms, the usual abbreviation $(\lambda x.t)\,u$ is no longer equivalent to $\mathbf{let}\, x = u \,\mathbf{in}\, t$ since in $(\lambda x.t)\,u$, the variable x may not occur in the scope of some name in t: this declaration of x is local to the current coroutine. On the other hand, the definition of x in $\mathbf{let}\, x = u \,\mathbf{in}\, t$ is global: any coroutine may access x in t.

Note that although global definitions are useful from a programming perspective, in this paper, the main purpose of **let** is just to avoid using the (meta-level) substitution in the term interpretation of the cut rule.

Example 5.6. As expected, a first class continuation like $\lambda y.\mathbf{set\text{-}context}\ \beta\ y$ is not safe with respect to coroutine contexts. On the other hand the following example is safe:

$$\lambda f.\mathbf{get\text{-}context}\ \alpha\ \lambda x.\ \mathbf{get\text{-}context}\ \beta\ \mathbf{set\text{-}context}\ \alpha\ (f\ \ (\mathbf{set\text{-}context}\ \beta\ x))$$

Indeed, in context α the variable f is visible (even if x is not) while in context β they are both visible.

Definition 5.7. *A $\lambda\mu^{\to+\times-}$-term t is **safe with respect to first-class coroutines** (or just "safe" for short) iff:*

1. *for any subterm of t which has the form $\lambda x.u$, for any free name δ of u, $x \notin \mathcal{S}_\delta(u)$ (i.e. t is safe with respect to coroutine contexts)*
2. *for any subterm of t which has the form $\mathbf{resume}\, c\, \mathbf{with}\, x \mapsto u$, $\mathcal{S}_{[]}(u) \subseteq \{x\}$*

Remark 5.8. If we violate one of these restrictions, we recover the (unsafe) $\lambda\mu^{\to+\times}$-calculus (and classical logic as a type system). For the first restriction, it is obvious. For the second restriction, consider the following $\lambda\mu^{\to+\times-}$-term:

$$\lambda y.(\mathbf{resume}\ (\mathbf{make\text{-}coroutine}\, e\ \beta)\ \mathbf{with}\ x \mapsto y) \rightsquigarrow \lambda y.\mathbf{set\text{-}context}\ \beta\ y$$

The contractum is indeed a first-class continuation. When it is resumed, a first-class coroutine is allowed to access only its own local data (which is packed together with the continuation). This is why we claim that such a pair (*value*, *continuation*) corresponds actually (for safe $\lambda\mu^{\to+\times-}$-terms) to a pair (*environment*, *continuation*) where the *environment* contains the only data visible in the coroutine when it resumes.

5.2 Typing safe $\lambda\mu^{\to+\times-}$-terms in $\mathrm{SND}_{\to\vee\wedge-}$

Proofs of $\mathrm{SND}_{\to\vee\wedge-}$ (*i.e.* constructive proofs of $\mathrm{CND}_{\to\vee\wedge-}$) and safe $\lambda\mu^{\to+\times-}$-terms are related by this theorem (and its corollary):

Theorem 5.9. *Given an annotated derivation, in system $\mathrm{CND}_{\to\vee\wedge-}$, of some typing judgement $t : \Gamma_1^{x_1}, \dots, \Gamma_n^{x_n} \vdash \Delta_1^{\alpha_1}, \dots, \Delta_m^{\alpha_m}; A$, then $x_i \in \mathcal{S}_{\alpha_j}(t)$ iff there is a link between $\Gamma_i^{x_i}$ and $\Delta_j^{\alpha_j}$ and $x_i \in \mathcal{S}_{[]}(t)$ iff there is a link between $\Gamma_i^{x_i}$ and A.*

Proof. We check that we have two inductive definitions of the same dependency relations between hypotheses and conclusions of a sequent (recall that [] is the name of the formula which is on the right-hand side of the semi-colon). Take for instance the rule $(E_\to)$ of $\mathrm{CND}_{\to\vee\wedge-}$:

$$\frac{t : \Gamma \vdash S_1' : \Delta_1^{\delta_1}, \dots, S_n' : \Delta_n^{\delta_n}; U : (A \to B) \qquad u : \Gamma \vdash S_1'' : \Delta_1^{\delta_1}, \dots, S_n'' : \Delta_n^{\delta_n}; V : A}{(t\,u) : \Gamma \vdash S_1' \cup S_1'' : \Delta_1^{\delta_1}, \dots, S_n' \cup S_n'' : \Delta_n^{\delta_n}; U \cup V : B}$$

By induction hypothesis, $S_i' = S_{\delta_i}(t)$, $S_i'' = S_{\delta_i}(u)$, $U = S_{[]}(t)$ and $V = S_{[]}(u)$ and by definition 5.1:

$$\begin{aligned} \mathcal{S}_{[]}(t\,u) &= \mathcal{S}_{[]}(t) \cup \mathcal{S}_{[]}(u) = U \cup V \\ \mathcal{S}_{\delta_i}(t\,u) &= \mathcal{S}_{\delta_i}(t) \cup \mathcal{S}_{\delta_i}(u) = S_i' \cup S_i'' \end{aligned}$$

The only difficulty comes from the introduction rule of subtraction (I_-) but since $C[\,]$ is a continuation context, we know that y occurs in S' and nowhere else. We obtain thus:

$$\frac{t\colon \Gamma \vdash ..., S_i\colon \Delta_i^{\delta_i}, ...(, S_{n+1}\colon C^\beta); S\colon A \qquad C[y]\colon \Gamma, B^y \vdash ..., S_i'\colon \Delta_i^{\delta_i}, ...(, S_{n+1}'\colon C^\beta); S'\colon C}{\mathbf{make\text{-}coroutine}\, t\, C_\beta\colon \Gamma \vdash ..., S_i \cup S_i'\colon \Delta_i^{\delta_i}, ..., S_{n+1} \cup S_{n+1}' \cup S'[S/y]\colon C^\beta; S\colon A - B}\,(I_-)$$

Again, by induction hypothesis, $S_i = S_{\delta_i}(t)$, $S_i' = S_{\delta_i}(C[y]) = S_{\delta_i}(C[\,])$, $S_{n+1} = \mathcal{S}_\beta(t)$, $S_{n+1}' = \mathcal{S}_\beta(C[y]) = \mathcal{S}_\beta(C[\,])$, $S'[S/y] = \mathcal{S}_{[]}(C[y])[S/y] = \mathcal{S}_{[]}(C[\,]) \cup S$ and by definition 5.1:

$$\begin{array}{rcl}
\mathcal{S}_{[]}(\mathbf{make\text{-}coroutine}\, t\, C_\beta) & = & \mathcal{S}_{[]}(t) \;=\; S \\
\mathcal{S}_\beta(\mathbf{make\text{-}coroutine}\, t\, C_\beta) & = & \mathcal{S}_{[]}(t) \cup \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_\beta(t) \cup \mathcal{S}_\beta(C[\,]) \;= S'[S/y] \cup S_{n+1} \cup S_{n+1}' \\
\mathcal{S}_{\delta_i}(\mathbf{make\text{-}coroutine}\, t\, C_\beta) & = & \mathcal{S}_{\delta_i}(t) \cup \mathcal{S}_{\delta_i}(C[\,]) \;=\; S_i \cup S_i'
\end{array}$$

□

Corollary 5.10. *Given a derivation of the typing judgement $t\colon \Gamma \vdash \Delta$ in $\mathrm{CND}_{\to\vee\wedge-}$, if t is safe with respect to first-class coroutines then the derivation of $t\colon \Gamma \vdash \Delta$ belongs to $\mathrm{SND}_{\to\vee\wedge-}$ (and is thus valid in subtractive logic).*

Proof. Let us consider an occurrence of the introduction rule for implication:

$$\frac{u\colon \Gamma, A^x \vdash \Delta_1^{\alpha_1}, ..., \Delta_m^{\alpha_m}; B}{\lambda x.u\colon \Gamma \vdash \Delta_1^{\alpha_1}, ..., \Delta_m^{\alpha_m}; A \to B}$$

Since t is safe with respect to coroutine contexts, for any α_j, $x \notin \mathcal{S}_{\alpha_j}(u)$ and then by theorem 5.9, there is no link between A^x and Δ, thus this occurrence of the introduction rule for implication is constructive. Let us now consider an occurrence of the elimination rule for subtraction:

$$\frac{t\colon \Gamma \vdash \Delta; A - B \qquad u\colon \Gamma, A^x \vdash \Delta; B}{\mathbf{resume}\ t\ \mathbf{with}\ x \mapsto u\colon \Gamma \vdash \Delta; C}$$

Since t is safe with respect to first-class coroutines, we know that $\mathcal{S}_{[]}(u) \subseteq \{x\}$ and then by theorem 5.9, there is no link between Γ and B, thus this occurrence of the elimination rule for subtraction is constructive. □

Corollary 5.11. *Given a derivation of the typing judgement $t\colon \Gamma \vdash \Delta$ in $\mathrm{CND}_{\to\vee\wedge}$, if t is safe with respect to coroutine contexts then the derivation of $t\colon \Gamma \vdash \Delta$ belongs to $\mathrm{SND}_{\to\vee\wedge}$ (and is thus valid in intuitionistic logic).*

Example 5.12. Let us decorate the proof of example 3.20 by $\lambda\mu^{\to+\times-}$-terms, we obtain:

$$\dfrac{\dfrac{\dfrac{z\colon A \vee B^z \vdash; A \vee B \qquad \dfrac{x\colon A^x \vdash; A}{\mathbf{set\text{-}context}\ \alpha\ x\colon A^x \vdash A^\alpha; B} \qquad y\colon B^y \vdash; B}{\mathbf{case}\ z\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto \mathbf{set\text{-}context}\ \alpha\ x \mid (\mathbf{inr}\, y) \mapsto y\colon A \vee B^z \vdash A^\alpha; B}(E_\vee) \qquad w\colon C^w \vdash; C}{\langle \mathbf{case}\ z\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto \mathbf{set\text{-}context}\ \alpha\ x \mid (\mathbf{inr}\, y) \mapsto y, w\rangle\colon A \vee B^z, C^w \vdash A^\alpha; B \wedge C}}{\lambda w.\langle \mathbf{case}\ z\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto \mathbf{set\text{-}context}\ \alpha\ x \mid (\mathbf{inr}\, y) \mapsto y, w\rangle\colon A \vee B^z \vdash A^\alpha; C \to (B \wedge C)}$$

One can check that this term is safe with respect to coroutine contexts, since we have:

$$w \notin \mathcal{S}_\alpha(\langle \mathbf{case}\ z\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto \mathbf{set\text{-}context}\ \alpha\ x \mid (\mathbf{inr}\, y) \mapsto y, w\rangle) = \{z\}$$

and consequently the derivation is valid in intuitionistic logic.

Remark 5.13. Note that the proof-term which decorates the conclusion in the previous example (let us call it t) is not a weakening term according to the definition 11 given by Ritter, Pym and Wallen in [Pym et al., 2000] since otherwise $A \vee B \vdash A$ should be derivable (by lemma 14), and the occurrence of α in t is not a weakening occurrence (according to the same definition) since otherwise $A \vee B \vdash C \to (B \wedge C)$ should be derivable. The term t is thus not intuitionistic according to definition 13.

On the other hand we conjecture that any intuitionistic term (according to definition 13 in [Pym et al., 2000]) is safe with respect to coroutine contexts. Consequently, our notion of "safeness" is likely to allow more $\lambda\mu^{\rightarrow+\times-}$-term as proof terms for sequents valid in intuitionistic logic.

6 Closure under reduction

The remainder of the paper is devoted to proving that the subset of safe $\lambda\mu^{\rightarrow+\times-}$-terms is closed under the reduction rules of the $\lambda\mu^{\rightarrow+\times-}$-calculus. We shall begin with proposition 6.8 which says that the reduction rules of the $\lambda\mu^{\rightarrow+\times-}$-calculus do not provide new dependencies (in safe terms). We first need the following additional lemmas about substitutions and contexts:

Lemma 6.1. *$\mathcal{S}_\delta(t)=\emptyset$ if δ does not occur in t.*

Lemma 6.2. *If u and v are $\lambda\mu^{\rightarrow+\times-}$-terms then:*

- $\mathcal{S}_{[]}(u\{v/x\}) \subseteq \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(v)/x]$
  $\mathcal{S}_\delta(u\{v/x\}) \subseteq \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x]\cup\mathcal{S}_\delta(v)$

Lemma 6.3. *If t is a $\lambda\mu^{\rightarrow+\times-}$-term then:*

- $\mathcal{S}_{[]}(t\{\beta/\alpha\}) \subseteq \mathcal{S}_{[]}(t)$
  $\mathcal{S}_\beta(t\{\beta/\alpha\}) \subseteq \mathcal{S}_\beta(t)\cup\mathcal{S}_\alpha(t)$
  $\mathcal{S}_\delta(t\{\beta/\alpha\}) \subseteq \mathcal{S}_\delta(t)$ *for any* $\delta\neq\beta$

Lemma 6.4. *If t is a $\lambda\mu^{\rightarrow+\times-}$-terms and $C[\,]$ is a simple $\lambda\mu^{\rightarrow+\times-}$-context such that α does not occur in $C[\,]$ then:*

- $\mathcal{S}_{[]}(t\{\alpha\leftarrow C[\,]\}) \subseteq \mathcal{S}_{[]}(t)$
  $\mathcal{S}_\alpha(t\{\alpha\leftarrow C[\,]\}) \subseteq \mathcal{S}_\alpha(t)\cup\mathcal{S}_{[]}(C[\,])$
  $\mathcal{S}_\delta(t\{\alpha\leftarrow C[\,]\}) \subseteq \mathcal{S}_\delta(t)\cup\mathcal{S}_\delta(C[\,])$ *for any* $\delta\neq\alpha$

Lemma 6.5. *Let u be a $\lambda\mu^{\rightarrow+\times-}$-term, let $C[\,]$ be an simple $\lambda\mu^{\rightarrow+\times-}$-context, and let δ be a free name in $C[u]$:*

- $\mathcal{S}_{[]}(C[u])=\mathcal{S}_{[]}(C[\,])\cup\mathcal{S}_{[]}(u)$
  $\mathcal{S}_\delta(C[u])=\mathcal{S}_\delta(C[\,])\cup\mathcal{S}_\delta(u)$

Lemma 6.6. *Given an instance $r\rightsquigarrow s$ of a rule of the $\lambda\mu^{\rightarrow+\times-}$-calculus, if r is safe then $\mathcal{S}_{[]}(s)\subseteq\mathcal{S}_{[]}(r)$ and $\mathcal{S}_\delta(s)\subseteq\mathcal{S}_\delta(r)$ for any a free name δ of s.*

Proof. Let us consider each reduction rule of the $\lambda\mu^{\rightarrow+\times-}$-calculus. Let y be a free variable of s (and thus a free variable of r) and let δ be a free name in s (and thus a free name in r):

Detour-reduction

a) $r=(\lambda x.u\,v)$ and $s=u\{v/x\}$. By lemma 6.2:

- $\mathcal{S}_{[]}(u\{v/x\})\subseteq\mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(v)/x]\subseteq\mathcal{S}_{[]}(u)\backslash\{x\}\cup\mathcal{S}_{[]}(v)=\mathcal{S}_{[]}(\lambda x.u\,v)$
- $\mathcal{S}_\delta(u\{v/x\})\subseteq\mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x]\cup\mathcal{S}_\delta(v)=\mathcal{S}_\delta(u)\cup\mathcal{S}_\delta(v)=\mathcal{S}_\delta(\lambda x.u\,v)$
  (since r is safe and thus $x\notin\mathcal{S}_\delta(u)$)

b) $r=\mathbf{fst}\ \langle t,u\rangle$ and $s=t$.

- $\mathcal{S}_{[]}(t)\subseteq\mathcal{S}_{[]}(t)\cup\mathcal{S}_{[]}(u)=\mathcal{S}_{[]}(\mathbf{fst}\ \langle t,u\rangle)$

- $\mathcal{S}_\delta(t) \subseteq \mathcal{S}_\delta(t) \cup \mathcal{S}_\delta(u) = \mathcal{S}_\delta(\textbf{fst}\ \langle t, u\rangle)$

c) Similar to b).

d) $r = \textbf{case}\ (\textbf{inl}\ t)\ \textbf{of}\ (\textbf{inl}\ x) \mapsto u \mid (\textbf{inr}\ y) \mapsto v$ and $s = u\{t/x\}$. By lemma 6.2:

- $\mathcal{S}_{[]}(u\{t/x\}) \subseteq \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(t)/x] \subseteq \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_{[]}(v)[\mathcal{S}_{[]}(t)/y] = \mathcal{S}_{[]}(r)$
- $\mathcal{S}_\delta(u\{t/x\}) \subseteq \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_\delta(t)$
  $\subseteq \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_\delta(v)[\mathcal{S}_{[]}(t)/y] \cup \mathcal{S}_\delta(t) = \mathcal{S}_\delta(r)$

e) Similar to d).

f) $r = u\{v/x\}$ and $s = \textbf{let}\ x = v\ \textbf{in}\ u$. By lemma 6.2:

- $\mathcal{S}_{[]}(u\{v/x\}) \subseteq \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(v)/x] = \mathcal{S}_{[]}(\textbf{let}\ x = v\ \textbf{in}\ u)$
- $\mathcal{S}_\delta(u\{v/x\}) \subseteq \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\delta(v) = \mathcal{S}_\delta(\textbf{let}\ x = v\ \textbf{in}\ u)$

g) $r = \textbf{resume}\ (\textbf{make-coroutine}\ v\ C_\alpha)\ \textbf{with}\ x \mapsto u$ and
$s = \textbf{set-context}\ \beta\ C[u\{v/x\}]$.

- $\mathcal{S}_{[]}(\textbf{set-context}\ \beta\ C[u\{v/x\}]) = \emptyset$
  $\subseteq \mathcal{S}_{[]}(\textbf{resume}\ (\textbf{make-coroutine}\ v\ C_\beta)\ \textbf{with}\ x \mapsto u)$
- $\mathcal{S}_\beta(\textbf{set-context}\ \beta\ C[u\{v/x\}]) = \mathcal{S}_\beta(C[u\{v/x\}]) \cup \mathcal{S}_{[]}(C[u\{v/x\}])$
  $= \mathcal{S}_\beta(C[\,]) \cup \mathcal{S}_\beta(u\{v/x\}) \cup \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(u\{v/x\})$
  $\subseteq \mathcal{S}_\beta(C[\,]) \cup \mathcal{S}_\beta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\beta(v) \cup \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(v)/x]$
  $\subseteq \mathcal{S}_\beta(C[\,]) \cup \mathcal{S}_\beta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\beta(v) \cup \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(v)$ since $\mathcal{S}_{[]}(u) \subseteq \{x\}$
  $= \mathcal{S}_\beta(u)[\mathcal{S}_{[]}(v)/x] \cup (\mathcal{S}_\beta(v) \cup \mathcal{S}_\beta(C[\,]) \cup \mathcal{S}_{[]}(v) \cup \mathcal{S}_{[]}(C[\,]))$
  $= \mathcal{S}_\beta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\beta(\textbf{make-coroutine}\ v\ C_\beta)$
  $= \mathcal{S}_\beta(u)[\mathcal{S}_{[]}(\textbf{make-coroutine}\ v\ C_\beta)/x] \cup \mathcal{S}_\beta(\textbf{make-coroutine}\ v\ C_\beta)$
  $= \mathcal{S}_\beta(\textbf{resume}\ (\textbf{make-coroutine}\ v\ C_\beta)\ \textbf{with}\ x \mapsto u)$
- $\mathcal{S}_\delta(\textbf{set-context}\ \beta\ C[u\{v/x\}]) = \mathcal{S}_\delta(\mathcal{C}[u\{v/x\}])$
  $= \mathcal{S}_\delta(C[\,]) \cup \mathcal{S}_\delta(u\{v/x\})$
  $\subseteq \mathcal{S}_\delta(C[\,]) \cup \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\delta(v)$
  $= \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x] \cup (\mathcal{S}_\delta(v) \cup \mathcal{S}_\delta(C[\,]))$
  $= \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(v)/x] \cup \mathcal{S}_\delta(\textbf{make-coroutine}\ v\ C_\beta)$
  $= \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(\textbf{make-coroutine}\ v\ C_\beta)/x] \cup \mathcal{S}_\delta(\textbf{make-coroutine}\ v\ C_\beta)$
  $= \mathcal{S}_\delta(\textbf{resume}\ (\textbf{make-coroutine}\ v\ C_\beta)\ \textbf{with}\ x \mapsto u)$

Structural reduction

a) $r = C[\mu\alpha.u]$ and $s = \mu\alpha.u\{\alpha \leftarrow C[\,]\}$. By lemma 6.4 and lemma 6.5:

- $\mathcal{S}_{[]}(\mu\alpha.u\{\alpha \leftarrow C[\,]\}) = \mathcal{S}_\alpha(u\{\alpha \leftarrow C[\,]\}) = \mathcal{S}_\alpha(u) \cup \mathcal{S}_{[]}(C[\,])$
  $= \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(\mu\alpha.u) = \mathcal{S}_{[]}(C[\mu\alpha.u])$
- $\mathcal{S}_\delta(\mu\alpha.u\{\alpha \leftarrow C[\,]\}) \subseteq \mathcal{S}_\delta(u\{\alpha \leftarrow C[\,]\}) \subseteq \mathcal{S}_\delta(u) \cup \mathcal{S}_\delta(C[\,])$
  $= \mathcal{S}_\delta(C[\mu\alpha.u])$

b) $r = C[\textbf{case}\ t\ \textbf{of}\ (\textbf{inl}\ x) \mapsto u \mid (\textbf{inr}\ y) \mapsto v]$ and
$s = \textbf{case}\ t\ \textbf{of}\ (\textbf{inl}\ x) \mapsto C[u] \mid (\textbf{inr}\ y) \mapsto C[v]$

- $\mathcal{S}_{[]}(\textbf{case}\ t\ \textbf{of}\ (\textbf{inl}\ x) \mapsto C[u] \mid (\textbf{inr}\ y) \mapsto C[v])$

$$
\begin{aligned}
&= \mathcal{S}_{[]}(C[u])[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_{[]}(C[v])[\mathcal{S}_{[]}(t)/y] \\
&= \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_{[]}(v)[\mathcal{S}_{[]}(t)/y] \\
&= \mathcal{S}_{[]}(C[\textbf{case } t \textbf{ of } (\textbf{inl } x) \mapsto u \mid (\textbf{inr } y) \mapsto v])
\end{aligned}
$$

- $\mathcal{S}_\delta(\textbf{case } t \textbf{ of } x \mapsto C[u] \mid y \mapsto C[v]) = \mathcal{S}_\delta(C[u])[\mathcal{S}_\delta(t)/x] \cup \mathcal{S}_\delta(C[v])[\mathcal{S}_\delta(t)/y]$
$= \mathcal{S}_\delta(C[\,]) \cup \mathcal{S}_\delta(u)[\mathcal{S}_\delta(t)/x] \cup \mathcal{S}_\delta(v)[\mathcal{S}_\delta(t)/y]$
$= \mathcal{S}_\delta(C[\textbf{case } t \textbf{ of } (\textbf{inl } x) \mapsto u \mid (\textbf{inr } y) \mapsto v])$

c) $r = C[\textbf{resume } t \textbf{ with } x \mapsto u]$ and $s = \textbf{resume } t \textbf{ with } x \mapsto u$

- $\mathcal{S}_{[]}(\textbf{resume } t \textbf{ with } x \mapsto u) = \mathcal{S}_{[]}(C[u])[\mathcal{S}_{[]}(t)/x]$
$= \mathcal{S}_{[]}(C[\,]) \cup \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(t)/x]$
$\subseteq \mathcal{S}_{[]}(u)[\mathcal{S}_{[]}(t)/x]$
$= \mathcal{S}_{[]}(C[\textbf{resume } t \textbf{ with } x \mapsto u])$

- $\mathcal{S}_\delta(\textbf{resume } t \textbf{ with } x \mapsto C[u]) = \mathcal{S}_\delta(C[u])[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_\delta(t)$
$= \mathcal{S}_\delta(C[\,]) \cup \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_\delta(t)$
$\subseteq \mathcal{S}_\delta(u)[\mathcal{S}_{[]}(t)/x] \cup \mathcal{S}_\delta(t)$
$= \mathcal{S}_\delta(C[\textbf{resume } t \textbf{ with } x \mapsto u])$

Simplification

a) $r = \mu\alpha[\alpha]u$ and $s = u$ where α does not occur free in u.

- $\mathcal{S}_{[]}(u) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_\alpha(u) = \mathcal{S}_\alpha([\alpha]u) = \mathcal{S}_{[]}(\mu\alpha[\alpha]u)$ (since $\mathcal{S}_\alpha(u) = \emptyset$)
- $\mathcal{S}_\delta(u) = \mathcal{S}_\delta([\alpha]u) = \mathcal{S}_\delta(\mu\alpha[\alpha]u)$

b) $r = [\beta]\mu\alpha.t$ and $s = t\{\beta/\alpha\}$. By lemma 6.3:

- $\mathcal{S}_{[]}(t\{\beta/\alpha\}) \subseteq \mathcal{S}_{[]}(t) = \emptyset = \mathcal{S}_\alpha([\beta]\mu\alpha.t)$ (since t has the form $[\delta]v$)
- $\mathcal{S}_\beta(t\{\beta/\alpha\}) \subseteq \mathcal{S}_\beta(t) \cup \mathcal{S}_\alpha(t) = \mathcal{S}_\beta(\mu\alpha.t) \cup \mathcal{S}_{[]}(\mu\alpha.t) = \mathcal{S}_\beta([\beta]\mu\alpha.t)$
- $\mathcal{S}_\delta(t\{\beta/\alpha\}) \subseteq \mathcal{S}_\delta(t) = \mathcal{S}_\delta(\mu\alpha.t) = \mathcal{S}_\delta([\beta]\mu\alpha.t)$ for any $\delta \neq \beta$

□

Lemma 6.7. *Given two $\lambda\mu^{\to+\times-}$-terms t, u and an arbitrary context $C[\,]$, if $\mathcal{S}_{[]}(u) \subseteq \mathcal{S}_{[]}(t)$ and $\mathcal{S}_\delta(u) \subseteq \mathcal{S}_\delta(t)$ then $\mathcal{S}_{[]}(C[u]) \subseteq \mathcal{S}_{[]}(C[t])$ and $\mathcal{S}_\delta(C[u]) \subseteq \mathcal{S}_\delta(C[t])$ for any a free name δ of $C[t]$.*

Proof. By induction on the context $C[\,]$. □

Proposition 6.8. *Given a $\lambda\mu^{\to+\times-}$-term t, if t is safe and $t \rightsquigarrow u$ then $\mathcal{S}_{[]}(u) \subseteq \mathcal{S}_{[]}(t)$ and $\mathcal{S}_\delta(u) \subseteq \mathcal{S}_\delta(t)$ for any a free name δ of t.*

Proof. By lemma 6.6 and lemma 6.7. □

Lemma 6.9. *Given two $\lambda\mu^{\to+\times-}$-terms u, v, if u and v are safe w.r.t. coroutine contexts then $u\{v/x\}$ is safe w.r.t. coroutine contexts.*

Proof. Let $\lambda y.t$ be a subterm of $u\{v/x\}$. Either $\lambda y.t$ is a subterm of v or y does not occur in v. Consequently, $y \notin \mathcal{S}_\delta(t)$ since v is safe in the former case and since u is safe in the latter case. □

Lemma 6.10. *Given an instance $r \rightsquigarrow s$ of a rule of the $\lambda\mu^{\to+\times-}$-calculus, if r is safe w.r.t. coroutine contexts then s is safe w.r.t. coroutine contexts.*

Proof. Again, we consider each rule of the $\lambda\mu^{\to+\times-}$-calculus:

Detour-reduction

a) $r = (\lambda x.u\, v)$ and $s = u\{v/x\}$. Apply lemma 6.9.

b) $r = \mathbf{fst}\ \langle t, u\rangle$ and $s = t$. Then $s = t$ is safe.

c) Similar to b).

d) $r = \mathbf{case}\ (\mathbf{inl}\, t)\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v$ and $s = u\{t/x\}$. Apply lemma 6.9.

e) Similar to d).

f) $r = u\{v/x\}$ and $s = \mathbf{let}\ x = v\ \mathbf{in}\ u$. Apply lemma 6.9.

g) $r = \mathbf{resume}\ (\mathbf{make\text{-}coroutine}\, v\, C_\alpha)\ \mathbf{with}\ x \mapsto u$ and $s = \mathbf{set\text{-}context}\ \beta\ C[u\{v/x\}]$. By lemma 6.9, $u\{v/x\}$ is safe w.r.t. coroutine contexts, and since $C[\,]$ is safe w.r.t. coroutine contexts, we know by lemma 6.7 that s is also safe w.r.t. coroutine contexts.

Structural reduction

a) $r = C[\mu\alpha.w]$ and $s = \mu\alpha.w\{\alpha \leftarrow C[\,]\}$.
Let $\lambda y.t$ be a subterm of $\mu\alpha.w\{\alpha \leftarrow C[\,]\}$, either $\lambda y.t$ is a subterm of $C[\,]$ or y does not occur in $C[\,]$. Consequently, $y \notin \mathcal{S}_\delta(t)$ since $C[\,]$ is safe in the former case and since w is safe in the latter case.

b) $r = C[\mathbf{case}\, w\, \mathbf{of}\, (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v]$ and
$s = \mathbf{case}\, w\, \mathbf{of}\, (\mathbf{inl}\, x) \mapsto C[u] \mid (\mathbf{inr}\, y) \mapsto C[v]$.
Let $\lambda y.t$ be a subterm of $\mathbf{case}\, w\, \mathbf{of}\, (\mathbf{inl}\, x) \mapsto C[u] \mid (\mathbf{inr}\, y) \mapsto C[v]$, either $\lambda y.t$ is a subterm of $C[\,]$ or y does not occur in $C[\,]$. Consequently, $y \notin \mathcal{S}_\delta(t)$ since $C[\,]$ is safe in the former case and since u, v and w are safe in the latter case.

c) $r = C[\mathbf{resume}\ c\ \mathbf{with}\ x \mapsto u]$ and $s = \mathbf{resume}\ c\ \mathbf{with}\ x \mapsto u$
Obviously s is safe w.r.t. coroutine contexts since it is a subterm of r.

Simplification

a) $r = \mu\alpha[\alpha]u$ and $s = u$ where α does not occur free in u. Then $s = u$ is safe.

b) $r = [\beta]\mu\alpha.u$ and $s = u\{\beta/\alpha\}$. Let $\lambda y.t$ be a subterm of $u\{\beta/\alpha\}$ and let $\lambda y.v$ be the subterm of u such as $\lambda y.v = \lambda y.t\{\beta/\alpha\}$. Then $y \notin \mathcal{S}_\delta(t) = \mathcal{S}_\delta(v)$ and $y \notin \mathcal{S}_\beta(t) = \mathcal{S}_\beta(v) \cup \mathcal{S}_\alpha(v)$ since u is safe.

□

Lemma 6.11. *Given two $\lambda\mu^{\to+\times-}$-terms t, u safe w.r.t. first-class coroutines and such that $\mathcal{S}_{[]}(u) \subseteq \mathcal{S}_{[]}(t)$ and $\mathcal{S}_\delta(u) \subseteq \mathcal{S}_\delta(t)$ and an arbitrary context $C[\,]$, if $C[t\,]$ is safe then $C[u]$ is safe w.r.t. first-class coroutines.*

Proof. By induction on the context $C[\,]$. □

Lemma 6.12. *Given an instance $r \rightsquigarrow s$ of a rule of the $\lambda\mu^{\to+\times-}$-calculus, if r is safe w.r.t. first-class coroutines then s is safe w.r.t. first-class coroutines.*

Proof. We already know by lemma 6.10 that s is safe w.r.t. coroutine contexts. We just have to check that for any subterm of s which has the form $\mathbf{resume}\, c\, \mathbf{with}\, x \mapsto u$, $\mathcal{S}_{[]}(u) \subseteq \{x\}$. This property follows from lemma

6.6 and lemma 6.7. □

Theorem 6.13. *Given a $\lambda\mu^{\rightarrow+\times-}$-term t, if t is safe with respect to first-class coroutines and $t \rightsquigarrow u$ then u is safe with respect to first-class coroutines.*

Proof. Let us write t as $C[r]$, where r is the redex to be reduced and let $r \rightsquigarrow s$ be the instance of the rule which is applied. By lemma 6.12, s is safe w.r.t. first-class coroutines, and by lemma 6.6, we have $\mathcal{S}_{[]}(s) \subseteq \mathcal{S}_{[]}(r)$ and $\mathcal{S}_\delta(s) \subseteq \mathcal{S}_\delta(r)$ for any a free name δ of s. Then, by lemma 6.11 u is safe w.r.t. first-class coroutines.

□

Corollary 6.14. *Given a $\lambda\mu^{\rightarrow+\times}$-term t, if t is safe with respect to coroutine contexts and $t \rightsquigarrow u$ then u is safe with respect to coroutine contexts.*

7 Conclusion and further work

We have defined the safe $\lambda\mu^{\rightarrow+\times}$-calculus, which is closed under reduction and whose type system corresponds to intuitionistic logic. In this calculus, continuations are not first-class objects but the ability of context-switching remains. The safe $\lambda\mu^{\rightarrow+\times-}$-calculus is an extension of the $\lambda\mu^{\rightarrow+\times}$-calculus with first-class coroutines. First-class coroutines are strictly less powerful than first-class continuations: the type system of the $\lambda\mu^{\rightarrow+\times-}$-calculus corresponds to subtractive logic, which is conservative over intuitionistic logic.

We have proved that first-class coroutines disappear during the normalization process (since the subformula property holds for normal forms) whenever the type of the term contains no subtraction (see proposition 4.9). The normal form belongs to the safe $\lambda\mu^{\rightarrow+\times}$-calculus. If we extend our work to the first-order framework, and since subtractive logic is conservative over CDL (and the existence property holds in CDL), we expect to be able to extract witnesses from normal proofs of existential formulas (which contain no subtraction). A first attempt could consist in exploiting the proof of conservativity given in appendix B. Another (better) solution would be to derive the existence property from the subformula property (however this is not straightforward in a deduction system with multi-conclusioned sequents). A forthcoming paper shall be devoted to this issue.

Applications of first-class coroutines were barely mentioned in this paper. In fact, practical applications of coroutines often use other extensions such as imperative features which do not easily fit in the formulae-as-types framework. However, purely functional examples have to be explored. On the other hand, in would be interesting to define an abstract machine for the safe $\lambda\mu^{\rightarrow+\times}$-calculus and to investigate what kind of optimization is allowed by the "safeness" property (as opposed to full-fledged continuations).

Eventually, the duality call-by-name/call-by-value (from the classical $\lambda\mu$-calculus) should be revisited in our framework, where the duality is likely to exchange functions and coroutines. We already know by construction of the safe $\lambda\mu^{\rightarrow+\times-}$-calculus that the dual of a safe $\lambda\mu^{\rightarrow+\times-}$-term is also safe.

Acknowledgment

I am very grateful to Serge Grigorieff for his careful reading and to Hugo Herbelin for helpful comments to earlier versions of this article. I also wish to thank one anonymous referee for his valuable report and his suggestions which helped to improve the manuscript in many ways.

Appendix A Properties of the $\lambda\mu^{\to+\times-}$-calculus

We shall take advantage of the definability of subtraction in classical logic to derive strong normalization and uniqueness of normal forms for the $\lambda\mu^{\to+\times-}$-calculus from the same properties of the $\lambda\mu^{\to+\times}$-calculus. In order to derive also the permutative reduction rules for the $\lambda\mu^{\to+\times-}$-calculus, we shall need a $\lambda\mu^{\to+\times}$-calculus with pattern matching: we thus add a new "tensor" product $\otimes$ to our type system. Then we shall be able to define $A - B$ as $A \otimes \neg B$ and then **make-coroutine**/**resume** as macro-definitions. Recall that these macros are in fact extracted from our derivations of the introduction/elimination rules of the (defined) subtraction.

In [de Groote, 2001], de Groote presents a proof of strong normalization of CND with primitive disjunction (and primitive conjunction with projections). We show here how to adapt this proof to take into account the rule for pattern matching. Although de Groote did not consider the simplification rules in his paper, it is easy to show that simplification (alone) is strongly normalizing and also that simplification may be postponed with respect with the other reduction relations.

De Groote's proof of strong normalization is two-fold: first he proves the strong normalization of the structural reductions (for untyped $\lambda\mu^{\to\wedge\vee\bot}$-terms) then he proves the strong normalization of typed $\lambda\mu^{\to\wedge\vee\bot}$-terms using a CPS-simulation. The latter proof is given in a propositional framework, but de Groote claims that since this CPS-translation is defined on the untyped $\lambda\mu$-terms, it may be raised to the second-order.

A.1 The $\lambda\mu^{\to+\times\otimes\bot}$-calculus

We add two term constructors to the syntax of the raw $\lambda\mu^{\to+\times}$-calculus:

$$M \ ::= \ \ldots \ | \quad (M, N) \ | \quad \textbf{match } M \textbf{ with } (x, y) \mapsto N$$

We also define the simple $\lambda\mu^{\to+\times\otimes\bot}$-contexts by extending the grammar of simple $\lambda\mu^{\to+\times}$-contexts:

$$C \ ::= \ \ldots \ | \quad \textbf{match } M \textbf{ with } (x, y) \mapsto N$$

Remark A.1. Recall that **abort** t is defined in the $\lambda\mu$-calculus as **set-context** $\varepsilon\ t$ where ε is a free name. Consequently, we shall not consider here **abort** as a primitive instruction.

A.1.1 Reduction rules

We extend the reduction rules of the $\lambda\mu^{\to+\times}$-calculus with this new detour-reduction rule:

f) **match** (u, v) **with** $(x, y) \mapsto t \rightsquigarrow t\{u/x, y/v\}]$

and the structural reduction is now defined by the following rules:

a) $C[\mu\alpha.u] \ \rightsquigarrow \ \mu\alpha.u\{\alpha \hookleftarrow C[\,]\}$

b) $C[\textbf{case } t \textbf{ of } (\textbf{inl}\, x) \mapsto u \mid (\textbf{inr}\, y) \mapsto v] \ \rightsquigarrow \ \textbf{case } t \textbf{ of } (\textbf{inl}\, x) \mapsto C[u] \mid (\textbf{inr}\, y) \mapsto C[v]$

c) $C[\textbf{match } t \textbf{ with } (x, y) \mapsto u] \ \rightsquigarrow \ \textbf{match } t \textbf{ with } (x, y) \mapsto C[u]$

where $C[\,]$ ranges over simple $\lambda\mu^{\to+\times\otimes\bot}$-contexts.

A.1.2 Typing rule for $\bot$

A.1.3 Typing rules for $\otimes$

$$\frac{t\colon \Gamma \vdash \Delta; A \qquad u\colon \Gamma \vdash \Delta; B}{(t, u)\colon \Gamma \vdash \Delta; A \otimes B}\,(\otimes_I) \qquad \frac{t\colon \Gamma \vdash \Delta; A \otimes B \qquad u\colon \Gamma, A^x, B^y \vdash \Delta; C}{\textbf{match } t \textbf{ with } (x, y) \mapsto u\colon \Gamma \vdash \Delta; C}\,(\otimes_E)$$

A.1.4 Strong normalization of the structural reductions

We provide $\lambda\mu^{\to+\times\otimes\bot}$-terms with a norm adapted from the norm introduced in [de Groote, 2001] (we recall the full definitions but the new cases are only the two last of each definition):

Definition A.2. *The norm $|\cdot|$ assigned to the $\lambda\mu^{\to+\times\otimes\bot}$-terms is inductively defined as follows:*

a) $|x|=1$

b) $|\lambda x.t|=|t|$

c) $|(t\,u)|=|t|+\#t\times|u|$

d) $|\langle t,u\rangle|=|t|+|u|$

e) $|\mathbf{fst}\,t|=|t|+\#t$

f) $|\mathbf{snd}\,t|=|t|+\#t$

g) $|\mathbf{inl}\,t|=|t|$

h) $|\mathbf{inr}\,t|=|t|$

i) $|\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\,x)\mapsto u\mid(\mathbf{inr}\,y)\mapsto v|=|t|+\#t\times(|t|+|u|)$

j) $|\mu\alpha.t|=|t|$

k) $|[\alpha]t|=|t|$

l) $|(t,u)|=|t|+|u|$

m) $|\mathbf{match}\ t\ \mathbf{with}\ (x,y)\mapsto u|=|t|+2\times\#t\times|u|$

where:

a) $\#x=1$

b) $\#\lambda x.t=1$

c) $\#(t\,u)=\#t$

d) $\#\langle t,\,u\rangle=1$

e) $\#\mathbf{fst}\,t=\#t$

f) $\#\mathbf{snd}\,t=\#t$

g) $\#\mathbf{inl}\,t=1$

h) $\#\mathbf{inr}\,t=1$

i) $\#\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\,x)\mapsto u\mid(\mathbf{inr}\,y)\mapsto v=(2\times\#t)\times(\#u+\#v)$

j) $\#\mu\alpha.t=\lfloor t\rfloor_\alpha$

k) $\#[\alpha]t=1$

l) $\#(t,\,u)=1$

m) $\#\mathbf{match}\ t\ \mathbf{with}\ (x,y)\mapsto u=(2\times\#t)\times(2\times\#u)$

and where:

a) $\lfloor x\rfloor_\alpha=0$

b) $\lfloor\lambda x.t\rfloor_\alpha=\lfloor t\rfloor_\alpha$

c) $\lfloor(t\,u)\rfloor_\alpha=\lfloor t\rfloor_\alpha+\#t\times\lfloor u\rfloor_\alpha$

d) $\lfloor\langle t,u\rangle\rfloor_\alpha=\lfloor t\rfloor_\alpha+\lfloor u\rfloor_\alpha$

e) $\lfloor\mathbf{fst}\,t\rfloor_\alpha=\lfloor t\rfloor_\alpha$

f) $\lfloor\mathbf{snd}\,t\rfloor_\alpha=\lfloor t\rfloor_\alpha$

$g)$ $\lfloor \mathbf{inl}\, t \rfloor_\alpha = \lfloor t \rfloor_\alpha$

$h)$ $\lfloor \mathbf{inr}\, t \rfloor_\alpha = \lfloor t \rfloor_\alpha$

$i)$ $\lfloor \mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v \rfloor_\alpha = \lfloor t \rfloor_\alpha + \#t \times (\lfloor u \rfloor_\alpha + \lfloor v \rfloor_\alpha)$

$j)$ $\lfloor \mu\alpha.t \rfloor_\alpha = \lfloor t \rfloor_\alpha$

$k)$ $\lfloor [\alpha]t \rfloor_\alpha = \lfloor t \rfloor_\alpha + \#t$

$l)$ $\lfloor [\beta]t \rfloor_\alpha = \lfloor t \rfloor_\alpha$

$m)$ $\lfloor (t, u) \rfloor_\alpha = \lfloor t \rfloor_\alpha + \lfloor u \rfloor_\alpha$

$n)$ $\lfloor \mathbf{match}\ t\ \mathbf{with}\ (x, y) \mapsto u \rfloor_\alpha = \lfloor t \rfloor_\alpha + 2 \times \#t \times \lfloor u \rfloor_\alpha$

Lemma A.3. *If t and u are two $\lambda\mu^{\to+\times\otimes\bot}$-terms and $t \rightsquigarrow_S u$ (where $\rightsquigarrow_S$ denotes the structural reduction) then $|t| > |u|$.*

A.1.5 CPS-simulation

We adapt here de Groote's modified CPS-translation, which simulates the relation of detour-reduction by strict β-reduction and the relation of structural reduction by equality.

Definition A.4. *The modified CPS-translation $\bar{\bar{t}}$ of any $\lambda\mu^{\to+\times\otimes\bot}$-term is defined as:*

$$\bar{\bar{t}} = \lambda k.(t : k)$$

where k is a fresh variable and where the infix operator ":" obeys the following definition:

$a)$ $x : k = (x\ k)$

$b)$ $\lambda x.t : k = (k\ \lambda x.\bar{\bar{t}}\ )$

$c)$ $(t\ u) : k = t : \lambda m.m\ \bar{\bar{u}}\ k$

$d)$ $\langle t, u \rangle : k = (k\ \lambda m.(m\ \bar{\bar{t}}\ \ \bar{\bar{u}}\ ))$

$e)$ $\mathbf{fst}\, t : k = t : \lambda m.(m\ \lambda i.\lambda j.(i\ k))$

$f)$ $\mathbf{snd}\, t : k = t : \lambda m.(m\ \lambda i.\lambda j.(j\ k))$

$g)$ $\mathbf{inl}\, t : k = (k\ \lambda i.\lambda j.(i\ \bar{\bar{t}}\ ))$

$h)$ $\mathbf{inr}\, t : k = (k\ \lambda i.\lambda j.(j\ \bar{\bar{t}}\ ))$

$i)$ $\mathbf{case}\ t\ \mathbf{of}\ (\mathbf{inl}\, x) \mapsto u \mid (\mathbf{inr}\, y) \mapsto v : k = t : \lambda m.(m\ \lambda x.(u\ k)\ \lambda y.(v\ k))$

$j)$ $\mu\alpha.t : k = (t : \lambda k.k)\{k/\alpha\}$ *if α occurs free in t*

$k)$ $\mu\alpha.t : k = \lambda\alpha.(t : \lambda k.k)\ k$ *otherwise*

$l)$ $[\alpha]t : k = t\ \alpha$

$m)$ $(t, u) : k = (k\ \lambda m.(m\ \bar{\bar{t}}\ \ \bar{\bar{u}}\ ))$

$n)$ $\mathbf{match}\ t\ \mathbf{with}\ (x, y) \mapsto u : k = t : \lambda m.(m\ \lambda x \lambda y.(u\ k)$

where m, i, j are fresh variables.

Lemma A.5. *Let t and u be two $\lambda\mu^{\to+\times\otimes\bot}$-terms. If $t \rightsquigarrow_D u$ (where $\rightsquigarrow_D$ denotes the structural reduction) then $\bar{\bar{t}} \rightsquigarrow_\beta \bar{\bar{u}}$.*

Lemma A.6. *Let t and u be two $\lambda\mu^{\to+\times\otimes\bot}$-terms. If $t \rightsquigarrow_S u$ then $\bar{\bar{t}} = \bar{\bar{u}}$.*

Theorem A.7. *The well-typed $\lambda\mu^{\to+\times\otimes\bot}$-terms are strongly normalizable.*

A.1.6 Church-Rosser property of the $\lambda\mu^{\to+\times\otimes\bot}$-calculus

The Church-Rosser property follows from the local confluence of the reductions, the strong normalization and Newman lemma.

A.2 Strong normalization and confluence of the $\lambda\mu^{\to+\times-}$-calculus

Theorem A.8. *The typed $\lambda\mu^{\to+\times-}$-calculus is strongly normalizing and enjoys the Church-Rosser property.*

Proof. Let us denote by Φ the translation from the $\lambda\mu^{\to+\times-}$-calculus into the $\lambda\mu^{\to+\times\otimes\bot}$-calculus defined by the following macro-definitions:

$$\begin{array}{rcl}\textbf{make-coroutine}\, t\, C_\alpha & \equiv & (t, \lambda z.\textbf{set-context}\ \alpha\ C[z]) \\ \textbf{resume}\ t\ \textbf{with}\ x \mapsto u & \equiv & \textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto\ \textbf{abort}\,(k\ u)\end{array}$$

It is enough to check the following properties: (1) Φ is a morphism for the reduction, (2) Φ preserves normal forms and (3) Φ is injective on normal forms. Properties (2) and (3) are easy to check. Let us verify that Φ is a morphism:

- $\textbf{resume}\ (\textbf{make-coroutine}\, t\, C_\alpha)\ \textbf{with}\ x \mapsto u$
  $\equiv \textbf{match}\ (t, C_\alpha)\ \textbf{with}\ (x,k) \mapsto \textbf{abort}\,(k\ u)$
  $\rightsquigarrow \textbf{abort}\,(C_\alpha\ u\{t/x\})$
  $\equiv \textbf{abort}\,(\lambda z.\textbf{set-context}\ \alpha\ C[z]\ u\{t/x\})$
  $\rightsquigarrow \textbf{abort}\,(\textbf{set-context}\ \alpha\ C[u\{t/x\}])$
  $\equiv \textbf{set-context}\ \varepsilon\ \textbf{set-context}\ \alpha\ C[u\{t/x\}]$
  $\rightsquigarrow \textbf{set-context}\ \alpha\ C[u\{t/x\}]$
- $C[\textbf{resume}\ t\ \textbf{with}\ x \mapsto u]$
  $\equiv C[\textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto \textbf{abort}\,(k\ u)]$
  $\rightsquigarrow \textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto C[\textbf{abort}\,(k\ u)]$
  $\equiv \textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto C[\textbf{set-context}\ \varepsilon\ (k\ u)]$
  $\rightsquigarrow \textbf{match}\ t\ \textbf{with}\ (x,k) \mapsto \textbf{set-context}\ \varepsilon\ (k\ u)$
  $\equiv \textbf{resume}\ t\ \textbf{with}\ x \mapsto u$

The remaining rules are straightforward to deal with. □

Appendix B $\mathrm{SND}_{\to\vee\wedge-}$ is sound and complete for SL

In this appendix, we show that $\mathrm{SND}_{\to\vee\wedge-}$ is conservative over $\mathrm{SND}^1_{\to\vee\wedge-}$ and thus it is sound and complete for Subtractive Logic (SL). We shall prove that any derivation of a sequent in $\mathrm{SND}_{\to\vee\wedge-}$ can be translated into a derivation which does not contain any (right-hand side) introduction rule of implication nor any left-hand side introduction rule of subtraction, but which depends only on axioms valid in $\mathrm{SND}^1_{\to\vee\wedge-}$.

The tricky part of the proof consists in showing that the constructive introduction rule of implication *commutes* with any other rule. Eventually, we conclude by applying the duality: the constructive left-hand side introduction rule for subtraction also *commutes* with any other rule (by duality). In order to prove this property, we have first to generalize these rules :

- We denote by $\mathrm{hyp}(\Delta)$ the set of (occurrences of) hypotheses linked to at least one conclusion of Δ and let us define the following generalization of the constructive introduction rule of implication:

$$\frac{\Gamma \vdash \Delta}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee} \quad \text{where } H \notin \mathrm{hyp}(\Delta\backslash S)$$

Note that H does not need to occur in Γ, and occurrences of hypotheses of S do not need to occur (and possibly none does) in Δ.

- Its dual rule, which generalizes the constructive left-hand side introduction rule for subtraction is the following (where cncl(Γ) denotes the set of occurrences of conclusions linked to at least one hypothesis of Γ)

$$\frac{\Gamma \vdash \Delta}{\Gamma\backslash S, S^{\wedge} - C \vdash \Delta\backslash\{C\}} \quad \text{where } C \notin \text{cncl}(\Gamma\backslash S)$$

Again, C does not need to occur in Γ, and occurrences of hypotheses of S do not need to occur (and possibly none does) in Γ.

Theorem B.1. *The system* $\text{SND}_{\to\vee\wedge-}$ *is conservative over* $\text{SND}^{1}_{\to\vee\wedge-}$.

Proof. We first deal with the case of axioms (in section B.1). Then, the main part of the proof consists in showing that the generalized introduction rule of implication *commutes* with any other rule (in section B.2). Eventually, we conclude by applying the duality: the generalized left-hand side introduction rule for subtraction also *commutes* with any other rule (by duality). Note that this procedure terminates since the generalized rules are always applied to smaller proofs after a replacement. □

B.1 Axioms

Proposition B.2. *In* $\text{SND}_{\to\vee\wedge}$*, the set of annotated sequents that belong to one of the three following collections:*

- $\Gamma, A \vdash \Delta, B$ *where* $A \vdash B$ *is valid in* SL*, and there is at most one link which annotates this sequent, and this link binds* A *and* B *together;*
- $\Gamma, Y \vdash \Delta$ *where* $Y \vdash \bot$ *is valid in* SL*;*
- $\Gamma \vdash \Delta, X$ *where* $\top \vdash X$ *is valid in* SL*.*

is closed under the generalized (right-hand side) introduction rule of implication (resp. left-hand side introduction rule for subtraction).

Proof. Let us consider the generalized introduction rule of implication for the three collections of sequents:

1. The upper sequent has the form $\Gamma, A \vdash \Delta, B$ where $A \vdash B$ is valid in SL, and the unique link which annotates this sequent binds A and B together.
   - First case, $H \neq A$ and $B \notin S$.
   $$\frac{\Gamma, A \vdash \Delta, B}{\Gamma\backslash\{H\}, A \vdash \Delta\backslash S, B, H \to S^{\vee}}$$
   the lower sequent is indeed of the first form.
   - Second case, $H \neq A$ and H is discharged onto $S \cup \{B\}$
   $$\frac{\Gamma, A \vdash \Delta, B}{\Gamma\backslash\{H\}, A \vdash \Delta\backslash S, H \to (S^{\vee} \vee B)}$$
   the lower sequent is indeed of the third form since if $A \vdash B$ is valid in SL and $B \in S$ then $A \vdash H \to (S^{\vee} \vee B)$ is also valid.
   - Third case, $H = A$ and A is discharged onto $S \cup \{B\}$
   $$\frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta\backslash S, A \to (S^{\vee} \vee B)}$$

the lower sequent is indeed of the third form since if $A \vdash B$ is valid in SL then $\top \vdash A \to (S^\vee \vee B)$ is also valid.

In the case $H = A$ and $B \notin S$, *i.e.* where A is discharged onto another conclusion that B, the constructive constraint does not hold. Consequently this case has not to be considered.

2. The upper sequent has the form $\Gamma, Y \vdash \Delta$ where $Y \vdash \bot$ is valid in SL.
   - First case, $H \neq Y$ and $H \notin \mathrm{hyp}(\Delta \backslash S)$
   $$\frac{\Gamma, Y \vdash \Delta}{\Gamma \backslash \{H\}, Y \vdash \Delta \backslash S, H \to S^\vee}$$
   the lower sequent is still of the second form.
   - Second case, $H = Y$ and $Y \notin \mathrm{hyp}(\Delta \backslash S)$
   $$\frac{\Gamma, Y \vdash \Delta}{\Gamma \vdash \Delta \backslash S, Y \to S^\vee}$$
   Since $Y \vdash \bot$ is valid in SL, we infer that $Y \vdash S^\vee$ and thus $\top \vdash Y \to S^\vee$ is also valid in SL, and the lower sequent is thus of the third form.
3. The upper sequent has the form $\Gamma \vdash \Delta, X$ where $\top \vdash X$ is valid in SL.
   - First case, $X \notin S$
   $$\frac{\Gamma \vdash \Delta, X}{\Gamma \backslash \{H\} \vdash \Delta \backslash S, X, H \to S^\vee}$$
   the lower sequent is still of the third form.
   - Second case, H is discharged onto $S \cup \{X\}$
   $$\frac{\Gamma \vdash \Delta, X}{\Gamma \backslash \{H\} \vdash \Delta \backslash S, H \to (S^\vee \vee X)}$$
   Since $\top \vdash X$ is derivable in SL, we infer that $H \vdash S^\vee \vee X$ and thus $\top \vdash H \to (S^\vee \vee X)$ are also valid in SL, and the lower sequent is thus still of the third form.

The closure under the left-hand side introduction rule for subtraction is obtained by duality. □

B.2 Rules

Left-hand side weakening rule

- First case, $H \neq A$:
$$\frac{\dfrac{\Gamma \vdash^1 \Delta}{\Gamma, A \vdash^2 \Delta}}{\Gamma \backslash \{H\}, A \vdash^3 \Delta \backslash S, A \to S^\vee}$$
where $H \notin \mathrm{hyp}_2(\Delta \backslash S)$ and thus $H \notin \mathrm{hyp}_1(\Delta \backslash S)$. Replace with:
$$\frac{\dfrac{\Gamma \vdash \Delta}{\Gamma \backslash \{H\} \vdash \Delta \backslash S, A \to S^\vee}}{\Gamma \backslash \{H\}, A \vdash \Delta \backslash S, A \to S^\vee}$$
- Second case, $H = A$:
$$\frac{\dfrac{\Gamma \vdash^1 \Delta}{\Gamma, A \vdash^2 \Delta}}{\Gamma \vdash^3 \Delta \backslash S, A \to S^\vee}$$
where $A \notin \mathrm{hyp}_2(\Delta \backslash S)$ and thus $A \notin \mathrm{hyp}_1(\Delta \backslash S)$. Replace with:
$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta \backslash S, A \to S^\vee}$$

Left-hand side contraction rule

- First case, $H = A^z$:

$$\dfrac{\dfrac{\Gamma, A^x, A^y \vdash^1 \Delta}{\Gamma, A^z \vdash^2 \Delta}}{\Gamma \vdash^3 \Delta\backslash S, A \to S^\vee}$$

  where $A^z \notin \mathrm{hyp}_2(\Delta\backslash S)$ and thus $A^x \notin \mathrm{hyp}_1(\Delta\backslash S)$ and $A^y \notin \mathrm{hyp}_1(\Delta\backslash\{B^S\})$. Replace with:

$$\dfrac{\dfrac{\Gamma, A^x, A^y \vdash \Delta}{\Gamma, A^z \vdash \Delta\backslash S, A \to S^\vee}}{\Gamma \vdash \Delta\backslash S, A \to (A \to S^\vee)}$$

  and then cut with the sequent $A \to (A \to S^\vee) \vdash A \to S^\vee$ valid in SL.

- Second case, $H \neq A^z$:

$$\dfrac{\dfrac{\Gamma, A^x, A^y \vdash^1 \Delta}{\Gamma, A^z \vdash^2 \Delta}}{\Gamma\backslash\{H\}, A^z \vdash^3 \Delta\backslash S, H \to S^\vee}$$

  where $H \notin \mathrm{hyp}_2(\Delta\backslash S)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S)$. Replace with:

$$\dfrac{\dfrac{\Gamma, A^x, A^y, H \vdash \Delta}{\Gamma\backslash\{H\}, A^x, A^y \vdash \Delta\backslash S, H \to S^\vee}}{\Gamma\backslash\{H\}, A^z \vdash \Delta\backslash S, H \to S^\vee}$$

Right-hand side contraction rule

- First case, $B \notin S$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta, B^\alpha, B^\beta}{\Gamma \vdash^2 \Delta, B^\gamma}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, H \to S^\vee, B^\gamma}$$

  where $H \notin \mathrm{hyp}_2(\Delta\backslash S, B^\gamma)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S, B^\alpha, B^\beta)$. Replace with:

$$\dfrac{\dfrac{\Gamma \vdash \Delta, B, B}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee, B, B}}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee, B}$$

- Second case, H is discharged onto $S \cup \{B\}$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta, B^\alpha, B^\beta}{\Gamma \vdash^2 \Delta, B^\gamma}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, H \to (S^\vee \vee B)}$$

  where $H \notin \mathrm{hyp}_2(\Delta\backslash S, B^\gamma)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S, B^\alpha, B^\beta)$. Replace with:

$$\dfrac{\Gamma \vdash \Delta, B, B}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to (S^\vee \vee B \vee B)}$$

  and then cut with the sequent $H \to (S^\vee \vee B \vee B) \vdash H \to (S^\vee \vee B)$ valid in SL.

Right-hand side weakening rule

- First case, $B \notin S$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta}{\Gamma \vdash^2 \Delta, B}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, H \to S^\vee, B}$$

  where $H \notin \mathrm{hyp}_2(\Delta\backslash S, B)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S)$. Replace with:

$$\dfrac{\dfrac{\Gamma \vdash \Delta}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee}}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee, B}$$

- Second case, H is discharged onto $S \cup \{B\}$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta}{\Gamma \vdash^2 \Delta, B^\alpha}}{\Gamma \backslash \{H\} \vdash^3 \Delta \backslash S, H \to (S^\vee \vee B)}$$

  where $H \notin \mathrm{hyp}_2(\Delta \backslash S, B^\alpha)$ and thus $H \notin \mathrm{hyp}_1(\Delta \backslash S)$. Replace with:

$$\dfrac{\Gamma \vdash \Delta}{\Gamma \backslash \{H\} \vdash \Delta \backslash S, H \to S^\vee}$$

  and then cut with the sequent $H \to S^\vee \vdash H \to (S^\vee \vee B)$ valid in SL.

Cut rule

$$\dfrac{\dfrac{\Gamma' \vdash^1 A, \Delta' \qquad \Gamma'', A \vdash^4 \Delta''}{\Gamma', \Gamma'' \vdash^2 \Delta', \Delta''}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash^3 (\Delta', \Delta'') \backslash S, H \to S^\vee}$$

where $H \notin \mathrm{hyp}_2((\Delta', \Delta'') \backslash S)$ and thus, by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$:

- First case, $H \notin \Gamma'$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A}{\Gamma' \vdash \Delta' \backslash S', H \to S'^\vee, A} \qquad \dfrac{\Gamma'', A \vdash \Delta''}{\Gamma'' \backslash \{H\}, A \vdash \Delta'' \backslash S', H \to S''^\vee}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, H \to S'^\vee, H \to S''^\vee}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, (H \to S'^\vee) \vee (H \to S''^\vee)}$$

  then cut with the sequent $(H \to S'^\vee) \vee (H \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

- Second case, $H \notin \Gamma''$ and $H \notin \mathrm{hyp}_1(A)$ since $H \notin \mathrm{hyp}_1(A, \Delta' \backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A}{\Gamma' \backslash \{H\} \vdash \Delta' \backslash S', H \to S'^\vee, A} \qquad \dfrac{\Gamma'', A \vdash \Delta''}{\Gamma'', A \vdash \Delta'' \backslash S', H \to S''^\vee}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, H \to S'^\vee, H \to S''^\vee}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, (H \to S'^\vee) \vee (H \to S''^\vee)}$$

  then cut with the sequent $(H \to S'^\vee) \vee (H \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

- Third case, $H \in \Gamma'$ and $H \in \mathrm{hyp}_1(A)$ and since $H \notin \mathrm{hyp}_1(\Delta'' \backslash S'')$ we have $A \notin \mathrm{hyp}_1(\Delta'' \backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A}{\Gamma' \backslash \{H\} \vdash \Delta' \backslash S', H \to (S'^\vee \vee A)} \qquad \dfrac{\Gamma'', A \vdash \Delta''}{\Gamma'' \vdash \Delta' \backslash S', A \to S''^\vee}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, (H \to (S'^\vee \vee A)) \wedge (A \to S''^\vee)}$$

  then cut with the sequent $(H \to (S'^\vee \vee A)) \wedge (A \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

Right-hand side elimination rule for the implication

- First case, $B \notin S$:

$$\dfrac{\dfrac{\Gamma' \vdash^1 \Delta', A \to B \qquad \Gamma'' \vdash^4 \Delta'', A}{\Gamma', \Gamma'' \vdash^2 \Delta', \Delta'', B}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash^3 (\Delta', \Delta'') \backslash S, B, H \to S^\vee}$$

  where $H \notin \mathrm{hyp}_2(B, (\Delta', \Delta'') \backslash S)$ and thus, by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$, we have $H \notin \mathrm{hyp}_1(A \to B, \Delta \backslash S')$ and $H \notin \mathrm{hyp}_4(A, \Delta \backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A \to B}{\Gamma' \backslash \{H\} \vdash \Delta' \backslash S', H \to S'^\vee, A \to B} \qquad \dfrac{\Gamma'' \vdash \Delta'', A}{\Gamma'' \backslash \{H\} \vdash \Delta'' \backslash S'', H \to S''^\vee, A}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, H \to S'^\vee, H \to S''^\vee, B}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, (H \to S'^\vee) \vee (H \to S''^\vee), B}$$

  and then cut with the sequent $(H \to S'^\vee) \vee (H \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

- Second case, H is discharged onto $S \cup \{B\}$:

$$\dfrac{\dfrac{\Gamma' \vdash^1 \Delta', A \to B \qquad \Gamma'' \vdash^4 \Delta'', A}{\Gamma', \Gamma'' \vdash^2 \Delta', \Delta'', B}}{(\Gamma, \Gamma')\backslash\{H\} \vdash^3 (\Delta', \Delta'')\backslash S, H \to (S^\vee \vee B)}$$

where $H \notin \mathrm{hyp}_2((\Delta', \Delta'')\backslash S)$ and thus, by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$, there is $H \notin \mathrm{hyp}_1(\Delta\backslash S')$ and $H \notin \mathrm{hyp}_4(\Delta\backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A \to B}{\Gamma'\backslash\{H\} \vdash \Delta'\backslash S', H \to (S'^\vee \vee (A \to B))} \qquad \dfrac{\Gamma'' \vdash \Delta'', A}{\Delta''\backslash S', H \to (S''^\vee \vee A)}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash (\Delta', \Delta'')\backslash S, (H \to (S'^\vee \vee (A \to B))) \wedge (H \to (S''^\vee \vee A))}$$

and then cut with the sequent $(H \to (S'^\vee \vee (A \to B))) \wedge (H \to (S''^\vee \vee A)) \vdash H \to (S^\vee \vee B)$ valid in SL.

Left-hand side introduction rule of disjunction

- First case, $H \neq A \vee B$:

$$\dfrac{\dfrac{\Gamma', A \vdash^1 \Delta' \qquad \Gamma'', B \vdash^4 \Delta''}{\Gamma', \Gamma'', A \vee B \vdash^2 \Delta', \Delta''}}{(\Gamma, \Gamma')\backslash\{H\}, A \vee B \vdash^3 (\Delta', \Delta'')\backslash S, H \to S^\vee}$$

where $H \notin \mathrm{hyp}_2((\Delta', \Delta'')\backslash S)$ and thus, by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$, we have $H \notin \mathrm{hyp}_1(\Delta\backslash S')$ and $H \notin \mathrm{hyp}_4(\Delta\backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma', A \vdash \Delta'}{\Gamma'\backslash\{H\}, A \vdash \Delta'\backslash S', H \to S'^\vee} \qquad \dfrac{\Gamma'', B \vdash \Delta''}{\Gamma''\backslash\{H\}, B \vdash \Delta'', H \to S''^\vee}}{(\Gamma', \Gamma'')\backslash\{H\}, A \vee B \vdash (\Delta', \Delta'')\backslash S, H \to S'^\vee, H \to S''^\vee}}{(\Gamma', \Gamma'')\backslash\{H\}, A \vee B \vdash (\Delta', \Delta'')\backslash S, (H \to S'^\vee) \vee (H \to S''^\vee)}$$

and then cut with the sequent $(H \to S'^\vee) \vee (H \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

- Second case, $H = A \vee B$:

$$\dfrac{\dfrac{\Gamma', A \vdash^1 \Delta' \qquad \Gamma'', B \vdash^4 \Delta''}{\Gamma', \Gamma'', A \vee B \vdash^2 \Delta', \Delta''}}{\Gamma', \Gamma'' \vdash^3 (\Delta', \Delta'')\backslash S, A \vee B \to S^\vee}$$

where $A \vee B \notin \mathrm{hyp}_2((\Delta', \Delta'')\backslash S)$ and thus, by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$, we have $A \notin \mathrm{hyp}_1(\Delta\backslash S')$ and $B \notin \mathrm{hyp}_4(\Delta\backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma', A \vdash \Delta'}{\Gamma'\backslash\{H\} \vdash \Delta'\backslash S', A \to S'^\vee} \qquad \dfrac{\Gamma'', B \vdash \Delta''}{\Delta''\backslash S', B \to S''^\vee}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash (\Delta', \Delta'')\backslash S, (A \to S'^\vee) \wedge (B \to S''^\vee)}}$$

and then cut with the sequent $(A \to S'^\vee) \wedge (B \to S''^\vee) \vdash (A \vee B) \to (S'^\vee \vee S''^\vee)$ valid in SL.

Left-hand side elimination rule of disjunction

We consider only the case of the first *injection*, the second one is similar.

- First case, $H \neq A$:

$$\dfrac{\dfrac{\Gamma, A \vee B \vdash^1 \Delta}{\Gamma, A \vdash^2 \Delta}}{\Gamma\backslash\{H\}, A \vdash^3 \Delta\backslash S, H \to S^\vee}$$

where $H \notin \mathrm{hyp}_2(\Delta\backslash S)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S)$. Replace with:

$$\dfrac{\dfrac{\Gamma, A \vee B \vdash \Delta}{\Gamma\backslash\{H\}, A \vee B \vdash \Delta\backslash S, H \to S^\vee}}{\Gamma\backslash\{H\}, A \vdash \Delta\backslash S, H \to S^\vee}$$

- Second case, $H = A$:

$$\dfrac{\dfrac{\Gamma, A \vee B \vdash^1 \Delta}{\Gamma, A \vdash^2 \Delta}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, A \to S^\vee}$$

where $H \notin \mathrm{hyp}_2(\Delta\backslash S)$ and thus $H \notin \mathrm{hyp}_1(\Delta\backslash S)$ and $A \notin \mathrm{hyp}_4(\Delta\backslash S)$. Replace with:

$$\dfrac{\Gamma, A \vee B \vdash \Delta}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, (A \vee B) \to S^\vee}$$

and then cut with the sequent $(A \vee B) \to S^\vee \vdash A \to S^\vee$ valid in SL.

Introduction rule for conjunction

- First case, $A \wedge B \notin S$:

$$\dfrac{\dfrac{\Gamma' \vdash^1 \Delta', A \qquad \Gamma'' \vdash^4 \Delta'', B}{\Gamma', \Gamma'' \vdash^2 \Delta', \Delta'', A \wedge B}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash^3 (\Delta', \Delta'')\backslash S, A \wedge B, H \to S^\vee}$$

where $H \notin \mathrm{hyp}_2(A, B, (\Delta', \Delta'')\backslash S)$ and thus by setting $S' = S \cap \Gamma'$ et $S'' = S \cap \Gamma''$, we have $H \notin \mathrm{hyp}_1(A, \Delta\backslash S')$ and $H \notin \mathrm{hyp}_4(B, \Delta\backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A}{\Gamma'\backslash\{H\} \vdash \Delta'\backslash S', H \to S'^\vee, A} \qquad \dfrac{\Gamma'' \vdash \Delta'', B}{\Gamma''\backslash\{H\} \vdash \Delta''\backslash S'', H \to S''^\vee, B}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash (\Delta', \Delta'')\backslash S, H \to S'^\vee, H \to S''^\vee, A \wedge B}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash (\Delta', \Delta'')\backslash S, (H \to S'^\vee) \vee (H \to S''^\vee), A \wedge B}$$

then cut with the sequent $(H \to S'^\vee) \vee (H \to S''^\vee) \vdash H \to (S'^\vee \vee S''^\vee)$ valid in SL.

- Second case, H is discharged onto $S \cup \{A \wedge B\}$:

$$\dfrac{\dfrac{\Gamma' \vdash^1 \Delta', A \qquad \Gamma'' \vdash^4 \Delta'', B}{\Gamma', \Gamma'' \vdash^2 \Delta', \Delta'', A \wedge B}}{(\Gamma', \Gamma')\backslash\{H\} \vdash^3 (\Delta', \Delta'')\backslash S, H \to (S^\vee \vee (A \wedge B))}$$

where $H \notin \mathrm{hyp}_2((\Delta', \Delta'')\backslash S)$ and thus by setting $S' = S \cap \Gamma'$ et $S'' = S \cap \Gamma''$, on a $H \notin \mathrm{hyp}_1(\Delta\backslash S')$ and $H \notin \mathrm{hyp}_4(\Delta\backslash S')$. Replace with:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \Delta', A}{\Gamma'\backslash\{H\} \vdash \Delta'\backslash S', H \to (S'^\vee \vee A)} \qquad \dfrac{\Gamma'' \vdash \Delta'', B}{\Delta''\backslash S', H \to (S''^\vee \vee B)}}{(\Gamma', \Gamma'')\backslash\{H\} \vdash (\Delta', \Delta'')\backslash S, (H \to (S'^\vee \vee A)) \wedge (H \to (S''^\vee \vee B))}}{}$$

then cut with the sequent $(H \to (S'^\vee \vee A)) \wedge (H \to (S''^\vee \vee B)) \vdash H \to (S^\vee \vee (A \wedge B))$ valid in SL.

Elimination rule for conjunction

We deal only with the first projection, the second one is similar.

- First case, $A \notin S$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta, A \wedge B}{\Gamma \vdash^2 \Delta, A}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, H \to S^\vee, A}$$

where $H \notin \mathrm{hyp}_2(A, \Delta\backslash S)$ and thus $H \notin \mathrm{hyp}_1(A \wedge B, \Delta\backslash S)$. Replace with:

$$\dfrac{\dfrac{\Gamma \vdash \Delta, A \wedge B}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee, A \wedge B}}{\Gamma\backslash\{H\} \vdash \Delta\backslash S, H \to S^\vee, A}$$

- Second case, H is discharged onto $S \cup \{A\}$:

$$\dfrac{\dfrac{\Gamma \vdash^1 \Delta, A \wedge B}{\Gamma \vdash^2 \Delta, A}}{\Gamma\backslash\{H\} \vdash^3 \Delta\backslash S, H \to (S^\vee \vee A)}$$

where $H \notin \mathrm{hyp}_2(\Delta \backslash S)$ and thus $H \notin \mathrm{hyp}_1(\Delta \backslash S)$ and $A \notin \mathrm{hyp}_4(\Delta \backslash S)$. Replace with:

$$\frac{\Gamma \vdash \Delta, A \wedge B}{\Gamma \backslash \{H\} \vdash \Delta \backslash S, H \to (S^{\vee} \vee (A \wedge B))}$$

then cut with the sequent $H \to (S^{\vee} \vee (A \wedge B)) \vdash H \to (S^{\vee} \vee A)$ valid in SL.

Left-hand side elimination rule for subtraction

- First case, $H \neq A$:

$$\frac{\dfrac{\Gamma', A - B \vdash^1 \Delta' \qquad \Gamma'', B \vdash^4 \Delta''}{\Gamma', \Gamma'', A \vdash^2 \Delta', \Delta''}}{(\Gamma', \Gamma'') \backslash \{H\}, A \vdash^3 (\Delta', \Delta'') \backslash S, H \to S^{\vee}}$$

where $H \notin \mathrm{hyp}_2((\Delta', \Delta'') \backslash S)$ and thus by setting $S' = S \cap \Gamma'$ et $S'' = S \cap \Gamma''$, we have $H \notin \mathrm{hyp}_1(\Delta \backslash S')$ and $H \notin \mathrm{hyp}_4(\Delta \backslash S')$. Replace with:

$$\frac{\dfrac{\dfrac{\Gamma', A - B \vdash \Delta'}{\Gamma' \backslash \{H\}, A - B \vdash \Delta' \backslash S', H \to S'^{\vee}} \qquad \dfrac{\Gamma'', B \vdash \Delta''}{\Gamma'' \backslash \{H\}, B \vdash \Delta'', H \to S''^{\vee}}}{(\Gamma', \Gamma'') \backslash \{H\}, A \vdash (\Delta', \Delta'') \backslash S, H \to S'^{\vee}, H \to S''^{\vee}}}{(\Gamma', \Gamma'') \backslash \{H\}, A \vdash (\Delta', \Delta'') \backslash S, (H \to S'^{\vee}) \vee (H \to S''^{\vee})}$$

then cut with the sequent $(H \to S'^{\vee}) \vee (H \to S''^{\vee}) \vdash H \to (S'^{\vee} \vee S''^{\vee})$ valid in SL.

- Second case, $H = A$:

$$\frac{\dfrac{\Gamma', A - B \vdash^1 \Delta' \qquad \Gamma'', B \vdash^4 \Delta''}{\Gamma', \Gamma'', A \vdash^2 \Delta', \Delta''}}{\Gamma', \Gamma'' \vdash^3 (\Delta', \Delta'') \backslash S, A \to S^{\vee}}$$

where $A \notin \mathrm{hyp}_2((\Delta', \Delta'') \backslash S)$ and thus by setting $S' = S \cap \Gamma'$ and $S'' = S \cap \Gamma''$, we have $A - B \notin \mathrm{hyp}_1(\Delta \backslash S')$ and $B \notin \mathrm{hyp}_4(\Delta \backslash S')$. Replace with:

$$\frac{\dfrac{\dfrac{\Gamma', A - B \vdash \Delta'}{\Gamma' \backslash \{H\} \vdash \Delta' \backslash S', (A - B) \to S'^{\vee}} \qquad \dfrac{\Gamma'', B \vdash \Delta''}{\Delta'' \backslash S', B \to S''^{\vee}}}{(\Gamma', \Gamma'') \backslash \{H\} \vdash (\Delta', \Delta'') \backslash S, ((A - B) \to S'^{\vee}) \wedge (B \to S''^{\vee})}$$

then cut with the sequent $((A - B) \to S'^{\vee}) \wedge (B \to S''^{\vee}) \vdash (A \to (S'^{\vee} \vee S''^{\vee})$ valid in SL.

Left-hand side introduction rule for subtraction

- First case, $H \neq A - B$:

$$\frac{\dfrac{\Gamma, A \vdash^1 \Delta, B}{\Gamma, A - B \vdash^2 \Delta}}{\Gamma \backslash \{H\}, A - B \vdash^3 \Delta \backslash S, H \to S^{\vee}}$$

where $H \notin \mathrm{hyp}_2(\Delta \backslash S)$ and $H \notin \mathrm{hyp}_1(B)$ and thus $H \notin \mathrm{hyp}_1(\Delta \backslash S, B)$. Replace with:

$$\frac{\dfrac{\Gamma, A \vdash \Delta, B}{\Gamma \backslash \{H\}, A \vdash \Delta \backslash S, H \to S^{\vee}, B}}{\Gamma \backslash \{H\}, A - B \vdash \Delta \backslash S, H \to S^{\vee}}$$

- Second case, $H = A - B$:

$$\frac{\dfrac{\Gamma, A \vdash^1 \Delta, B}{\Gamma, A - B \vdash^2 \Delta}}{\Gamma \vdash^3 \Delta \backslash S, (A - B) \to S^{\vee}}$$

where $H \notin \mathrm{hyp}_2(\Delta \backslash S)$ and $H \notin \mathrm{hyp}_1(B)$ and thus $H \notin \mathrm{hyp}_1(\Delta \backslash S, B)$. Replace with:

$$\frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta \backslash S, A \to (S^{\vee} \vee B)}$$

then cut with the sequent $A \to (S^{\vee} \vee B) \vdash (A - B) \to S^{\vee}$ valid in SL.

Bibliography

[Barbanera and Berardi, 1994] Barbanera, F. and Berardi, S. (1994). Extracting constructive content from classical logic via control-like reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag.

[Beth, 1956] Beth, E. W. (1956). Semantic construction of intuitionistic logic. *Koninklijke Nederlandse Akademie van Wetenschappen, Mededelingen, Nieuwe Reeks*, 19(11):357–388.

[Brauner and de Paiva, 1997] Brauner, T. and de Paiva, V. (1997). A formulation of linear logic based on dependency-relations. In Springer-Verlag, editor, *Proceedings of Annual Conference of the European Association for Computer Science Logic*, volume 1414 of *LNCS*.

[Cooper and Morrisett, 1990] Cooper, E. C. and Morrisett, J. G. (1990). Adding threads to standard ML. Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Crolard, 1996] Crolard, T. (1996). Extension de l'isomorphisme de Curry-Howard au traitement des exceptions (application d'une étude de la dualité en logique intuitionniste). Thèse de Doctorat. Université Paris 7.

[Crolard, 1999] Crolard, T. (1999). A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming*, 9(6):625–647.

[Crolard, 2001] Crolard, T. (2001). Subtractive Logic. *Theoretical Computer Science*, 254(1–2):151–185.

[Curien and Herbelin, 2000] Curien, P.-L. and Herbelin, H. (2000). The duality of computation. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243, N.Y. ACM Press.

[de Groote, 1994] de Groote, P. (1994). On the relation between the lambda-calculus and the syntactic theory of sequential control. *Lecture Notes in Computer Science*, 822:31–43.

[de Groote, 1995] de Groote, P. (1995). A simple calculus of exception handling. In *Second International Conference on Typed Lambda Calculi and Applications*, LNCS, pages 201–215, Edinburgh, United Kingdom.

[de Groote, 2001] de Groote, P. (2001). Strong normalization of classical natural deduction with disjunction. In Calculi, T. L. and Applications, editors, *LNCS*, pages LNCS 2044, p. 182 ff.1–13. Springer-Verlag.

[Dragalin, 1988] Dragalin, A. G. (1988). Mathematical intuitionism: introduction to proof theory. In *Translations of Mathematical Monographs*, volume 67. American Matematical Society, Providence, Rhode Island.

[Duba et al., 1991] Duba, B. F., Harper, R., and MacQueen, D. (1991). Typing first-class continuations in ML. In ACM-SIGACT, A.-S., editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 163–173, Orlando, FL, USA. ACM Press.

[Dyckhoff, 1992] Dyckhoff, R. (1992). Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807.

[Filinski, 1989] Filinski, A. (1989). Declarative Continuations: An Investigation of Duality in Programming Language Semantics. In *Category Theory and Comp. Sci.*, volume 389 of *LNCS*, pages 224–249. Springer-Verlag.

[Friedman et al., 1984] Friedman, D. P., Haynes, C. T., and Wand, M. (1984). Continuations and coroutines. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298.

[Friedman et al., 1986] Friedman, D. P., Haynes, C. T., and Wand, M. (1986). Obtaining coroutines with continuations. *Journal of Computer Languages*, 11(3/4):143–153.

[Gabbay, 1981] Gabbay, D. M. (1981). *Semantical Investigations in Heyting's Intuitionistic Logic*. Reidel, Dordrecht.

[Goré, 2000] Goré, R. (2000). Dual intuitionistic logic revisited. In Dyckhoff, R., editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2000, St Andrews, Scotland, UK, July 3-7, 2000, Proceedings*, volume 1847 of *Lecture Notes in Computer Science*, pages 252–267. Springer.

[Görnemann, 1971] Görnemann, S. (1971). A logic stronger than intuitionism. *The Journal of Symbolic Logic*, 36:249–261.

[Griffin, 1990] Griffin, T. G. (1990). A formulæ-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Langages*, pages 47–58.

[Group, 1997] Group, T. O. (1997). The Single UNIX Specification, Version 2. (http://www.UNIX-systems.org/online.html).

[Hyland and de Paiva, 1993] Hyland, M. and de Paiva, V. (1993). Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64(3):273–291.

[Kameyama, 1997] Kameyama, Y. (1997). A new formulation of the catch/throw mechanism. In Ida, T., Ohori, A., and Takeichi, M., editors, *Second Fuji International Workshop on Functional and Logic Programming*, Word Scientific, pages 106–122.

[Kameyama and Sato, 1998] Kameyama, Y. and Sato, M. (1998). A classical catch/throw calculus with tag abstraction and its strong normalizability. In Lin, X., editor, *Proc. the 4th Australasian Theory Symposium*, volume 20-3 of *Australian Computer Science Communications*, pages 183–197. Springer-Verlag.

[Kameyama and Sato, 2002] Kameyama, Y. and Sato, M. (2002). Strong normalizability of the non-deterministic catch/throw calculi. *Theoretical Computer Science*, 272(1–2):223–245.

[Kleene, 1952] Kleene, S. C. (1952). *Introduction to metamathematics*. North-Holland, Amsterdam. (Eighth reprint 1980).

[Krivine, 1994] Krivine, J.-L. (1994). Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78.

[Murthy, 1991] Murthy, C. R. (1991). Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science.

[Nakano, 1994a] Nakano, H. (1994a). A constructive logic behind the catch and throw mechanism. *Annals of Pure and Applied Logic*, 69(3):269–301.

[Nakano, 1994b] Nakano, H. (1994b). The non-deterministic catch and throw mechanism and its subject reduction property. In *Logic, Language and Computation*, volume 592 of *LNCS*, pages 61–72. Springer-Verlag.

[Nakano, 1995] Nakano, H. (1995). *The Logical Structures of the Catch and Throw Mechanism*. PhD thesis, The University of Tokyo.

[Ono, 1983] Ono, H. (1983). Model extension theorem and Graigs interpolation theorem for intermediate predicate logics. *Rep. Math. Logic*, 15:41–57.

[Parigot, 1992] Parigot, M. (1992). $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Logic Prog. and Autom. Reasoning*, volume 624 of *LNCS*, pages 190–201.

[Parigot, 1993a] Parigot, M. (1993a). Classical proofs as programs. In *Computationnal logic and theory*, volume 713 of *LNCS*, pages 263–276. Springer-Verlag.

[Parigot, 1993b] Parigot, M. (1993b). Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*.

[Pym and Ritter, 2001] Pym, D. and Ritter, E. (2001). On the semantics of classical disjunction. *Journal of Pure and Applied Algebra*, 159:315–338.

[Pym et al., 1996] Pym, D., Ritter, E., and Wallen, L. (1996). Proof-terms for classical and intuitionistic resolution. In McRobbie, M. A. and Slaney, J. K., editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 17–31, Berlin. Springer.

[Pym et al., 2000] Pym, D., Ritter, E., and Wallen, L. (2000). On the intuitionistic force of classical search. *Theoretical Computer Science*, 232(1-2):299–333.

[Ramsey, 1990] Ramsey, N. (1990). Concurrent programming in ML. Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, Princeton, NJ.

[Rauszer, 1974a] Rauszer, C. (1974a). A formalization of the propositional calculus of H-B logic. *Studia Logica*, 33(1):23–34.

[Rauszer, 1974b] Rauszer, C. (1974b). Semi-boolean algebras and their applications to intuitionistic logic with dual operations. In *Fundamenta Mathematicae*, volume 83, pages 219–249.

[Rauszer, 1980] Rauszer, C. (1980). An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathematicae*, volume 167. Institut Mathématique de l'Académie Polonaise des Sciences.

[Rehof and Sørensen, 1994] Rehof, N. J. and Sørensen, M. H. (1994). The λ_δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag.

[Reppy, 1990] Reppy, J. H. (1990). Asynchronous signals is standard ML. Technical Report TR90-1144, Cornell University, Computer Science Department.

[Reppy, 1995] Reppy, J. H. (1995). First-class synchronous operations. *Lecture Notes in Computer Science*, 907:235–252.

[Restall, 1997] Restall, G. (1997). Extending intuitionistic logic with subtraction. Available as http://consequently.org/papers/extendingj.pdf.

[Selinger, 2001] Selinger, P. (2001). Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260.

[Szabo, 1969] Szabo, M. E. (1969). *Gentzen Collected work*. North-Holland, Amsterdam.

[Wadler, 2003] Wadler, P. (2003). Call-by-value is dual to call-by-name. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden. ACM Press.

[Wand, 1980] Wand, M. (1980). Continuation-based multiprocessing. In Allen, J., editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA. The Lisp Company. Republished by ACM.

Chapitre 2

Procédures d'ordre supérieur et variables procédurales

∗. Les résultats présentés dans ce chapitre ont été obtenus en collaboration avec E. Polonowski et P. Valarcher et ont été publiés dans *ACM Transactions on Computational Logic. Special Issue on Implicit Computational Complexity.* Volume 10, Number 4, 2009, pp 1-36.

Extending the Loop Language with Higher-Order Procedural Variables

Abstract

We extend Meyer and Ritchie's Loop language with higher-order procedures and procedural variables and we show that the resulting programming language (called Loop$^\omega$) is a natural imperative counterpart of Gödel System T. The argument is two-fold:

1. we define a translation of the Loop$^\omega$ language into System T and we prove that this translation actually provides a lock-step simulation,
2. using a converse translation, we show that Loop$^\omega$ is expressive enough to encode any term of System T.

Moreover, we define the "iteration rank" of a Loop$^\omega$ program, which corresponds to the classical notion of "recursion rank" in System T, and we show that both translations preserve ranks. Two applications of these results in the area of implicit complexity are described.

1 Introduction

Primitive recursive functionals of finite type are representable as terms of the simply typed λ-calculus equipped with a type of natural numbers and primitive recursion at all types. This calculus (called System T) was first introduced by Gödel in its proof-theoretic study of Peano arithmetic (see its Dialectica paper on the consistency of arithmetic [Gödel, 1958], reproduced with English translation in [Gödel, 1990]). Moreover, System T is well suited to give a formal semantics to constructs found in (higher-order) programming languages [Girard et al., 1989].

The Loop language [Meyer and Ritchie, 1976] is a core imperative language in which programs consist only of assignments, sequences, and bounded loops. Meyer and Ritchie's proved in particular that Loop programs compute exactly the class of primitive recursive functions. The Loop language has since been widely studied in the literature (see for instance the textbooks [Davis and Weyuker, 1983] and [Calude, 1988]).

In this paper, we introduce a statically typed extension of the Loop language (called Loop$^\omega$) with higher-order procedures and procedural variables and we argue that this programming language is a natural imperative counterpart of Gödel System T. The argument is two-fold:

1. we define a translation of the Loop$^\omega$ language into System T and we prove that this translation actually provides a lock-step simulation,
2. using a converse translation, we show that Loop$^\omega$ is expressive enough to encode any term of System T.

The first result is the main contribution of this paper: we prove that Loop$^\omega$ is not only extensionally equivalent to System T, but also intensionally. In order to develop this point, we need to give some details about the formal semantics of Loop$^\omega$. The overall design of the language follows mostly the principles advocated in [Schmidt, 1994]. The operational semantics of Loop$^\omega$ is presented first as a natural semantics (also called "big-step" semantics) in the style of [Kahn, 1987]. Then a transition semantics (also called "small-step" semantics [Plotkin, 1981]) tailored of the lock-step simulation and which refines the natural semantics is defined. The simulation theorem states that each evaluation step of a Loop$^\omega$ program (according to the transition semantics) is mapped to one reduction step of the transformed program in System T (using a call-by-value strategy). As a corollary of this theorem, we obtain that all Loop$^\omega$ programs are always terminating.

Note that LOOP$^\omega$ is a genuine imperative language with first-class procedures (true closures) and mutable procedural variables (aka function pointers). For instance, any LOOP$^\omega$ can easily be written using $C\#$ syntax (where anonymous first-class procedures are called delegates [ISO, 2003]) and then compiled with a $C\#$ compiler. However, it is a "pure" imperative language in the following sense: its type system forbids side-effects, parameter-induced aliasing (which are controversial features known to complicate the semantics). Moreover the type system forbids the well-known back-patching technique which exploits procedural variables to define arbitrary recursive procedures [Landin, 1964] (and LOOP$^\omega$ is thus a "total" programming language as advocated in [Turner, 2004]). As a consequence of these choices, both semantics presented in this paper are simple but rather unusual for imperative languages: they are both location-free semantics (see [Donahue, 1977], [Plotkin, 1981] and [Felleisen and Friedman, 1987]) and they rely crucially on the distinction between mutable and read-only (immutable) variables.

The second result is obtained by defining a converse translation (from System T into LOOP$^\omega$) and proving that the composition the two translations yields the identity in System T (up to $\beta\pi$-equivalence). Since both translations are syntax directed (compositional) and type-preserving, we can be even more specific about the correspondence between a functional program and its imperative counterpart. Recall that a major result concerning System T from [Kreisel, 1951] states that functions on the natural numbers that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see also the survey [Avigad and Feferman, 1998] or the textbook [Schütte, 1967]). More precisely, that there is a syntactic hierarchy of fragments T_n of System T such that the class of functions representable in T_n is identical to the class of functions provably recursive in the fragment of Peano arithmetic where induction is restricted to Σ_{n+1} formulas. In particular, T_0 corresponds to the class of primitive recursive functions.

We define thus a similar hierarchy of fragments LOOP^n of LOOP$^\omega$ and we show that both translations relate programs of LOOP^n and terms of T_n. As a corollary, we obtain that the functions representable in a language with higher-order procedures but without procedural variables (which is a sub-language of LOOP^0) are primitive recursive. This corollary generalizes thus previous results presented in [Crolard et al., 2006] where Meyer and Ritchie's LOOP language was translated into T_0. On the other hand, the Ackermann function which is known not to be primitive recursive is representable in T_1 and thus also in LOOP^1. As far as we know, LOOP$^\omega$ is the first total imperative language allowing to program the Ackermann function.

Applications. A first application of these results is extensional: we derive a new characterization of the class of Csillag-Kalmar elementary functions (the class $\mathcal{E}_3$ in Grzegorczyk hierarchy). In [Beckmann and Weiermann, 2000], such a characterization is based on a syntactic restriction on terms of a variant of Gödel System T. As a corollary of various properties of our two translations we provide an imperative counterpart of this restriction. In particular, we obtain that any LOOP$^\omega$ program in which any bound of loop is a read-only input variable is elementary.

A second application is intensional and is related to the so-called *minimum problem*. In [Colson and Fredholm, 1998], the authors proved that in call-by-value System T, any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. As a consequence of this property, there is no term that computes the minimum of two natural numbers n and m in time $O(min(n,m))$. As a corollary of the lock-step simulation we obtain a similar negative result for LOOP$^\omega$ programs.

Related works. There has been a lot of work on the semantics of Algol (major contributions are collected in [O'Hearn and Tennent, 1997]) and this work has influenced the design of most modern programming languages. For instance, the LOOP$^\omega$ language is very close in spirit to the language partially described in [Reynolds, 1978] whose purpose was in particular to show that it is possible to avoid aliasing while retaining Algol-like higher-order procedures. More generally, our work can also be seen as providing a denotational semantics for an imperative language using a λ-calculus and thus follows the Scott-Strachey tradition [Stoy, 1977].

The idea of translating an imperative program into a functional one actually goes back to [McCarthy, 1960]. However, our translation is somewhat closer to the state monad [Moggi, 1991, Wadler, 1990] used to encode a mutable state (or references) in a pure functional language. Although this encoding is usually global and thus changes the type of all terms, a local encoding can be obtained if an effect system [Gifford and Lucassen, 1986] is used for the source language (instead of a conventional type system). A simulation based on such an encoding is described in [Wadler, 1998].

Similarly, in [Filliâtre, 2003], the author defines a monadic translation of simply-typed functional programs with references (annotated with Floyd-Hoare assertions) into a type theory. Their translation provides the core of the proof technique developed afterward for imperative programs [Filliâtre and Marché, 2004]. Although similar to the one described in this paper, their translation is only considered in the context of program verification (no simulation is defined).

Finally, compiling one programming language into another in order to derive complexity properties is a common approach. The reader is referred for instance to [Jones, 1997] for various applications of this technique.

Plan of the paper. In Section 2, we present our variant of Gödel System T. In Section 3, we describe the LOOP$^\omega$ language (syntax, type system and semantics). In Section 4, we show how to translate LOOP$^\omega$ programs into functional programs and we prove the simulation theorem. In Section 5, we define the converse translation, and we use it to prove that LOOP$^\omega$ is as expressive as System T. Finally, in Section 6, we describe the two applications.

2 Gödel System T

Gödel System T is usually defined as the simply typed λ-calculus extended with a type of natural numbers and with primitive recursion at all types. We consider in this paper a variant of System T with product types (tuples and n-ary functions) and a constant-time predecessor operation. Moreover, since we are mainly interested here in the call-by-value evaluation strategy, we formulate this system directly as a context semantics (a set of reduction rules together with an inductive definition of evaluation contexts). As usual, we consider terms up to α-conversion and the set $FId(t)$ of free identifiers of a term t is defined in the standard way. The rewriting system is summarized in Figure 2.1, where variables $x, x_1, ..., x_n$ range over a set of identifiers and $t[v_1/x_1, ..., v_n/x_n]$ denotes the usual capture-avoiding substitution. We also recall the type system in Figure 2.2 and we consider only well-typed terms in the sequel.

Remark 2.1. In order to distinguish the successor S (which is a constructor) from the successor seen as an operation (whose evaluation should imply a reduction step), we use the keyword **succ** as an abbreviation for $\lambda x.S(x)$. Note that we also consider a primitive constant-time predecessor (called **pred**) for complexity reasons: although this function is clearly definable in System T its complexity would be at least linear under the call-by-value evaluation strategy [Colson and Fredholm, 1998]. More details on this question are given in Section 6.2 and Appendix A.

Notation 2.2. *We use the more verbose syntax* $\mathbf{fn}\ (\vec{x}:\vec{\sigma}) \Rightarrow t$ *instead of simply* $\lambda\vec{x}.t$ *whenever we wish to make the types explicit. Moreover, we write* $\vec{\tau}^{\times}$ *as an abbreviation for* $\tau_1 \times ... \times \tau_n$ *and we consider unit as the special case of* $\vec{\tau}^{\times}$ *obtained when* $n = 0$.

Remark 2.3. As usual [Landin, 1964], we write $\mathbf{let}\ (x_1, ..., x_n) = u\ \mathbf{in}\ t$ be an abbreviation for the redex $\lambda(x_1, ..., x_n).t\ u$. The following typing rule and evaluation rule can be obtained:

$$\frac{\Gamma \vdash u : (\tau_1 \times ... \times \tau_n) \quad \Gamma, x_1 : \tau_1, ... x_n : \tau_n \vdash t : \sigma}{\Gamma \vdash \mathbf{let}\ (x_1, ..., x_n) = u\ \mathbf{in}\ t : \sigma}$$

$$C[\mathbf{let}\ (x_1, ..., x_n) = (v_1, ..., v_n)\ \mathbf{in}\ t] \rightsquigarrow C[t[v_1/x_1, ..., v_n/x_n]]$$

(types)

$$\begin{array}{rcl} \tau & ::= & \mathit{int} \\ & | & \mathit{unit} \\ & | & \tau_1 \to \tau_2 \\ & | & \tau_1 \times \ldots \times \tau_n \end{array}$$

(values)

$$\begin{array}{rcl} v & ::= & x \\ & | & 0 \\ & | & S(v) \\ & | & (v_1, \ldots, v_n) \\ & | & \lambda(x_1, \ldots, x_n).t \end{array}$$

(terms)

$$\begin{array}{rcl} t & ::= & x \\ & | & 0 \\ & | & S(t) \\ & | & \mathbf{pred}(t) \\ & | & t_1\ t_2 \\ & | & \lambda(x_1, \ldots, x_n).t \\ & | & (t_1, \ldots, t_n) \\ & | & \mathbf{rec}(t_1, t_2, t_3) \end{array}$$

(contexts)

$$\begin{array}{rcl} C[\,] & ::= & [\,] \\ & | & C[\,]\ t \\ & | & v\ C[\,] \\ & | & S(C[\,]) \\ & | & \mathbf{pred}(C[\,]) \\ & | & \mathbf{rec}(C[\,], t_2, t_3) \\ & | & \mathbf{rec}(v_1, C[\,], t_3) \\ & | & \mathbf{rec}(v_1, v_2, C[\,]) \\ & | & (v_1, \ldots v_{i-1}, C[\,], t_{i+1} \ldots, t_n) \end{array}$$

(evaluation rules)

$$\begin{array}{rcl} C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\ C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\ C[\mathbf{rec}(0, v_2, \lambda x.t)] & \rightsquigarrow & C[v_2] \\ C[\mathbf{rec}(S(v_1), v_2, \lambda x.t)] & \rightsquigarrow & C[t[v_1/x]\ \mathbf{rec}(v_1, v_2, \lambda x.t)] \\ C[\lambda(x_1, \ldots, x_n).t\ (v_1, \ldots, v_n)] & \rightsquigarrow & C[t[v_1/x_1, \ldots, v_n/x_n]] \end{array}$$

Figure 2.1. Gödel System T

$$\frac{x{:}\,\tau \in \Gamma}{\Gamma \vdash x{:}\,\tau} \quad (\textsc{ident})$$

$$\Gamma \vdash 0{:}\,\mathit{int} \quad (\textsc{zero})$$

$$\frac{\Gamma \vdash t{:}\,\mathit{int}}{\Gamma \vdash S(t){:}\,\mathit{int}} \quad (\textsc{succ})$$

$$\frac{\Gamma \vdash t{:}\,\mathit{int}}{\Gamma \vdash \mathbf{pred}(t) : \mathit{int}} \quad (\textsc{pred})$$

$$\frac{\Gamma \vdash t_1{:}\,\tau_1 \quad \ldots \quad \Gamma \vdash t_n{:}\,\tau_n}{\Gamma \vdash (t_1, \ldots, t_n){:}\,\tau_1 \times \ldots \times \tau_n} \quad (\textsc{tuple})$$

$$\frac{\Gamma, x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n \vdash t{:}\,\sigma}{\Gamma \vdash \lambda(x_1, \ldots, x_n).t\ : (\tau_1 \times \ldots \times \tau_n) \to \sigma} \quad (\textsc{abs})$$

$$\frac{\Gamma \vdash t_1{:}\,\sigma \to \tau \quad \Gamma \vdash t_2{:}\,\sigma}{\Gamma \vdash t_1\ t_2 : \tau} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t_1{:}\,\mathit{int} \quad \Gamma \vdash t_2{:}\,\tau \quad \Gamma \vdash t_3{:}\,\mathit{int} \to \tau \to \tau}{\Gamma \vdash \mathbf{rec}(t_1, t_2, t_3) : \tau} \quad (\textsc{rec})$$

Figure 2.2. Functional type system

Finally, note that **let** $(x_1, \ldots, x_n) = [\,]$ **in** t is an evaluation context.

Let us recall the well-known infinite syntactic hierarchy of fragment T_n of Gödel System T. We call "recursion rank" of a term t the maximum degree of the types of the recursors which occur in t, where the degree of a type is defined as follows:

Definition 2.4. *The degree $\partial(\tau)$ of a type τ is defined inductively by:*

– $\partial(\mathit{int}) = 0$

– $\partial(\sigma \to \tau) = \max(\partial(\sigma)+1, \partial(\tau))$

– $\partial(\tau_1 \times ... \times \tau_n) = \max(\partial(\tau_1), ..., \partial(\tau_n))$

Remark 2.5. Using standard type isomorphisms, any type is isomorphic either to *unit* or to some type which does not contain *unit*. Thus we may consider without restriction that τ does not contain *unit* in the above definition.

Definition 2.6. *The fragment T_n of System T is defined as the set of terms with recursion rank less or equal to n.*

2.1 Example: the Ackermann function

Our running example in this paper is the Ackermann function. This function is known not to be primitive recursive [Peter, 1968] but it can be represented in System T, for instance as follows:

$$ack(m,n) = (\mathbf{rec}(m, \lambda y.\mathbf{succ}(y), \lambda i.\lambda h.\lambda y.\mathbf{rec}(y, (h\ S(0)), \lambda j.\lambda k.(h\ k)))\ n)$$

It is well-known (see [Avigad and Feferman, 1998] for instance) that functions of type $N^k \to N$ that can be represented by a term of T_0 correspond exactly to primitive recursive functions. Not surprisingly, the term given in the above definition does not belong to T_0 but to T_1 (the outermost **rec** in the above definition has type $int \to int$ and thus has degree 1).

3 The higher-order Loop language

In this section, we present the Loop$^\omega$ language. First, we describe the syntax and the type system, then we detail the semantics of the language. Since the structured operational semantics is tailored for the lock-step simulation, we first present the natural semantics. Our purpose is to give evidence that the various syntactic constructs have the usual semantics. In particular, this semantics extends the usual semantics for first order Loop programs with full-fledged higher-order procedures.

However natural semantics relates a program and an initial store directly to some final store and thus describes only the evaluation of terminating programs. In particular, it fails to distinguish between a non-terminating program and a run-time error. For this reason, we also define a transition semantics which refines the natural semantics. We use the transition semantics for proving the soundness of type systems and the simulation theorem (in order to derive that Loop$^\omega$ programs are always terminating).

Both semantics are location-free semantics. The benefit is clear: this kind of semantics is simpler than a traditional two-level semantics (where the environment binds variables to locations and the store maps locations to values). The main drawback is the difficulty to account for advanced features such as variable aliasing. However, for our purpose, a location-free semantics is sufficient. A discussion on this subject can be found in [Plotkin, 1981] p. 70 (see also [Felleisen and Friedman, 1987], [Donahue, 1977] and [Reynolds, 1981]).

3.1 Syntax

The syntax of imperative types and phrase types is the following:

$$\begin{aligned} \sigma, \tau &::= int \mid \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \\ \mathcal{T} &::= \tau \mid \mathbf{comm} \mid \mathbf{seq} \end{aligned}$$

Note that we consider only two formal parameter modes **in** and **out** (borrowed from Ada [DOD, 1980]) which specify "abstractly" the direction of data flow between caller and callee (without implying a specific parameter-passing mechanism).

The raw syntax of imperative programs is given below. There is nothing particular to this syntax except that we annotate each block $\{s\}_{\vec{x}}$ with a list of variables $\vec{x}$ (see Remark 3.2). In the following grammar, x, y, z range over a set of identifiers, $\bar{q}$ ranges over natural numbers (i.e., constant literals) and ε denotes the empty sequence.

$$
\begin{array}{rrcl}
(\textit{command}) & c & ::= & \{s\}_{\vec{x}} \\
 & & | & \textbf{for } y := 0 \textbf{ until } e\ \{s\}_{\vec{x}} \\
 & & | & y := e \ \mid\ \textbf{inc}(y) \ \mid\ \textbf{dec}(y) \\
 & & | & p(\vec{e}\,;\vec{y}\,) \\
 & & & \\
(\textit{sequence}) & s & ::= & \varepsilon \\
 & & | & c\,;\ s \\
 & & | & \textbf{cst } y = e;\ s \\
 & & | & \textbf{var } y{:}\,\tau := e;\ s \\
 & & & \\
(\textit{anonymous procedure}) & a & ::= & \textbf{proc } (\textbf{in } \vec{y}{:}\,\vec{\tau}\,;\textbf{out } \vec{z}{:}\,\vec{\sigma}\,)\ \{s\}_{\vec{z}} \\
 & & & \\
(\textit{expression}) & e & ::= & y \ \mid\ \bar{q} \ \mid\ a \\
(\textit{procedure}) & p & ::= & y \ \mid\ a \\
(\textit{value}) & w & ::= & \bar{q} \ \mid\ a
\end{array}
$$

Remark 3.1. Note that the body of a procedure is annotated exactly by its **out** parameters. Besides, since anonymous procedures are not very popular in imperative languages, we use in the examples the more conventional notation for declaring local (named) procedures:

$$\textbf{proc } p(\textbf{in } \vec{y}{:}\,\vec{\tau}\,;\textbf{out } \vec{z}{:}\,\vec{\sigma}\,)\ \{s_1\}_{\vec{z}};\ s_2$$

Following Landin's correspondence principle [Landin, 1964], this notation is defined as an abbreviation for a constant declaration:

$$\textbf{cst } p = \textbf{proc } (\textbf{in } \vec{y}{:}\,\vec{\tau}\,;\textbf{out } \vec{z}{:}\,\vec{\sigma}\,)\ \{s_1\}_{\vec{z}};\ s_2$$

3.2 Type system

The type system of Loop$^\omega$ may be seen as a simple effect system [Gifford and Lucassen, 1986, Talpin and Jouvelot, 1994] since it is able to guarantee the absence of side-effects, aliasing and fix-points in well-typed programs. Its main feature is the distinction between mutable variables and read-only variables. More formally, a typing environment has the form $\Gamma;\ \Omega$ where Γ and Ω are (possibly empty) lists of pairs $x{:}\,\tau$ (x ranges over variables and τ over types). Γ stands for read-only variables (constants and **in** parameters) and Ω stands for mutable variables (local variables and **out** parameters). The type system is given in Figure 3.1. As usual, we consider programs up to renaming of bound variables, where the notion of free variable of a command is defined in the standard way.

Remark 3.2. *(scoping rules).* As usual for C-like languages, the scope of a constant (rule T.CST) or a variable (rule T.VAR) extends from the point of declaration to the end of the block containing the declaration. Moreover, the type system guarantees that in a block $\{s\}_{\vec{x}}$, the variables $\vec{x}$ are visible and they contain all the free mutable variables occurring in the sequence. For simplicity, we assume that these annotations are supplied by the programmer, but missing annotations can automatically be inferred as follows:

- a block or a loop is annotated by its free mutable variables,
- the body of a procedure is annotated by its **out** parameters.

$$\frac{x{:}\,\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x{:}\,\tau} \quad (\textsc{t.env})$$

$$\Gamma;\Omega \vdash \bar{q}{:}\,\mathit{int} \quad (\textsc{t.num})$$

$$\frac{}{\Gamma;\Omega \vdash \varepsilon{:}\,\mathbf{seq}} \quad (\textsc{t.seq-i})$$

$$\frac{\Gamma;\Omega \vdash c{:}\,\mathbf{comm} \qquad \Gamma;\Omega \vdash s{:}\,\mathbf{seq}}{\Gamma;\Omega \vdash c\,;s\,{:}\,\mathbf{seq}} \quad (\textsc{t.seq-ii})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma, y{:}\,\tau;\Omega \vdash s{:}\,\mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{cst}\ y = e;\ \ s\,{:}\,\mathbf{seq}} \quad (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma;y{:}\,\tau,\Omega \vdash s{:}\,\mathbf{seq} \quad s \neq \varepsilon}{\Gamma;\Omega \vdash \mathbf{var}\ y{:}\,\tau := e;\ \ s\,{:}\,\mathbf{seq}} \quad (\textsc{t.var})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma;\vec{x}{:}\,\vec{\sigma} \vdash s{:}\,\mathbf{seq}}{\Gamma;\Omega \vdash \{s\}_{\vec{x}}{:}\,\mathbf{comm}} \quad (\textsc{t.comm})$$

$$\frac{y{:}\,\mathit{int} \in \Omega}{\Gamma;\Omega \vdash \mathbf{inc}(y){:}\,\mathbf{comm}} \quad (\textsc{t.inc})$$

$$\frac{y{:}\,\mathit{int} \in \Omega}{\Gamma;\Omega \vdash \mathbf{dec}(y){:}\,\mathbf{comm}} \quad (\textsc{t.dec})$$

$$\frac{y{:}\,\tau \in \Omega \qquad \Gamma;\Omega \vdash e{:}\,\tau}{\Gamma;\Omega \vdash y := e{:}\,\mathbf{comm}} \quad (\textsc{t.assign})$$

$$\frac{\vec{x} \subset \Omega \qquad \Gamma;\Omega \vdash e{:}\,\mathit{int} \qquad \Gamma, y{:}\,\mathit{int};\vec{x}{:}\,\vec{\sigma} \vdash s{:}\,\mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}}{:}\,\mathbf{comm}} \quad (\textsc{t.for})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}{:}\,\vec{\tau};\vec{z}{:}\,\vec{\sigma} \vdash s{:}\,\mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\,\vec{\tau};\mathbf{out}\ \vec{z}{:}\,\vec{\sigma})\,\{s\}_{\vec{z}}{:}\,\mathbf{proc}\ (\mathbf{in}\ \vec{\tau};\mathbf{out}\ \vec{\sigma})} \quad (\textsc{t.proc})$$

$$\frac{\Gamma;\Omega \vdash p{:}\,\mathbf{proc}\ (\mathbf{in}\ \vec{\tau};\mathbf{out}\ \vec{\sigma}) \qquad \Gamma;\Omega \vdash \vec{e}{:}\,\vec{\tau} \qquad \vec{r}{:}\,\vec{\sigma} \in \Omega}{\Gamma;\Omega \vdash p(\vec{e}\,;\vec{r}\,){:}\,\mathbf{comm}} \quad (\textsc{t.call})$$

Figure 3.1. Imperative type system

Remark 3.3. *(no aliasing).* In order to avoid parameter-induced aliasing problems, we assume that all r_i are pairwise distinct in rule (T.CALL). Indeed, we will see that in our semantics, mutable variables directly denote values and consequently different variables cannot refer to the same location in the store. However, aliasing is known to be problematic for most aspects of programming languages (see for instance [Gellerich and Plödereder, 2001] and [Filliâtre, 2003]) and we regard the absence of parameter-induced aliasing in Loop$^{\omega}$ more as a feature than as a limitation.

Remark 3.4. *(no side-effects).* Rule (T.PROC) implies that the only mutable variables which may occur inside the body of a procedure are its **out** parameters and its local mutable variables. This is enough to guarantee the absence of side-effects (i.e., modification of non-local variables). The purpose of this restriction is mainly to simplify the semantics of the language. However, side-effects can still be simulated in most cases by passing the non-local variable as an explicit **in out** parameter. This simulation is often referred to as the "state-passing transform" (see for instance [Filinski, 1994] and [Wadler, 1990]).

Remark 3.5. *(no fix-points).* Rule (T.PROC) also forbids the reading of non-local mutable variables: this is necessary to prevents the definition of fix-points in the language. Indeed, there is a well-known technique called "tying the recursive knot" [Landin, 1964] which takes advantage of higher-order mutable variables (or

function pointers) to define arbitrary recursive functions. This technique is used for instance in Scheme's semantics of the **letrec** construct [Kelsey et al., 1998]. Here is such a definition of a fix-point in the LOOP$^\omega$ syntax, where *fix*: **proc** (**out** *int*) is a mutable variable, but this command is not typable in LOOP$^\omega$ (since *fix* occurs in the body of f):

$$
\begin{array}{l}
\{ \\
\quad \textbf{var}\ f\colon \textbf{proc}\ (\textbf{out}\ int) := \textbf{proc}\ (\textbf{out}\ x\colon int)\{\ fix(x);\ \}_x; \\
\quad fix := f; \\
\}_{fix}
\end{array}
$$

On the other hand, local procedures are not required to be closed: non-local read-only variable (such as **in** parameters of enclosing procedures) are allowed. We shall see that this is sufficient to encode a pure functional language.

3.3 Natural semantics

Since the evaluation relation in only defined for well-typed states, we also need to define the notion of well-typed store.

Definition 3.6. *A store μ is a finite mapping from (mutable) variables to closed imperative values (i.e., integer literals and procedures). A state is a pair (c, μ) consisting of a command c and μ.*

Definition 3.7. (store typing). *We say that μ is typable in Ω, which we write as $\Omega \vdash \mu$, if and only if for all $x\colon \tau \in \Omega$, $x \in dom(\mu)$ and $\emptyset;\emptyset \vdash \mu(x)\colon \tau$.*

Definition 3.8. (state typing). *We say that a state (c, μ) is typable in Ω, which we write as $\Omega \vdash (c, \mu)$, if and only if $\emptyset;\Omega \vdash c\colon \textbf{comm}$ and $\Omega \vdash \mu$.*

Note that expressions do not require any evaluation (since they are either variables or values), but only fetching the corresponding value from the store whenever the expression is a mutable variable. We introduce thus the following notation:

Notation 3.9. *Given a store μ, let φ_μ be the trivial extension of μ to expressions defined as follows $\varphi_\mu(x) = \mu(x)$ if x is a variable and $\varphi_\mu(w) = w$ otherwise. In the sequel, we write $e =_\mu w$ for $\varphi_\mu(e) = w$.*

In order to assign default values to **out** parameters, we define a closed imperative value for each imperative type.

Definition 3.10. *For each type σ, we define inductively a default closed imperative value $\epsilon(\sigma)$ as follows:*

- $\epsilon(int) = 0$
- $\epsilon(\textbf{proc}\ (\textbf{in}\ \vec{\tau};\textbf{out}\ \vec{\sigma})) = \textbf{proc}\ (\textbf{in}\ \vec{y}\colon\vec{\tau};\textbf{out}\ z\colon\vec{\sigma})\{\}_{\vec{z}}$

Lemma 3.11. *The typing judgment $\emptyset;\emptyset \vdash \epsilon(\sigma)\colon \sigma$ is derivable.*

Remark 3.12. Default values could be dispensed with using standard data-flow analysis (see [Appel, 1998] for instance). This technique (also called liveness analysis) allows to determine whether there is a potential execution path on which a variable is used before it has been assigned an initial value. We preferred not to complicate the static analysis of LOOP$^\omega$ programs although such an analysis is certainly convenient from a practical standpoint.

Remark 3.13. In the sequel, we shall allow for uninitialized local variables: they are assumed to be implicitly initialized by the default value corresponding to their type. More precisely, $\mathbf{var}\ y{:}\tau;\ s$ is an abbreviation for: $\mathbf{var}\ y{:}\tau{:=}\epsilon(\tau);\ s$.

Notation 3.14. *Let c be a command. We write $c[x \leftarrow w]$ for the substitution of a read-only variable x by a closed imperative value w and $c[y \leftarrow\!\!\rightarrow z]$ for the renaming of a mutable variable y by a mutable variable z. The formal definitions are given in Appendix B.*

We are now ready to define the natural semantics of Loop$^\omega$. The inductive definition of the evaluation relation written $\Downarrow$ is summarized in Figure 3.2 (where $\mu[y \leftarrow w]$ denotes the store obtained by replacing the value of

$$\frac{(s,\mu)\Downarrow\mu'}{(\{s\}_{\vec{z}},\mu)\Downarrow\mu'} \quad \text{(N.BLOCK)}$$

$$\frac{(s,\mu)\Downarrow\mu'}{((\{\}_{\vec{z}};s),\mu)\Downarrow\mu'} \quad \text{(N.SEQ-I)}$$

$$\frac{(c,\mu)\Downarrow\mu' \qquad (s,\mu')\Downarrow\mu''}{((c;s),\mu)\Downarrow\mu''} \quad \text{(N.SEQ-II)}$$

$$\frac{(s,\mu)\Downarrow\mu'}{((\{\mathbf{var}\ y{:}\tau{:=}e;\}_{\vec{z}};s),\mu)\Downarrow\mu'} \quad \text{(N.VAR-I)}$$

$$\frac{e=_\mu w \qquad (s,(\mu,y\leftarrow w))\Downarrow(\mu',y\leftarrow w'))}{((\mathbf{var}\ y{:}\tau{:=}e;\ s),\mu))\Downarrow\mu'} \quad \text{(N.VAR-II)}$$

$$\frac{e=_\mu w \qquad (s,\mu[y\leftarrow w])\Downarrow\mu'}{((y:=e;s),\mu)\Downarrow\mu'} \quad \text{(N.ASSIGN)}$$

$$\frac{\mu(y)=\bar{q}}{(\mathbf{inc}(y),\mu))\Downarrow\mu[y\leftarrow\overline{q+1}]} \quad \text{(N.INC)}$$

$$\frac{\mu(y)=\bar{q}}{(\mathbf{dec}(y),\mu)\Downarrow\mu[y\leftarrow\overline{q\dot{-}1}]} \quad \text{(N.DEC)}$$

$$\frac{\vec{e}=_\mu\vec{w},p=_\mu\mathbf{proc}\,(\mathbf{in}\ \vec{y}{:}\vec{\sigma};\mathbf{out}\ \vec{z}{:}\vec{\tau})\,\{s\}_{\vec{z}} \qquad (\{s[\vec{y}\leftarrow\vec{w}][\vec{z}\leftarrow\!\!\rightarrow\vec{r}]\}_{\vec{r}},\mu[\vec{r}\leftarrow\epsilon(\vec{\tau})])\Downarrow\mu'}{(p(\vec{e};\vec{r}),\mu)\Downarrow\mu'} \quad \text{(N.CALL)}$$

$$\frac{e=_\mu w \qquad (s[y\leftarrow w],\mu)\Downarrow\mu'}{((\mathbf{cst}\ y=e;\ s),\mu)\Downarrow\mu'} \quad \text{(N.CST)}$$

$$\frac{e=_\mu\bar{0}}{(\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}},\mu)\Downarrow\mu} \quad \text{(N.FOR-I)}$$

$$\frac{e=_\mu\overline{q+1} \qquad (\mathbf{for}\ y:=0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{z}},\mu)\Downarrow\mu' \qquad (\{s\}_{\vec{z}}[y\leftarrow\bar{q}],\mu')\Downarrow\mu''}{(\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}},\mu)\Downarrow\mu''} \quad \text{(N.FOR-II)}$$

Figure 3.2. Natural semantics

variable y in μ by w and $(\mu, y\leftarrow w)$ denotes the store obtained by extending the store μ with a new variable y mapped to w).

Remark 3.15. We rely on the renaming and substitution meta-operations in rule (N.CALL) in order to avoid building closures in the semantics. Besides, **in** parameters are passed by copy (i.e., by value) whereas **out** parameters are passed by reference. Note that since parameter-induced aliasing is forbidden, this latter choice has no consequence on the denotational semantics of the language (but it is important from a complexity standpoint).

3.4 Transition semantics

As expected, the transition semantics of Loop$^\omega$ is given by a transition system which defines inductively a binary relation between states [Plotkin, 1981]. The main design choices were influenced by the expected cost of each command. For instance, this explains the rule (s.assign) where the assignment is dealt with directly inside the sequence. Similarly, the notation $=_\mu$ allows us to hide the cost of fetching the value of a variable from the store. The transition semantics is summarized in Figure 3.3.

Remark 3.16. This semantics is clearly deterministic since there is always at most one rule which can be applied (depending on the content of the store and the shape of the command). Moreover, the only case where no rule can be applied corresponds to the final state (when the program is reduced to an empty block).

Remark 3.17. It is worth mentioning that rules (s.var-i) and (s.var-ii) allows to give a simple semantics to local variable without dealing with an explicit stack. Although this is usual in natural semantics, this technique is not widespread in transition semantics. The ingenious idea consisting in updating a local variable directly in the source (in rule s.var-ii) is attributed to Eugene Fink in [Reynolds, 1998] p. 130. This technique is however well-known in functional language semantics [Felleisen and Friedman, 1987].

As expected, the "subject reduction" property holds for the transition semantics. The proof of the following theorem is given in Appendix B.

Theorem 3.18. *For any environment Ω and any state (c, μ), we have that $\emptyset; \Omega \vdash (c, \mu)$ and $(c, \mu) \mapsto (c', \mu')$ implies $\emptyset; \Omega \vdash (c', \mu')$ and $dom(\mu) = dom(\mu')$.*

Moreover, the equivalence of the natural and transition semantics for Loop$^\omega$ can be proved by establishing the following two usual lemmas.

Lemma 3.19. *The relation $(\{s\}_{\vec{x}}, \mu) \mapsto^\star (\{\}_{\vec{x}}, \mu')$ is closed under the defining conditions of the $\Downarrow$ relation.*

Lemma 3.20. *The $\Downarrow$ relation is closed under head expansion: if $(\{s\}_{\vec{x}}, \mu) \mapsto (\{s'\}_{\vec{x}}, \mu')$ and $(\{s'\}_{\vec{x}}, \mu') \Downarrow \mu''$ then $(\{s\}_{\vec{x}}, \mu) \Downarrow \mu''$.*

4 Lock-step simulation

In this section, we show how to translate a Loop$^\omega$ program into a term of System T and we prove the simulation theorem. Then we exhibit a hierarchy of fragments Loop^n which is an imperative counterpart of the fragments T_n of Gödel System T. Finally, we introduce the notion of "singular" Loop$^\omega$ program whose translation does not require the product type in System T.

4.1 Translation

In order to translate default imperative values, we shall need corresponding default functional values.

Definition 4.1. *For each type σ, we define inductively a default closed value $\delta(\sigma)$ as follows:*

- $\delta(int) = 0$
- $\delta(\sigma \to \tau) = \mathbf{fn}\ (x{:}\sigma) \Rightarrow \delta(\tau)$

$$\frac{(s,\mu)\mapsto(s',\mu')}{(\{s\}_{\vec{z}},\mu)\mapsto(\{s'\}_{\vec{z}},\mu')} \quad \text{(S.BLOCK)}$$

$$((\{\}_{\vec{z}};s),\mu)\mapsto(s,\mu) \quad \text{(S.SEQ-I)}$$

$$\frac{(c,\mu)\mapsto(c',\mu')}{((c;s),\mu)\mapsto((c';s),\mu')} \quad \text{(S.SEQ-II)}$$

$$\frac{e=_\mu w \qquad (s,(\mu,y\leftarrow w))\mapsto(\varepsilon,(\mu',y\leftarrow w'))}{((\mathbf{var}\ y{:}\tau{:=}e;\ s),\mu)\mapsto(\varepsilon,\mu')} \quad \text{(S.VAR-I)}$$

$$\frac{e=_\mu w \qquad (s,(\mu,y\leftarrow w))\mapsto(s',(\mu',y\leftarrow w'))}{((\mathbf{var}\ y{:}\tau{:=}e;\ s),\mu)\mapsto((\mathbf{var}\ y{:}\tau{:=}w';\ s'),\mu')} \quad \text{(S.VAR-II)}$$

$$\frac{e=_\mu w}{((y:=e;s),\mu)\mapsto(s,\mu[y\leftarrow w])} \quad \text{(S.ASSIGN)}$$

$$\frac{\mu(y)=\bar{q}}{(\mathbf{inc}(y),\mu)\mapsto(y:=\overline{q+1},\mu)} \quad \text{(S.INC)}$$

$$\frac{\mu(y)=\bar{q}}{(\mathbf{dec}(y),\mu)\mapsto(y:=\overline{q\dot{-}1},\mu)} \quad \text{(S.DEC)}$$

$$\frac{\vec{e}=_\mu\vec{w} \qquad p=_\mu\mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\vec{\sigma};\mathbf{out}\ \vec{z}{:}\vec{\tau})\,\{s\}_{\vec{z}}}{(p(\vec{e};\vec{r}),\mu)\mapsto(\{s[\vec{y}\leftarrow\vec{w}][\vec{z}\hookleftarrow\vec{r}]\}_{\vec{r}},\mu[\vec{r}\leftarrow\epsilon(\vec{\tau})])} \quad \text{(S.CALL)}$$

$$\frac{e=_\mu w}{((\mathbf{cst}\ y=e;\ s),\mu)\mapsto(s[y\leftarrow w],\mu)} \quad \text{(S.CST)}$$

$$\frac{e=_\mu\bar{0}}{(\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}},\mu)\mapsto(\{\}_{\vec{z}},\mu)} \quad \text{(S.FOR-I)}$$

$$\frac{e=_\mu\overline{q+1}}{(\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}},\mu)\mapsto(\{\mathbf{for}\ y:=0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{z}};\ s[y\leftarrow\bar{q}]\}_{\vec{z}},\mu)} \quad \text{(S.FOR-II)}$$

Figure 3.3. Transition semantics

– $\delta(\sigma_1\times\ldots\times\sigma_n)=(\delta(\sigma_1),\ldots,\delta(\sigma_n))$

We also write $\delta(\vec{\sigma})$ *as an abbreviation for* $(\delta(\sigma_1),\ldots,\delta(\sigma_n))$.

Lemma 4.2. *The typing judgment* $\vdash\delta(\sigma){:}\,\sigma$ *is derivable.*

We are now ready to define the translation of imperative types. Note that the translation of a procedure type encode exactly the data-flow specified by the formal parameter modes **in** and **out** (and this is sufficient since side-effects are not allowed).

Definition 4.3. *The translations* $\sigma^\star$ *of an imperative type* σ *is defined inductively as follows:*

– $int^\star=int$

– $\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})^\star=(\vec{\sigma}^\star)^\times\to(\vec{\tau}^\star)^\times$

Moreover, if Γ *is* $x_1{:}\,\tau_1,\ldots,x_n{:}\,\tau_n$*, we define* $\Gamma^\star$ *as* $x_1{:}\,\tau_1^\star,\ldots,x_n{:}\,\tau_n^\star$.

The intuition behind the translation of imperative programs is the following: a block $\{c_1;\ldots;c_n\}_{\vec{x}}$ is translated into:

$$\mathbf{let}\ \vec{x}_1=c_1^\star\ \mathbf{in}\ldots\ \mathbf{let}\ \vec{x}_n=c_n^\star\ \mathbf{in}\ \vec{x}$$

where each $\vec{x}_i \subseteq \vec{x}$ corresponds to the "output" of command c_i and $\vec{x}$ is the output of the block. Note in particular that the same identifier is used again and again in order to simulate imperative updates. For instance, the block $\{\mathbf{inc}(x); \mathbf{inc}(x)\}_x$ is translated as:

$$\mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ x$$

Let us now give the formal definition of the translation and then present a complete example.

Definition 4.4. *For any expression e, block b, sequence s and variables $\vec{x}$, the translations $e^\star$, $b^\star$ and $(s)^\star_{\vec{x}}$ into terms of System T are defined by mutual induction as follows:*

- $\bar{n}^\star = S^n(0)$
- $y^\star = y$
- $(\mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma}; \mathbf{out}\ \vec{z}:\vec{\tau})\ \{s\}_{\vec{z}})^\star = \mathbf{fn}\ (\vec{y}:\vec{\sigma}^\star) \Rightarrow \{s\}^\star_{\vec{z}}\ [\delta(\vec{\tau}^\star)/\vec{z}]$
- $\{s\}^\star_{\vec{x}} = (s)^\star_{\vec{x}}$
- $(\varepsilon)^\star_{\vec{x}} = \vec{x}$
- $(\mathbf{var}\ y:\tau := e;\ s)^\star_{\vec{x}} = (s)^\star_{\vec{x}}[e^\star/y]$
- $(\mathbf{cst}\ y = e;\ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(y := e;\ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{inc}(y);\ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{dec}(y);\ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(p(\vec{e}\,;\vec{z}\,);\ s)^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = p^\star\ \vec{e}^{\,\star}\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = \{s_1\}^\star_{\vec{z}}\ \mathbf{in}\ (s_2)^\star_{\vec{x}}$
- $(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = \mathbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda\vec{z}\,.\{s_1\}^\star_{\vec{z}})\ \mathbf{in}\ (s_2)^\star_{\vec{x}}$

Remark 4.5. It is possible to factor out the translation of commands in the above definition. Indeed, the translation of a non-empty block $\{c; s\}_{\vec{x}}$ always follows the same pattern: $\{c; s\}^\star_{\vec{x}} = \mathbf{let}\ \vec{z} = c^\star\ \mathbf{in}\ \{s\}^\star_{\vec{x}}$ where $\vec{z}$ is a list of "output" variables of depending on the command c. The translation $c^\star$ of a command c together with its output variables $\mathcal{O}(c)$ are summarized in the following table:

c	$c^\star$	$\mathcal{O}(c)$
$y := e$	$e^\star$	y
$\mathbf{inc}(y)$	$\mathbf{succ}(y)$	y
$\mathbf{dec}(y)$	$\mathbf{pred}(y)$	y
$p(\vec{e}\,;\vec{z}\,)$	$p^\star\ \vec{e}^{\,\star}$	$\vec{z}$
$\{s\}_{\vec{z}}$	$\{s\}^\star_{\vec{z}}$	$\vec{z}$
$\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{z}}$	$\mathbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda\vec{z}\,.\{s\}^\star_{\vec{z}})$	$\vec{z}$

Lemma 4.6. *For any imperative type σ, we have $\epsilon(\sigma)^\star = \delta(\sigma^\star)$.*

The following theorem states that translation $(\ )^\star$ is type-preserving (its proof is given in Appendix C).

Theorem 4.7. *For any environments Γ and Ω, any expression e and any block $\{s\}_{\vec{x}}$ we have:*

- $\Gamma; \Omega \vdash e : \tau$ *implies* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$
- $\Gamma; \Omega, \vec{x}:\vec{\sigma} \vdash s : \mathbf{seq}$ *implies* $\Gamma^\star, \Omega^\star, \vec{x}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times$
- $\Gamma; \vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}} : \mathbf{comm}$ *implies* $\Gamma^\star, \vec{x}:\vec{\sigma}^\star \vdash \{s\}^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times$

4.2 Example: the Ackermann function

Here is an implementation of the Ackermann function as an anonymous procedure of LOOP$^\omega$. This program was actually programmed by hand, but we shall see in Section 5 how to get almost the same program from the definition of *ack* in System T.

```
proc (in m:int, n: int; out r : int) {
    proc next(in y: int; out p: int) {
        p := y;
        inc(p);
    }_p;
    var g : proc (in int; out int);
    g := next;
    for i := 1 to m {
        cst h = g;
        proc aux(in y: int; out p: int) {
            h(1, p);
            for j := 1 to y {
                h(p; p);
            }_p;
        }_p;
        g := aux;
    }_g;
    g(n, r);
}_r;
```

By applying translation $(\ )^\star$, we obtain the following anonymous function. For clarity, we use Standard ML derived forms [Milner et al., 1997] for declaring several functions (keyword **fun**) and values (keyword **val**) within a unique **let**.

```
fn (m: int, n: int) ⇒ let
    fun next(y: int) = let
        val p = y
        val p = succ(p)
    in p end
    val g = next
    val g = rec(m, g, λ i. λ g.let
                val h = g
                fun aux(y: int) = let
                    val p = h S(0)
                    val p = rec(y, p, λ j. λ p.let
                                val p = h p
                            in p end)
                in p end
                val g = aux
            in g end)
    val r = g n
in r end
```

Note that although we could prove by hand that this functional program is equivalent to the term *ack* given in Section 2.1, we shall see that this property is mostly an application of the main theorem of Section 5.

4.3 Simulation theorem

Let us prove the main theorem which states that for any block $\{s\}_{\vec{x}}$, the evaluation of $\{s\}_{\vec{x}}$ runs in lock-step with the reduction of $\{s\}^\star_{\vec{x}}$. We first need some preliminary substitution lemmas:

Notation 4.8. *If μ is a store and $\vec{x} \subset dom(\mu)$, we write $\mu(\vec{x})$ for $(\mu(x_1), ..., \mu(x_n))$. Moreover, if $\vec{v} = (v_1, ..., v_k)$ we write $\vec{v}^\star$ for the tuple $(v_1^\star, ..., v_k^\star)$.*

Lemma 4.9. *For any block $\{s\}_{\vec{z}}$ such that $\Gamma; \vec{z} : \vec{\tau} \vdash \{s\}_{\vec{z}} : \mathbf{comm}$, the following equality holds: $\{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}] = \{s[\vec{z} \hookleftarrow \vec{r}]\}^\star_{\vec{r}}[\delta(\vec{\tau})/\vec{r}]$.*

Lemma 4.10. *For any command c, any read-only variable x and any closed imperative value w, we have $(c[\vec{x} \leftarrow \vec{w}])^\star = c^\star[\vec{w}^\star/\vec{x}]$.*

Lemma 4.11. *If $e =_\mu w$ then either $e = w$ or e is some variable x_i with $\mu(x_i) = w$ and then $c^\star[\mu(\vec{x})^\star/\vec{x}] = c^\star[w^\star/x_i][\mu(\vec{x})^\star/\vec{x}]$.*

Theorem 4.12. *For any well-typed state (s, μ), if $\vec{x} = dom(\mu)$ and $\vec{z} \subseteq \vec{x}$ we have:*

$$(s, \mu) \mapsto (s', \mu') \text{ implies } (s)^\star_{\vec{z}}[\mu(\vec{x})^\star/\vec{x}] \rightsquigarrow (s')^\star_{\vec{z}}[\mu'(\vec{x})^\star/\vec{x}]$$

Proof. By induction on the derivation of $(s, \mu) \mapsto (s', \mu')$. For brevity, we write $\vec{v}$ for $\mu(\vec{x})$ and $\vec{v}\,'$ for $\mu'(\vec{x})$.

- (S.BLOCK)

$$\frac{(s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \rightsquigarrow (s')^\star_{\vec{z}}[\vec{v}\,'^\star/\vec{x}]}{\{s\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]) \rightsquigarrow \{s'\}^\star_{\vec{z}}[\vec{v}\,'^\star/\vec{x}]}$$

  Indeed:

$$\begin{aligned}
&\{s\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \\
&\quad = (s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \\
&\quad \rightsquigarrow (s')^\star_{\vec{z}}[\vec{v}\,'^\star/\vec{x}] \text{ by induction hypothesis (since } \vec{z} \subseteq \vec{x}) \\
&\quad = \{s'\}^\star_{\vec{z}}[\vec{v}\,'^\star/\vec{x}]
\end{aligned}$$

- (S.SEQ-I)

$$(\{\}_{\vec{y}}; s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \rightsquigarrow (s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

  Indeed:

$$\begin{aligned}
&(\{\}_{\vec{y}}; s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{y} = \{\}^\star_{\vec{y}}\ \mathbf{in}\ (s)^\star_{\vec{z}})[\vec{v}^\star/\vec{x}] \\
&\quad = (\mathbf{let}\ \vec{y} = \vec{y}\ \mathbf{in}\ (s)_{\vec{z}})^\star[\vec{v}^\star/\vec{x}] \\
&\quad \rightsquigarrow (s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]
\end{aligned}$$

- (S.SEQ-II)

$$\frac{c^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow c'^\star[\vec{v}\,'^\star/\vec{x}]}{(c; s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \rightsquigarrow (c'; s)^\star_{\vec{z}}[\vec{v}\,'^\star/\vec{x}]}$$

  Indeed, if $\vec{r} = \mathcal{O}(c)$ as defined in Remark 4.5, we have:

$$
\begin{aligned}
&(c; s)^\star_{\vec z}[\vec v^{\,\star}/\vec x]\\
&\quad = (\textbf{let } \vec r = c^\star \textbf{ in } (s)^\star_{\vec z})[\vec v^{\,\star}/\vec x]\\
&\quad = (\textbf{let } \vec r = c^\star[\vec v^{\,\star}/\vec x] \textbf{ in } (s)^\star_{\vec z})[\vec v^{\,\star}/\vec x]\\
&\quad \rightsquigarrow (\textbf{let } \vec r = c'^\star[\vec v\,'^\star/\vec x] \textbf{ in } (s)^\star_{\vec z})[\vec v^{\,\star}/\vec x] \text{ by induction hypothesis}\\
&\quad = (\textbf{let } \vec r = c'^\star[\vec v\,'^\star/\vec x] \textbf{ in } (s)^\star_{\vec z})[\vec v\,'^\star/\vec x] \text{ since } \mu^\star(y) = \mu'^\star(y) \text{ for any } y \notin \vec r\\
&\quad = (\textbf{let } \vec r = c'^\star \textbf{ in } (s)^\star_{\vec z})[\vec v\,'^\star/\vec x]\\
&\quad = (c'; s)^\star_{\vec z}[\vec v\,'^\star/\vec x]
\end{aligned}
$$

- (S.VAR-I)

$$\frac{(s)^\star_{\vec z}[v^\star/\vec x\,, w^\star/y] \rightsquigarrow (\varepsilon)^\star_{\vec z}[v'^\star/\vec x\,, w'^\star/y]}{(\textbf{var } y{:}\,\tau{:=}e;\ s)^\star_{\vec z}[\vec v^{\,\star}/\vec x] \rightsquigarrow (\varepsilon)^\star_{\vec z}[\vec v\,'^\star/\vec x]}$$

  Since $e =_\mu w$, by lemma 4.11:

$$
\begin{aligned}
&(\textbf{var } y{:}\,\tau{:=}e;\ s)^\star_{\vec z}[\vec v^{\,\star}/\vec x]\\
&\quad = ((\varepsilon)^\star_{\vec z}[e^\star/y])[\vec v^{\,\star}/\vec x]\\
&\quad = ((s)^\star_{\vec z}[w^\star/y])[\vec v^{\,\star}/\vec x]\\
&\quad \rightsquigarrow (\varepsilon)^\star_{\vec z}[w^\star/y'][\vec v\,'^\star/\vec x] \text{ by induction hypothesis (since } \vec z \subseteq \vec x \subseteq \vec x\,, y)\\
&\quad = ((\varepsilon)^\star_{\vec z}[\vec v\,'^\star/\vec x] \text{ since } y \notin \vec z
\end{aligned}
$$

- (S.VAR-II)

$$\frac{(s)^\star_{\vec z}[v^\star/\vec x\,, w^\star/y] \rightsquigarrow (s')^\star_{\vec z}[v'^\star/\vec x\,, w'^\star/y]}{(\textbf{var } y{:}\,\tau{:=}e;\ s)^\star_{\vec z}[\vec v^{\,\star}/\vec x] \rightsquigarrow (\textbf{var } y{:}\,\tau{:=}w';\ s')^\star_{\vec z}[\vec v\,'^\star/\vec x]}$$

  Since $e =_\mu w$, by lemma 4.11:

$$
\begin{aligned}
&(\textbf{var } y{:}\,\tau{:=}e;\ s)^\star_{\vec z}[\vec v^{\,\star}/\vec x]\\
&\quad = ((s)^\star_{\vec z}[e^\star/y])[\vec v^{\,\star}/\vec x]\\
&\quad = ((s)^\star_{\vec z}[w^\star/y])[\vec v^{\,\star}/\vec x]\\
&\quad \rightsquigarrow ((s')^\star_{\vec z}[w'^\star/y])[\vec v\,'^\star/\vec x] \text{ by induction hypothesis (since } \vec z \subseteq \vec x \subseteq \{\vec x\,, y\})\\
&\quad = (\textbf{var } y{:}\,\tau{:=}w';\ s')^\star_{\vec z}[\vec v\,'^\star/\vec x]
\end{aligned}
$$

- (S.ASSIGN)

$$(y := e; s)^\star_{\vec z}[\vec v^{\,\star}/\vec x] \rightsquigarrow (s)^\star_{\vec z}[\vec v\,'^\star/\vec x]$$

  with $\vec v\,' = \mu[y \leftarrow w](\vec x)$. Since $e =_\mu w$, by lemma 4.11:

$$
\begin{aligned}
&(y{:=}e; s)^\star_{\vec z}[\vec v^{\,\star}/\vec x]\\
&\quad = (\textbf{let } y = e^\star \textbf{ in } (s)^\star_{\vec z})[\vec v^{\,\star}/\vec x]\\
&\quad = (\textbf{let } y = w^\star \textbf{ in } (s)^\star_{\vec z})[\vec v^{\,\star}/\vec x]\\
&\quad \rightsquigarrow (s)^\star_{\vec z}[w^\star/y][\vec v^{\,\star}/\vec x]\\
&\quad = (s)^\star_{\vec z}[\vec v\,'^\star/\vec x]
\end{aligned}
$$

- (S.INC)

$$(\textbf{inc}(y))^\star[\vec v^{\,\star}/\vec x] \rightsquigarrow \overline{q+1}^\star[\vec v^{\,\star}/\vec x]$$

  Indeed, since $\mu(y) = \bar q$:

$$
\begin{aligned}
&(\textbf{inc}(y))^\star[\vec v^{\,\star}/\vec x]\\
&\quad = \textbf{succ}(y)[\vec v^{\,\star}/\vec x]\\
&\quad = \textbf{succ}(S^q(0))\\
&\quad \rightsquigarrow (S^{q+1}(0)))\\
&\quad = \overline{q+1}^\star
\end{aligned}
$$

- (S.DEC) This case is similar to (S.INC).

- (S.CALL)

$$p(\vec{e}\,,\vec{r}\,)^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow \{s[\vec{z} \hookleftarrow \vec{r}\,][\vec{y} \leftarrow \vec{w}]\}^\star_{\vec{r}}[\vec{v}\,'^\star/\vec{x}]$$

where $\vec{v}\,' = \mu'(\vec{x}) = \mu[\vec{r} \leftarrow \epsilon(\vec{\tau})](\vec{x})$. By Lemma 4.11, since $\vec{e} =_\mu \vec{w}$ and $p =_\mu \mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma}\,;\mathbf{out}\ \vec{z}:\vec{\tau}\,)\,\{s\}_{\vec{z}}$:

$$\begin{aligned}
&p(\vec{e}\,,\vec{r}\,)^\star[\vec{v}^\star/\vec{x}]\\
&\quad = (p^\star\ \vec{e}^{\,\star})[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma}\,;\mathbf{out}\ \vec{z}:\vec{\tau}\,)\,\{s\}_{\vec{z}})^\star\ \vec{w}^\star)[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{fn}\ (\vec{y}:\vec{\sigma}^\star) \Rightarrow \{s\}^\star_{\vec{z}}[\delta(\vec{\tau}^\star)/\vec{z}])\ \vec{w}^\star)[\vec{v}^\star/\vec{x}]\\
&\quad \rightsquigarrow \{s\}^\star_{\vec{z}}[\delta(\vec{\tau}^\star)/\vec{z}][\vec{w}^\star/\vec{y}][\vec{v}^\star/\vec{x}]\\
&\quad = \{s[\vec{z} \hookleftarrow \vec{r}\,]\}^\star_{\vec{r}}[\delta(\vec{\tau}^\star)/\vec{r}][\vec{w}^\star/\vec{y}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.9}\\
&\quad = \{s[\vec{z} \hookleftarrow \vec{r}\,]\}^\star_{\vec{r}}[\vec{w}^\star/\vec{y}][\epsilon(\vec{\tau})^\star/\vec{r}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.6}\\
&\quad = \{s[\vec{z} \hookleftarrow \vec{r}\,][\vec{y} \leftarrow \vec{w}]\}^\star_{\vec{r}}[\epsilon(\vec{\tau})^\star/\vec{r}][\vec{v}^\star/\vec{x}] \text{ by Lemma 4.10}\\
&\quad = \{s[\vec{z} \hookleftarrow \vec{r}\,][\vec{y} \leftarrow \vec{w}]\}^\star_{\vec{r}}[\vec{v}\,'^\star/\vec{x}] \text{ since } \vec{r} \subseteq \vec{x}\,.
\end{aligned}$$

- (S.CST)

$$(\mathbf{cst}\ y = e;\ s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \rightsquigarrow (s[y \leftarrow w])^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e =_\mu w$, by Lemma 4.11:

$$\begin{aligned}
&(\mathbf{cst}\ y = e;\ s)^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{let}\ y = w^\star\ \mathbf{in}\ (s)^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad \rightsquigarrow ((s)^\star_{\vec{z}}[w^\star/y])[\vec{v}^\star/\vec{x}]\\
&\quad = (s[y \leftarrow w])^\star_{\vec{z}}[\vec{v}^\star/\vec{x}] \text{ by Lemma 4.10}
\end{aligned}$$

- (S.FOR-I)

$$(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}] \rightsquigarrow \{\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e =_\mu 0$, by Lemma 4.11:

$$\begin{aligned}
&(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\\
&\quad = \mathbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad = \mathbf{rec}(0, \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad \rightsquigarrow \vec{z}\,[\vec{v}^\star/\vec{x}]\\
&\quad = \{\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]
\end{aligned}$$

- (S.FOR-II)

$$(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}] \rightsquigarrow \{\mathbf{for}\ y := 0\ \mathbf{until}\ \bar{q}\ \ \{s\}_{\vec{z}}; s[y \leftarrow \bar{q}\,]\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]$$

Since $e =_\mu \overline{q+1}$, by Lemma 4.11:

$$\begin{aligned}
&(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{z}})^\star[\vec{v}^\star/\vec{x}]\\
&\quad = \mathbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad = \mathbf{rec}(S^{q+1}(0), \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad \rightsquigarrow ((\lambda \vec{z}\,.\{s\}^\star_{\vec{z}}[S^q(0)/y])\ \mathbf{rec}(S^q(0), \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{z}}))[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{let}\ \vec{z} = \mathbf{rec}(S^q(0), \vec{z}\,, \lambda y.\lambda \vec{z}\,.\{s\}^\star_{\vec{x}})\ \mathbf{in}\ \{s[y \leftarrow \bar{q}\,]\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad = (\mathbf{let}\ \vec{z} = (\mathbf{for}\ y := 0\ \mathbf{until}\ \bar{q}\ \ \{s\}_{\vec{z}})^\star\ \mathbf{in}\ \{s[y \leftarrow \bar{q}\,]\}^\star_{\vec{z}})[\vec{v}^\star/\vec{x}]\\
&\quad = \{\mathbf{for}\ y := 0\ \mathbf{until}\ \bar{q}\ \ \{s\}_{\vec{z}}; s[y \leftarrow \bar{q}\,]\}^\star_{\vec{z}}[\vec{v}^\star/\vec{x}]
\end{aligned}$$

□

Corollary 4.13. *All programs of* LOOP$^\omega$ *are terminating.*

Proof. Indeed, since System T is strongly normalizing (see [Girard et al., 1989] for instance). □

Let us now focus on the imperative counterpart of the hierarchy of fragment T_n of Gödel System T. We define first the level of an imperative type and then the "iteration rank" of a LOOP$^\omega$ program as follows:

Definition 4.14. *The level $\ell(\tau)$ of an imperative type τ is defined inductively by:*

- $\ell(int) = 0$
- $\ell(\mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma}; \mathbf{out}\ \vec{z}:\vec{\tau})\,\{s\}_{\vec{z}}) = \max(\ell(\vec{\sigma})+1, \ell(\vec{\tau}))$
  where $\ell(\vec{\tau}) = \max(\ell(\tau_1), ..., \ell(\tau_n))$.

Definition 4.15. *We call "iteration rank" of a* Loop$^\omega$ *program p the maximum level of the types of mutable variables which annotate a loop of p.*

Definition 4.16. *The fragment* Loop^n *of* Loop$^\omega$ *is defined as the set of programs with iteration rank less than n.*

Lemma 4.17. *For any imperative types $\vec{\tau}$, $\partial((\vec{\tau}^\star)^\times) = \ell(\vec{\tau})$.*

Proposition 4.18. *For any* Loop$^\omega$ *program p, the iteration rank of p is the same as the recursion rank of $p^\star$.*

Proof. Indeed, since for any loop which occurs in p with mutable variables $\vec{z}:\vec{\tau}$ we have:

$$(\mathbf{for}\ y := 0\ \mathbf{until}\ e\,\{s\}_{\vec{z}})^\star = \mathbf{rec}(e^\star, \vec{z}, \lambda y.\lambda\vec{z}.\{s\}^\star_{\vec{z}})$$

where this recursor has type $(\vec{\tau}^\star)^\times$ and $\partial((\vec{\tau}^\star)^\times) = \ell(\vec{\tau})$ by lemma 4.17. □

Remark 4.19. A function computable in the fragment Loop^0 is thus computable in T_0, hence it is primitive recursive. Consequently, adding only higher-order procedures (without procedural variables) to the Loop language does not increase its expressive power.

In the next section, we shall be interested in Loop$^\omega$ programs which are translated into terms which contain no pairing and no pattern matching. The following lemma is straightforward (by definition of the translation).

Definition 4.20. *We call "singular" a* Loop$^\omega$ *program in which all procedures have exactly one* **in** *parameter and one* **out** *parameter and all blocks are annotated by exactly one mutable variable.*

Lemma 4.21. *For any singular* Loop$^\omega$ *program p, the term $p^\star$ contains no product types (no tuples and no pattern matching).*

5 Expressiveness

In this section, we show how to represent any functional of Gödel's system T by some Loop$^\omega$ program. However, since there is no direct counterpart to pattern-matching in Loop$^\omega$, we consider in this section a variant of System T with explicit pairing function and projections. On the other hand, it is straightforward to encode a (lazy) pair as an anonymous procedure with no input parameter (a thunk).

Definition 5.1. *We define $(\sigma * \tau)$ as $unit \to (\sigma \times \tau)$ and call $\mathcal{S}$ the set of types built inductively from int using the type constructors $\to$ and $*$.*

Definition 5.2. *For any terms $t:\sigma$, $u:\tau$ we write $<t,u>:\sigma*\tau$ as an abbreviation for the term* **fn** $() \Rightarrow (t,u)$. *Moreover, we define $\pi^1_{\sigma,\tau}:\sigma*\tau\to\sigma$ and $\pi^2_{\sigma,\tau}:\sigma*\tau\to\tau$ as the following abbreviations:*

$$\pi^1_{\sigma,\tau}(z) = \mathbf{let}\ (x,y) = z()\ \mathbf{in}\ x \qquad \textit{and} \qquad \pi^2_{\sigma,\tau}(z) = \mathbf{let}\ (x,y) = z()\ \mathbf{in}\ y$$

Lemma 5.3. *The following reduction rules are derivable (where v, w are values):*

$$\pi^1_{\sigma,\tau} <v,w> \rightsquigarrow^\star v \qquad \pi^2_{\sigma,\tau} <v,w> \rightsquigarrow^\star w$$

Definition 5.4. *For any type $\sigma \in \mathcal{S}$ the translation $\sigma^\diamond$ is defined as follows:*

- $int^\diamond = int$
- $(\sigma \to \tau)^\diamond = \mathbf{proc}\ (\mathbf{in}\ \sigma^\diamond; \mathbf{out}\ \tau^\diamond)$
- $(\sigma * \tau)^\diamond = \mathbf{proc}\ (\mathbf{out}\ \sigma^\diamond, \tau^\diamond)$

Lemma 5.5. *For any type $\sigma \in \mathcal{S}$, we have $(\sigma^\diamond)^\star = \sigma$.*

We are now ready to translate terms of System T into Loop$^\omega$ programs. More precisely, a term t of type σ is translated into a command c with exactly one free mutable r variable of type $\sigma^\diamond$ for the result (*i.e.*, $\mathcal{O}(c) = r$). For simplicity, we shall define the translation only for terms that obey some syntactic criterion:

Definition 5.6. *We call $\mathcal{V}$ and $\mathcal{L}$ the subset of values and terms of System T defined by the following syntax:*

$$\begin{array}{rcl} v, w & ::= & x \mid S^n(0) \mid \lambda x.t \mid <t, u> \\ \\ t, u & ::= & v \mid \mathbf{succ}(v) \mid \mathbf{pred}(v) \mid \mathbf{rec}(v, u, \lambda y.\lambda z.t) \\ & & \mid (v\ w) \mid (t\ v) \mid (v\ u) \mid \pi^1_{\sigma,\tau}(v) \mid \pi^2_{\sigma,\tau}(v) \end{array}$$

Remark 5.7. Clearly, terms of $\mathcal{L}$ can be seen as terms of System T since **succ**, $\pi^1_{\sigma,\tau}$, $\pi^2_{\sigma,\tau}$ and $<t, u>$ are just macro-definitions. Conversely, any term of System T (as defined in figure 2.1) is clearly equivalent to a term of $\mathcal{L}$. Indeed, the requirements that v be a value in any sub-term of the form $\pi^1_{\sigma,\tau}(v)$, $\pi^2_{\sigma,\tau}(v)$ or $\mathbf{rec}(v, u, \lambda y.\lambda z.t)$ and that either t or u be a value in an application $(t\ u)$ are easily fulfilled by naming intermediate results when necessary (using a β-redex to simulate a **let**).

Definition 5.8. *For any value v in $\mathcal{V}$ and any term t in $\mathcal{L}$ such that $\Gamma \vdash t : \sigma$, the translation $v^\diamond$ and $t^\diamond_r$ where r is a fresh variable of type $\sigma^\diamond$ are defined by mutual induction as follows (the types of the terms are written as superscript when required):*

- $S^n(0)^\diamond = \bar{n}$
- $y^\diamond = y$
- $(\mathbf{fn}\ x : \tau \Rightarrow t^\omega)^\diamond = \mathbf{proc}\ (\mathbf{in}\ x : \tau^\diamond; \mathbf{out}\ y : \omega^\diamond)\ t^\diamond_y$
- $(<t, u>)^\diamond = \mathbf{proc}\ (\mathbf{out}\ x : \sigma^\diamond, y : \tau^\diamond)\{t^\diamond_x; u^\diamond_y\}_{x,y}$
- $(v)^\diamond_r = r := v^\diamond$
- $\mathbf{succ}(v)^\diamond_r = \{r := v^\diamond;\ \mathbf{inc}(r)\}_r$
- $\mathbf{pred}(v)^\diamond_r = \{r := v^\diamond;\ \mathbf{dec}(r)\}_r$
- $\mathbf{rec}(v, u, \lambda y.\lambda z.t)^\diamond_r = \{u^\diamond_r;\ \mathbf{for}\ y := 0\ \mathbf{until}\ v^\diamond\ \{\mathbf{cst}\ z = r;\ t^\diamond_r\}_r\}_r$
- $(v\ w)^\diamond_r = v^\diamond(w^\diamond; r)$
- $(v\ u^\tau)^\diamond_r = \{\mathbf{var}\ y : \tau^\diamond;\ u^\diamond_y;\ v^\diamond(y; r)\}_r$
- $(t^{\tau\to\omega}\ v)^\diamond_r = \{\mathbf{var}\ x : (\tau \to \omega)^\diamond;\ t^\diamond_x;\ x(v^\diamond; r)\}_r$
- $\pi^1_{\sigma,\tau}(v)^\diamond_r = \{\mathbf{var}\ y : \tau^\diamond;\ v^\diamond(r, y)\}_r$
- $\pi^2_{\sigma,\tau}(v)^\diamond_r = \{\mathbf{var}\ x : \sigma^\diamond;\ v^\diamond(x, r)\}_r$

Remark 5.9. In Definition 5.8, we tried to reach a compromise between the simplicity of the translation and the effort required to prepare the term before its translation. It is however possible to generalize the above translation to arbitrary terms of System T (without restricting them to $\mathcal{L}$). For instance, $\mathbf{rec}(e, u, \lambda z.\lambda y.t)^\diamond_r$ where e is an arbitrary terms (not necessarily a value) can be translated as follows:

$$\mathbf{rec}(e, u, \lambda z.\lambda y.t)^\diamond_r = \{u^\diamond_r;\ \mathbf{var}\ n : int;\ e^\diamond_n;\ \mathbf{for}\ y := 0\ \mathbf{until}\ n\ \{\mathbf{cst}\ z = r;\ t^\diamond_r\}_r\}_r$$

Example 5.10. Recall the definition of the Ackermann function given in Section 2:

$$ack(n,m) = (\mathbf{rec}(m, \lambda y.\mathbf{succ}(y), \lambda h.\lambda i.\lambda y.\mathbf{rec}(y, (h\ S(0)), \lambda k.\lambda j.(h\,k)))\ n)$$

By applying the translation with variable r: *int* for the result, we get the following block for $ack(n,m)^{\diamond}_{r}$:

```
{
    var g: proc (in int; out int); {
        g := proc (in y: int; out p: int) {
            p := y;
            inc(p);
        }_p;
        for i := 1 to m {
            cst h = g;
            g := proc (in y: int; out p: int) {
                h(1; p);
                for j := 1 to y {
                    cst k = p;
                    h(k; p);
                }_p;
            }_p;
        }_g;
    }_g;
    g(n; r);
}_r;
```

We obtain thus essentially the body of the imperative version of the Ackermann function given in Section 2.1 (apart from non-mandatory blocks and constant declarations).

Note that all identifiers of the source term are mapped to read-only variables (hence the introduction of a constant declaration in the translation of a recursor). This property ensures that mutable variables do not occur in the body of procedures in the resulting Loop$^{\omega}$ program: the only mutable variables are fresh variables introduced during the translation.

Lemma 5.11. *Given a term $t \in \mathcal{L}$ such that $\Gamma \vdash t\colon \sigma$ and a fresh mutable variable r of type $\sigma^{\diamond}$ we have $\Gamma^{\diamond}; r\colon \sigma^{\diamond} \vdash t^{\diamond}_{r}\colon \mathbf{comm}$.*

Proof. By induction on t. □

For any term $t \in \mathcal{L}$, the fresh mutable variable r in $t^{\diamond}_{r}$ is always initialized before it is read, and thus r is not free in $(t^{\diamond}_{r})^{\star}$. This property is stated more formally by the following lemma.

Lemma 5.12. *Given a term $t \in \mathcal{L}$ such that $\Gamma \vdash t\colon \sigma$ and a fresh mutable variable r of type $\sigma^{\diamond}$ we have $r \notin FId((t^{\diamond}_{r})^{\star})$.*

Proof. By induction on t. □

Let us write $\twoheadrightarrow$ for the reflexive, transitive and contextual closure (with respect to arbitrary contexts) of the following general $\beta\pi$-reduction rule (where $t_1, ..., t_n$ are not necessarily values):

$$\mathbf{let}\ (x_1, ..., x_n) = (t_1, ..., t_n)\ \mathbf{in}\ u \mapsto u[t_1/x_1, ..., t_n/x_n]$$

As expected, the translated $(t^{\diamond}_{r})^{\star}$ of a term t requires several "administrative" reductions in order to recover the original term.

Theorem 5.13. *Given a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and a fresh mutable variable r of type $\sigma^\diamond$ we have $(t_r^\diamond)^\star \twoheadrightarrow t$.*

Proof. By mutual induction, we prove that for any value $v \in \mathcal{V}$ such that $\Gamma \vdash v : \tau$ we have $(v^\diamond)^\star \twoheadrightarrow v$ and for any term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \sigma$ and any fresh mutable variable r of type $\sigma^\diamond$ we have $(t_r^\diamond)^\star \twoheadrightarrow t$.

- $(S^n(0)^\diamond)^\star = \bar{n}^\star = S^n(0)$
- $(y^\diamond)^\star = y^\star = y$
- $((\mathbf{fn}\ x : \tau \Rightarrow t^\omega)^\diamond)^\star$
  $= (\mathbf{proc}(\mathbf{in}\ x : \tau^\diamond; \mathbf{out}\ y : \omega^\diamond)\ t_y^\diamond)^\star$
  $= \mathbf{fn}\ (x : \tau) \Rightarrow (t_y^\diamond)^\star[\delta(\omega)/y]$
  $= \mathbf{fn}\ (x : \tau) \Rightarrow (t_y^\diamond)^\star$ since $y \notin FId((t_y^\diamond)^\star)$
  $\twoheadrightarrow \mathbf{fn}\ x : \tau \Rightarrow t$ by induction hypothesis.
- $(<t, u>_r^\diamond)^\star$
  $= (\mathbf{proc}(\mathbf{out}\ x : \sigma^\diamond, y : \tau^\diamond)\{t_x^\diamond; u_y^\diamond\}_{x,y})^\star$
  $= (\mathbf{fn}\ () \Rightarrow \mathbf{let}\ x = (t_x^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ y = (u_y^\diamond)^\star\ \mathbf{in}\ (x, y))$
  $\twoheadrightarrow \mathbf{fn}\ () \Rightarrow ((t_x^\diamond)^\star, (u_y^\diamond)^\star)$
  $\twoheadrightarrow <t, u>$ by induction hypothesis.
- $((v)_r^\diamond)^\star$
  $= (r := v^\diamond)^\star$
  $= (v^\diamond)^\star$
  $\twoheadrightarrow v$ by induction hypothesis.
- $(\mathbf{succ}(v)_r^\diamond)^\star$
  $= \{r := v^\diamond; \mathbf{inc}(r)\}_r^\star$
  $= \mathbf{let}\ r = (v^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r = \mathbf{succ}(r)\ \mathbf{in}\ r$
  $\twoheadrightarrow \mathbf{let}\ r = \mathbf{succ}((v^\diamond)^\star)\ \mathbf{in}\ r$
  $\twoheadrightarrow \mathbf{succ}((v^\diamond)^\star)$
  $\twoheadrightarrow \mathbf{succ}(v)$ by induction hypothesis.
- The case of **pred** is similar to **succ**.
- $(\mathbf{rec}(v, u, \lambda y.\lambda z.t)_r^\diamond)^\star$
  $= \{u_r^\diamond;\ \mathbf{for}\ y := 0\ \mathbf{until}\ v^\diamond\ \{\mathbf{cst}\ z = r;\ t_r^\diamond\}_r\}_r^\star$
  $= \mathbf{let}\ r = (u_r^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r = \mathbf{rec}((v^\diamond)^\star, r, \lambda y.\lambda r.\mathbf{let}\ z = r\ \mathbf{in}\ (t_r^\diamond)^\star)\ \mathbf{in}\ r$
  $\twoheadrightarrow \mathbf{let}\ r = \mathbf{rec}((v^\diamond)^\star, (u_r^\diamond)^\star, \lambda y.\lambda z.\mathbf{let}\ z = z\ \mathbf{in}\ (t_r^\diamond)^\star)\ \mathbf{in}\ r$ since $r \notin FId((t_r^\diamond)^\star)$
  $\twoheadrightarrow \mathbf{rec}((v^\diamond)^\star, (u_r^\diamond)^\star, x, \lambda z.\lambda y.(t_r^\diamond)^\star)$
  $\twoheadrightarrow \mathbf{rec}(v, u, \lambda z.\lambda y.t)$ by induction hypothesis.
- $((v^{\tau\to\sigma}\ w)_r^\diamond)^\star$
  $= (v^\diamond(w^\diamond; r))^\star$
  $= (v^\diamond)^\star(w^\diamond)^\star$
  $\twoheadrightarrow (v\ w)$ by induction hypothesis.
- $((t^{\tau\to\sigma}\ v)_r^\diamond)^\star$
  $= \{\mathbf{var}\ p : (\tau \to \sigma)^\diamond;\ t_p^\diamond;\ p(v^\diamond; r)\}_r^\star$
  $= (\mathbf{let}\ p = (t_p^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r = p\ (v^\diamond)^\star\ \mathbf{in}\ r)[\delta(\tau \to \sigma)/p]$
  $= \mathbf{let}\ p = (t_p^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r = p\ (v^\diamond)^\star\ \mathbf{in}\ r$ since $p \notin FId((t_p^\diamond)^\star)$
  $\twoheadrightarrow (t_p^\diamond)^\star\ (v^\diamond)^\star$
  $\twoheadrightarrow (t\ v)$ by induction hypothesis.
- $((v^{\tau\to\sigma}\ u)_r^\diamond)^\star$

$=\{\mathbf{var}\ y{:}\tau^\diamond;\ u_y^\diamond;\ v(y;r)\}_r^\star$
$=(\mathbf{let}\ y=(u_y^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r=(v^\diamond)^\star\ y\ \mathbf{in}\ r)[\delta(\tau)/y]$
$=\mathbf{let}\ y=(u_y^\diamond)^\star\ \mathbf{in}\ \mathbf{let}\ r=(v^\diamond)^\star\ y\ \mathbf{in}\ r$ since $y\notin FId((u_y^\diamond)^\star)$
$\twoheadrightarrow(v^\diamond)^\star\ (u_y^\diamond)^\star$
$\twoheadrightarrow(v\ u)$ by induction hypothesis.

- $(\pi^1_{\sigma,\tau}(v)_r^\diamond)^\star=$
  $=\{\mathbf{var}\ y{:}\tau^\diamond;\ v^\diamond(r,y)\}_r^\star$
  $=(\mathbf{let}\ (r,y)=(v^\diamond)^\star()\ \mathbf{in}\ r)[\delta(\tau)/y]$
  $=\mathbf{let}\ (r,y)=(v^\diamond)^\star()\ \mathbf{in}\ r$ since $y\notin FId((v^\diamond)^\star)$
  $\twoheadrightarrow\mathbf{let}\ (r,y)=v()\ \mathbf{in}\ r$ by induction hypothesis.
  $=\pi^1_{\sigma,\tau}(v)$ by definition of $\pi^1_{\sigma,\tau}$.
- The case of $\pi^2_{\sigma,\tau}$ is similar to $\pi^1_{\sigma,\tau}$. □

Lemma 5.14. *For any functional type τ, $\partial(\tau^\diamond)=\ell(\tau)$.*

Proposition 5.15. *Given a term $t\in\mathcal{L}$ such that $\Gamma\vdash t{:}\sigma$ and a fresh mutable variable p of type $\sigma^\diamond$, the iteration rank of $t_p^\diamond$ is the same as the recursion rank of t.*

Proof. Indeed, any occurrence of a recursor type τ in t is translated as follows:

$$\mathbf{rec}(x,u,\lambda y.\lambda z.s)_r^\diamond=\{u_r^\diamond;\ \mathbf{for}\ y:=0\ \mathbf{until}\ x\ \{\mathbf{cst}\ z=r;\ s_r^\diamond\}_r\}_r$$

where r is a mutable variable of type $\tau^\diamond$. We can conclude since r occurs inside a loop and $\partial(\tau^\diamond)=\ell(\tau)$ by lemma 5.14. □

Remark 5.16. Note that a term with no product types (no pairs and no projections) is translated into a *singular* Loop$^\omega$ program.

6 Applications

6.1 Characterizing elementary functions

A first application of the results of this paper is extensional: we give a new characterization of the class of Csillag-Kalmar elementary recursive functions (the class $\mathcal{E}_3$ in Grzegorczyk hierarchy). In [Beckmann and Weiermann, 2000], the authors consider a variant of System T with an explicit discriminator $\mathcal{D}$ and an iterator $\mathcal{I}$ instead of a recursor and they prove the following theorem:

Theorem 6.1. [Beckmann and Weiermann, 2000] *The set of terms of System T which contain no λ-abstraction of the form $\lambda x...\mathcal{I}(t,...)$ where x occurs in t compute exactly the class of elementary recursive functions.*

It is well-known that both operators $\mathcal{D}$ and $\mathcal{I}$ are special cases of the recursor where $\mathcal{D}(n,a,b)\equiv\mathbf{rec}(n,a,\lambda i.\lambda x.b)$ and $\mathcal{I}(n,b,\lambda y.a)\equiv\mathbf{rec}(n,b,\lambda i.\lambda y.a)$ where x,i are fresh identifiers. On the imperative side, these operators correspond respectively to the standard compound statements **if** ... **then** ... **else** ... and **loop** n {...} (where there is no iteration variable available inside the body of the loop).

Given a term t, their restriction amounts to requiring that in any sub-term $\mathcal{I}(u,...)$ of t the free identifiers of u are actually free identifiers of t (input identifiers). Let us now define the imperative counterpart of this restriction.

Definition 6.2. *We call "elementary" any singular* Loop$^\omega$ *program p such that $\Gamma;r{:}\,int\vdash p{:}\,\mathbf{comm}$ and any loop has the form $\{\mathbf{var}\ n{:}\,int;\ \{s_1\}_n;\ \mathbf{loop}\ n\ \{s_2\}_z\}_z$ where the only free read-only variables of s_1 are in Γ.*

Remark 6.3. In this definition, the **loop** statement corresponds to a macro-definition for a **for** loop in which the iteration variable does not occur. Note also that **if** ... **then** ... **else** ... statements are allowed without restriction in p. In particular, we obtain that any LOOP$^\omega$ program in which any bound of loop is a read-only input variable is elementary.

As a corollary of Theorems 4.12 and 5.13 we obtain the following result:

Corollary 6.4. *The elementary singular* LOOP$^\omega$ *programs compute exactly the class of elementary recursive functions.*

Proof. Indeed, let p be and elementary singular LOOP$^\omega$ program. For any occurrence of a loop statement in p we have:

$$\begin{array}{l}\{\mathbf{var}\ n\colon \mathit{int};\ \{s_1\}_n;\ \mathbf{loop}\ n\ \{s_2\}_z\}^\star_z\\ \quad = \mathbf{let}\ n = \{s_1\}^\star_n\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{rec}(n, z, \lambda z.\lambda y.\{s_2\}^\star_z)\ \mathbf{in}\ z\\ \quad \twoheadrightarrow \mathbf{rec}(\{s_1\}^\star_n, z, \lambda y.\lambda z.\{s_2\}^\star_z) = \mathcal{I}(\{s_1\}^\star_n, z, \lambda z.\{s_2\}^\star_z)\end{array}$$

Since the only free read-only variables of $\{s_1\}^\star_n$ are input variables, the term $p^\star$ computes thus an elementary function. Conversely, for any term $t\colon\sigma$ of System T which obeys Beckmann and Weirmann's restriction, the term $t^\diamond_r$ (where r is a fresh mutable variable of type $\sigma^\diamond$) is an elementary singular LOOP$^\omega$ programs (see remark 5.9). □

Remark 6.5. If x is an identifier in $\mathbf{rec}(x, z, \lambda y.\lambda z.\{s\}^\star_z)$ there is no need for introducing the local variable n in its imperative translation and we obtain thus the following simpler form **loop** x $\{s\}_z$. As a special case, we obtain thus that any LOOP$^\omega$ program in which any bound of a loop is an input read-only variable is elementary. Actually, we conjecture that such programs are sufficient to represent any elementary function (a proof of this result would follow step by step the proof given for System T in Section 4 of [Beckmann and Weiermann, 2000]).

Remark 6.6. For simplicity, we restricted ourselves to singular LOOP$^\omega$ programs since [Beckmann and Weiermann, 2000] do not consider product types. However, since their result is extensional, any encoding of product types in System T (which does not use the recursor) would allow to generalize the above corollary to non-singular elementary LOOP$^\omega$ programs. Such an encoding is the following:

The product $\sigma\times\tau$ is defined by induction on σ and then on τ. Firstly, $\mathit{int}\times\mathit{int}$ is encoded as $\mathit{int}\to\mathit{int}$ where $\langle t,u\rangle_{\mathit{int}\times\mathit{int}} \equiv \lambda b.\mathcal{D}(b,t,u)$ and with $\pi^1_{\mathit{int}\times\mathit{int}}(p) \equiv (p\ 0)$ and $\pi^2_{\mathit{int}\times\mathit{int}}(p) \equiv (p\ S(0))$. Now $\mathit{int}\times(\sigma\to\varsigma)$ is defined as $\sigma\to(\mathit{int}\times\varsigma)$ and $(\sigma\to\varsigma)\times\tau$ as $\sigma\to(\varsigma\times\tau)$. The corresponding pairing functions and projections are summarized below. It is straightforward to prove that $\pi^1\langle t,u\rangle = t$ and $\pi^2\langle t,u\rangle = u$ for any types σ,τ (note however that the η-rule is required).

$$\begin{array}{lcllll}\langle t,u\rangle_{\mathit{int}\times(\sigma\to\varsigma)} & = & \lambda x\colon\sigma.\langle t,(u\ x)\rangle_{\mathit{int}\times\varsigma} & \langle t,u\rangle_{(\sigma\to\varsigma)\times\tau} & = & \lambda x\colon\sigma.\langle(t\ x),u\rangle_{\varsigma\times\tau}\\ \pi^1_{\mathit{int}\times(\sigma\to\varsigma)}(z) & = & \pi^1_{\mathit{int}\times\varsigma}(z\ \delta(\sigma)) & \pi^1_{(\sigma\to\varsigma)\times\tau}(z) & = & \lambda x\colon\sigma.\pi^1_{\varsigma\times\tau}(z\ x)\\ \pi^2_{\mathit{int}\times(\sigma\to\varsigma)}(z) & = & \lambda x\colon\sigma.\pi^2_{\mathit{int}\times\varsigma}(z\ x) & \pi^2_{(\sigma\to\varsigma)\times\tau}(z) & = & \pi^2_{\varsigma\times\tau}(z\ \delta(\sigma))\end{array}$$

6.2 The minimum problem

The second application is intensional and is related to the so-called *minimum problem*. In [Colson and Fredholm, 1998], the authors proved that in call-by-value System T, any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. As a consequence, they obtain the following result:

Theorem 6.7. [Colson and Fredholm, 1998] *There is no term of call-by-value System T which computes the minimum of two natural numbers with time-complexity* $O(min(n,m))$.

Remark 6.8. It is worth noticing that this property depends strongly on the reduction rules used for the recursor (and not only on the strategy). For instance, the property does not hold if we consider the following reduction rule for rec (which is derivable in call-by-name, but not in call-by-value):

$$\mathbf{rec}(S(n), v, \lambda x.\lambda y.t) \;\rightsquigarrow\; t[n/x, \mathbf{rec}(n, b, \lambda x.\lambda y.t)/y]$$

Indeed, since we consider weak reduction and since y may occurs under the scope of a λ-abstraction in t, there no reason that $\mathbf{rec}(n, b, \lambda x.\lambda y.t)$ be the next redex to contract. In fact, one can easily show that call-by-name evaluation can be simulated under call-by-value evaluation using this rule and the usual thunk-based encoding [Hatcliff and Danvy, 1997].

Note that [Colson and Fredholm, 1998] do not consider product types nor a predecessor operation. However, as a direct corollary of this theorem and the lock-step simulation, we can still obtain:

Corollary 6.9. *There is no singular* Loop$^\omega$ *program without* **dec** *which computes the minimum of two natural numbers n,m with time-complexity* $O(min(n, m))$.

In order to generalize this result to arbitrary Loop$^\omega$ programs, we need to extend first Colson and Fredholm's theorem. Although their proof technique seems to apply to product types, we did not succeed in extending it with a one-step predecessor operation. However, we present a direct proof of the result for our variant of System T in Appendix A and then we get the expected corollary.

Corollary 6.10. *There is no* Loop$^\omega$ *program which computes the minimum of two natural numbers n,m with time-complexity* $O(min(n, m))$.

Appendix A Ultimate Obstinacy

In this appendix, we show that (non-trivial) terms of System T are "ultimately obstinate" (this terminology is borrowed from [Colson, 1991]) even in the presence a one-step predecessor operation and product types. However, since the techniques developed in [Colson and Fredholm, 1998] do not seem to generalize easily to this case, we present here a direct proof of this result.

We first extend the syntax of terms and values with an infinite number of constants $\bot_i^k$ of type N and we supplement our set of reduction rules with the following rule:

$$C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$$

We denote by $\rightsquigarrow_\bot$this extended rewriting system. Intuitively, a constant $\bot_i^k$ represents a "sufficiently large" natural number (on which the predecessor can be applied at least k times). The strong normalization of reduction $\rightsquigarrow_\bot^*$ for well-typed terms is a consequence of the following lemma (since the reduction $\rightsquigarrow$ is strongly normalizing for well-typed terms).

Lemma A.1. *If* $t[\bot_1^{k_1}/x_1, ..., \bot_n^{k_n}/x_n] \rightsquigarrow u[\bot_1^{k_1}/x_1, ..., \bot_n^{k_n}/x_n]$ *for some terms* t, u *then* $t[0/x_1, ..., 0/x_n] \rightsquigarrow u[0/x_1, ..., 0/x_n]$.

Since terms now possibly contain constants $\bot_i^k$ of type N, even a well-typed term can get stuck during evaluation (the term cannot be evaluated further) although it is not yet a value. This lemma allows us to split the set of terms into constant, trivial and ultimately obstinate terms.

Lemma A.2. *Given a well-typed term* t *of type* N *with free variables* $x_1, ..., x_n$ *of type* N, *if* $t[\bot_1^0/x_1, ..., \bot_n^0/x_n] \rightsquigarrow_\bot^p v$ *where* v *is in normal form then one of the following cases holds:*

- v *is a value* $S^k(0)$ *(we say that* t *is "constant")*
- v *is a value* $S^k(\bot_i^m)$ *with* $m \leqslant p$ *(we say that* t *is "trivial")*

– *there is an evaluation context* $C[\,]$ *such that* $v = C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ *and* $m \leqslant p$
(we say that t *is "ultimately obstinate")*

Proof. By inspection of normal forms (and then by induction on the derivation to show that $m \leqslant p$). □

Definition A.3. *The S-substitution* $t[\![S(\bot_i)/\bot_i]\!]$ *is the homomorphic extension of the basic function defined as follows:*

– $\bot_i^0[\![S(\bot_i)/\bot_i]\!] = S(\bot_i^0)$
– $\bot_i^{k+1}[\![S(\bot_i)/\bot_i]\!] = \bot_i^k$

We write $t[\![S^p(\bot_i)/\bot_i]\!]$ *for the generalized S-substitution defined by induction on* p *with* $t[\![S^0(\bot_i)/\bot_i]\!] = t$ *and* $t[\![S^{p+1}(\bot_i)/\bot_i]\!] = t[\![S(\bot_i)/\bot_i]\!][\![S^p(\bot_i)/\bot_i]\!]$.

Lemma A.4. *For any terms* t, u, *if* $t \rightsquigarrow u$ *then* $t[\![S(\bot_i)/\bot_i]\!] \rightsquigarrow u[\![S(\bot_i)/\bot_i]\!]$

Proof. Check that S-substitution commutes with the reduction rules given in Figure 2.1 and with the rule $C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$. □

The following lemma (together with Lemma A.1) allows us to apply Lemma A.2 on genuine natural numbers.

Lemma A.5. *Given a well-typed term* t *with constants* $\bot_1^0, ..., \bot_n^0$ *of type* N, *if* $t \rightsquigarrow_\bot v$ *then* $t[\![S^p(\bot_1)/\bot_1, ..., S^p(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot v[\![S^p(\bot_1)/\bot_1, ..., S^p(\bot_n)/\bot_n]\!]$

Proof. Apply Lemma A.4 repeatedly p times for each $\bot_i$. □

Lemma A.6. (CONSTANT TERMS) *Given a well-typed constant term* $t\colon N$ *with free variables* $x_1, ..., x_n$ *of type* N, *there exist* $k, p \in \mathbb{N}$ *such that for any* $p_1 \geqslant 0, ..., p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, ..., S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$.

Proof. Let $t' = t[\bot_1^0/x_1, ..., \bot_n^0/x_n]$ and p, k be such that $t' \rightsquigarrow^p S^k(0)$ by Lemma A.2. Since by Lemma A.5, $t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(0)$ and since $t[S^{p_1}(\bot_1^0)/x_1, ..., S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!]$ we have $t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(0)$. We conclude by Lemma A.1 that $t[S^{p_1}(0)/x_1, ..., S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$. □

Lemma A.7. (TRIVIAL TERMS) *Given a well-typed trivial term* $t\colon N$ *with free variables* $x_1, ..., x_n$ *of type* N, *there exist* i, m, p, k *with* $1 \leqslant i \leqslant n$ *and* $m \leqslant p$ *such that for any* $p_1 \geqslant 0, ..., p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, ..., S^{p_n}(0)/x_n] \rightsquigarrow^p S^{p_i \dot{-} m + k}(0)$.

Proof. Let $t' = t[\bot_1^0/x_1, ..., \bot_n^0/x_n]$ and p, k, m be such that $t' \rightsquigarrow^p S^k(\bot_i^m)$ by Lemma A.2. By Lemma A.5, $t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(S^{p_i - m}(\bot_i^0))$ if $p_i \geqslant m$, and $t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(\bot_i^{m-p_i})$ if $p_i < m$. Since $t[S^{p_1}(\bot_1^0)/x_1, ..., S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, ..., S^{p_n}(\bot_n)/\bot_n]\!]$, we conclude by Lemma A.1 that $t[S^{p_1}(0)/x_1, ..., S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(S^{p_i \dot{-} m}(0))$. □

Remark A.8. Clearly, trivial terms compute only functions definable (with the usual mathematical notation) as $(x_1, ..., x_n) \mapsto (x_i \dot{-} m) + k$ (for some constants m, k).

Lemma A.9. *Given an evaluation context* $C[\,]$, *if* $C[\mathbf{rec}(S^k(0), v_2, v_3)]$ *is well-typed and* $k > 0$ *then* $C[\mathbf{rec}(S^k(0), v_2, v_3)] \rightsquigarrow C'[\mathbf{rec}(S^{k-1}(0), v_2, v_3)]$ *where* $C'[\,]$ *is again an evaluation context.*

Proof. Indeed, $C' = C[(v_3\ [\,]\ S^k(0))]$. □

Theorem A.10. (ULTIMATELY OBSTINATE TERMS) *Given an ultimately obstinate term* t *of type* N *with free variables* $x_1, ..., x_n$ *of type* N, *there exist* $i, m \in \mathbb{N}$ *with* $1 \leqslant i \leqslant n$ *such that for any* $p_1 \geqslant p, ..., p_n \geqslant p$, $t[S^{p_1}(0)/x_1, ..., S^{p_n}(0)/x_n]$ *reaches its normal form in at least* p_i *reductions steps.*

Proof. If $t' = t[\bot_1^0/x_1..., \bot_n^0/x_n]$, by Lemma A.2, $t' \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ for some p, m. Now, $t[S^{p_1}(\bot_1^0)/x_1..., S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1..., S^{p_n}(\bot_n)/\bot_n]\!]$ and since $p_i \geqslant m$, $t'[\![S^{p_1}(\bot_1)/\bot_1..., S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^{p_i-m}, v_2, v_3)]$ by Lemma A.5. We have then $t[S^{p_1}(0)/x_1..., S^{p_n}(0)/x_n] \rightsquigarrow^p C[\mathbf{rec}(S^{p_i-m}(0), v_2, v_3)]$ by Lemma A.1. Finally $C[\mathbf{rec}(S^{p_i-m}(0), v_2, v_3)] \rightsquigarrow^{p_i-m} C'[\mathbf{rec}(0, v_2, v_3)]$ by Lemma A.9 and we can conclude since $p \geqslant m$ and thus $p + p_i - m \geqslant p_i$. □

Appendix B Subject reduction for Loop$^\omega$

B.1 Substitution and renaming

Recall that renaming is only defined for mutable variables and that the imperative substitution is only defined for read-only variables. Since we consider commands up to α-conversion, we may assume that no variable capture can occur.

Definition B.1. *The renaming meta-operation on commands, sequences and expressions, denoted respectively by $c[x \hookleftarrow y]$, $\{s\}_{\vec{z}}[x \hookleftarrow y]$ and $e'[x \hookleftarrow y]$ is defined in Table B.1.*

$x[x \hookleftarrow y]$	$=$	y	
$z[x \hookleftarrow y]$	$=$	z	*if* $x \neq z$
$w[x \hookleftarrow y]$	$=$	w	
$\varepsilon[x \hookleftarrow y]$	$=$	ε	
$(c; s)[x \hookleftarrow y]$	$=$	$c[x \hookleftarrow y]; s[x \hookleftarrow y]$	
$(\mathbf{cst}\ x = e';\ s)[x \hookleftarrow y]$	$=$	$\mathbf{cst}\ x = (e'[x \hookleftarrow y]);\ s$	
$(\mathbf{cst}\ z = e';\ s)[x \hookleftarrow y]$	$=$	$\mathbf{cst}\ z = (e'[x \hookleftarrow y]); (s[x \hookleftarrow y])$	*if* $x, y \neq z$
$(\mathbf{var}\ x : \tau := e';\ s)[x \hookleftarrow y]$	$=$	$\mathbf{var}\ x : \tau := (e'[x \hookleftarrow y]);\ s$	
$(\mathbf{var}\ z : \tau := e';\ s)[x \hookleftarrow y]$	$=$	$\mathbf{var}\ z : \tau := (e'[x \hookleftarrow y]); (s[x \hookleftarrow y])$	*if* $x, y \neq z$
$\{s\}_{\vec{z}}[x \hookleftarrow y]$	$=$	$\{s[x \hookleftarrow y]\}_{\vec{z}[x \hookleftarrow y]}$	
$(\mathbf{for}\ x := 0\ \mathbf{until}\ e'\ \{s\}_{\vec{r}})[x \hookleftarrow y]$	$=$	$\mathbf{for}\ x := 0\ \mathbf{until}\ (e'[x \hookleftarrow y])\ \{s\}_{\vec{r}}$	
$(\mathbf{for}\ z := 0\ \mathbf{until}\ e'\ \{s\}_{\vec{r}})[x \hookleftarrow y]$	$=$	$\mathbf{for}\ z := 0\ \mathbf{until}\ (e'[x \hookleftarrow y])(\{s\}_{\vec{r}}[x \hookleftarrow y])$	*if* $x, y \neq z$
$(x := e')[x \hookleftarrow y]$	$=$	$y := (e'[x \hookleftarrow y])$	
$(z := e')[x \hookleftarrow y]$	$=$	$z := (e'[x \hookleftarrow y])$	*if* $x \neq z$
$(\mathbf{inc}(x))[x \hookleftarrow y]$	$=$	$\mathbf{inc}(y)$	
$(\mathbf{inc}(z))[x \hookleftarrow y]$	$=$	$\mathbf{inc}(z)$	*if* $x \neq z$
$(\mathbf{dec}(x))[x \hookleftarrow y]$	$=$	$\mathbf{dec}(y)$	
$(\mathbf{dec}(z))[x \hookleftarrow y]$	$=$	$\mathbf{dec}(z)$	*if* $x \neq z$
$(e'(\vec{e}\,'', \vec{y}))[x \hookleftarrow y]$	$=$	$e'[x \hookleftarrow y](\vec{e}\,''[x \hookleftarrow y], \vec{y}[x \hookleftarrow y])$	

Table B.1. *The renaming meta-operation*

Definition B.2. *The substitution meta-operation on commands, sequences and expressions, denoted respectively by $c[x \leftarrow e]$, $\{s\}_{\vec{z}}[x \leftarrow e]$ and $e'[x \leftarrow e]$ is defined in Table B.2.*

Lemma B.3. *If $\Gamma, x : \tau; \Omega \vdash c : \mathcal{T}$ and $\emptyset; \emptyset \vdash e : \tau$ then $\Gamma; \Omega \vdash c[x \leftarrow e] : \mathcal{T}$.*

Proof. Straightforward induction on c. □

Lemma B.4. *If $\Gamma; x : \tau, \Omega \vdash c : \mathcal{T}$ then $\Gamma; y : \tau, \Omega \vdash c[x \hookleftarrow y] : \mathcal{T}$.*

Proof. Straightforward induction on c. □

B.2 Preliminary lemmas

Lemma B.5. *If $\Omega \vdash \mu$ and for all $\Delta \subset \Omega$, $\emptyset; \Delta \vdash e : \tau$ and $e =_\mu w$, then we have $\emptyset; \emptyset \vdash w : \tau$.*

$x[x \leftarrow e]$	$=$	e	
$y[x \leftarrow e]$	$=$	y	*if* $x \neq y$
$\bar{q}[x \leftarrow e]$	$=$	$\bar{q}$	
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y}, x{:}\vec{\sigma}; \mathbf{out}\ \vec{z}{:}\vec{\tau})\ \{s\}_{\vec{z}})[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y}, x{:}\vec{\sigma}; \mathbf{out}\ \vec{z}{:}\vec{\tau})\ \{s\}_{\vec{z}}$	
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\vec{\sigma}; \mathbf{out}\ \vec{z}, x{:}\vec{\tau})\ \{s\}_{\vec{z}})[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\vec{\sigma}; \mathbf{out}\ \vec{z}, x{:}\vec{\tau})\ \{s\}_{\vec{z}}$	
$(\mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\vec{\sigma}; \mathbf{out}\ \vec{z}{:}\vec{\tau})\ \{s\}_{\vec{z}})[x \leftarrow e]$	$=$	$\mathbf{proc}\ (\mathbf{in}\ \vec{y}{:}\vec{\sigma}; \mathbf{out}\ \vec{z}{:}\vec{\tau})(\{s\}_{\vec{z}}[x \leftarrow e])$	*if* $x \notin \vec{y}, \vec{z}$
$\varepsilon[x \leftarrow e]$	$=$	ε	
$(c; s)[x \leftarrow e]$	$=$	$c[x \leftarrow e]; s[x \leftarrow e]$	
$(\mathbf{cst}\ x = e';\ s)[x \leftarrow e]$	$=$	$\mathbf{cst}\ x = (e'[x \leftarrow e]);\ s$	
$(\mathbf{cst}\ y = e';\ s)[x \leftarrow e]$	$=$	$\mathbf{cst}\ y = (e'[x \leftarrow e]); (s[x \leftarrow e])$	*if* $x \neq y$
$(\mathbf{var}\ x{:}\tau := e';\ s)[x \leftarrow e]$	$=$	$\mathbf{var}\ x{:}\tau := (e'[x \leftarrow e]);\ s$	
$(\mathbf{var}\ y{:}\tau := e';\ s)[x \leftarrow e]$	$=$	$\mathbf{var}\ y{:}\tau := (e'[x \leftarrow e]); (s[x \leftarrow e])$	*if* $x \neq y$
$\{s\}_{\vec{z}}[x \leftarrow e]$	$=$	$\{s[x \leftarrow e]\}_{\vec{z}}$	
$(\mathbf{for}\ x := 0\ \mathbf{until}\ e'\ \{s\}_{\vec{z}})[x \leftarrow e]$	$=$	$\mathbf{for}\ x := 0\ \mathbf{until}\ (e'[x \leftarrow e])\ \{s\}_{\vec{z}}$	
$(\mathbf{for}\ y := 0\ \mathbf{until}\ e'\ \{s\}_{\vec{z}})[x \leftarrow e]$	$=$	$\mathbf{for}\ y := 0\ \mathbf{until}\ (e'[x \leftarrow e])(\{s\}_{\vec{z}}[x \leftarrow e])$	*if* $x \neq y$
$(y := e')[x \leftarrow e]$	$=$	$y := (e'[x \leftarrow e])$	
$(\mathbf{inc}(y))[x \leftarrow e]$	$=$	$\mathbf{inc}(y)$	
$(\mathbf{dec}(y))[x \leftarrow e]$	$=$	$\mathbf{dec}(y)$	
$(e'(\vec{e}\,'', \vec{y}))[x \leftarrow e]$	$=$	$e'[x \leftarrow e](\vec{e}\,''[x \leftarrow e], \vec{y})$	

Table B.2. *The substitution meta-operation*

Proof. The case $e = w$ is trivial and if e is some variable x then by definition of $\Omega \vdash \mu$, we have $\emptyset; \emptyset \vdash \mu(x) = w : \tau$. □

Lemma B.6. *If* $\Omega \vdash \mu$*, then for all* $\Gamma \subset \Omega$*, we have* $\Gamma \vdash \mu$*.*

Proof. By definition of store typing. □

B.3 Proof of Theorem 3.18

THEOREM 3.18. For any environment Ω and any state (c, μ), we have that

- $\emptyset; \Omega \vdash (c, \mu)$ and $(c, \mu) \mapsto (c', \mu')$ implies $\emptyset; \Omega \vdash (c', \mu')$
- $\emptyset; \Omega \vdash (s, \mu)$ and $(s, \mu) \mapsto (s', \mu')$ implies $\emptyset; \Omega \vdash (s', \mu')$

and $dom(\mu) = dom(\mu')$.

Proof. By induction on the derivation of $(c, \mu) \mapsto (c', \mu')$, and then by analysis of the typing derivation.

- (S.BLOCK): we have $\Omega \vdash \mu$ and

$$\frac{\vec{z} \subset \Omega \quad \Gamma; \vec{z}{:}\vec{\sigma} \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash \{s\}_{\vec{z}} : \mathbf{comm}}$$

  By Lemma B.6 we get $\vec{z}{:}\vec{\sigma} \vdash \mu$, and by induction hypothesis on $\vec{z}{:}\vec{\sigma} \vdash (s, \mu)$, we obtain $dom(\mu) = dom(\mu')$ and $\vec{z}{:}\vec{\sigma} \vdash (s', \mu')$ which gives us $\emptyset; \Omega \vdash s' : \mathbf{seq}$ and $\Omega \vdash \mu'$. We can build the following typing derivation:

$$\frac{\vec{z} \subset \Omega \quad \Gamma; \vec{z}{:}\vec{\sigma} \vdash s' : \mathbf{seq}}{\emptyset; \Omega \vdash \{s'\}_{\vec{z}} : \mathbf{comm}}$$

- (S.SEQ-I): we have $\Omega \vdash \mu$ and

$$\frac{\dfrac{\vec{z} \subset \Omega \qquad \overline{\emptyset; \vec{z}{:}\vec{\sigma} \vdash \varepsilon : \mathbf{seq}}}{\emptyset; \Omega \vdash \{\}_{\vec{z}} : \mathbf{comm}} \qquad \emptyset; \Omega \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash \{\}_{\vec{z}}; s : \mathbf{seq}}$$

then we get $\Omega \vdash \mu$ and $\emptyset; \Omega \vdash s : \mathbf{seq}$.

- (S.SEQ-II): we have $\Omega \vdash \mu$ and
$$\frac{\emptyset; \Omega \vdash c : \mathbf{comm} \quad \emptyset; \Omega \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash c; s : \mathbf{seq}}$$
By induction hypothesis on $\Omega \vdash (c, \mu)$, we obtain $dom(\mu) = dom(\mu')$ and $\Omega \vdash (c', \mu')$ which gives us $\emptyset; \Omega \vdash c' : \mathbf{comm}$ and $\Omega \vdash \mu'$. We can build the following typing derivation:
$$\frac{\emptyset; \Omega \vdash c' : \mathbf{comm} \quad \emptyset; \Omega \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash c'; s : \mathbf{seq}}$$
- (S.VAR-I): we have $\Omega \vdash \mu$ and
$$\frac{\emptyset; \Omega \vdash e : \tau \qquad \emptyset; \Omega, y : \tau \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{var}\ y : \tau := e; s : \mathbf{seq}}$$
By Lemma B.5, $\emptyset; \Omega \vdash e : \tau$, $\Omega \vdash \mu$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w : \tau$. By definition of store typing, $\Omega \vdash \mu$ then implies $\Omega, y : \tau \vdash (\mu, y \leftarrow w)$. By induction hypothesis, since $\Omega, y : \tau \vdash (s, (\mu, y \leftarrow w))$ is derivable, we obtain $dom(\mu, y \leftarrow w) = dom(\mu', y \leftarrow w')$, that is $dom(\mu) = dom(\mu')$, and $\Omega, y : \tau \vdash (\varepsilon, (\mu', y \leftarrow w'))$ which gives us $\Omega, y : \tau \vdash \mu', y \leftarrow w'$, that is $\Omega \vdash \mu'$. We get $\emptyset; \Omega \vdash \varepsilon : \mathbf{seq}$.
- (S.VAR-II): we have $\Omega \vdash \mu$ and
$$\frac{\emptyset; \Omega \vdash e : \tau \quad \emptyset; \Omega, y : \tau \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{var}\ y : \tau := e;\ s : \mathbf{seq}}$$
By Lemma B.5, $\emptyset; \Omega \vdash e : \tau$, $\Omega \vdash \mu$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w : \tau$. By definition of store typing, $\Omega \vdash \mu$ then implies $\Omega, y : \tau \vdash (\mu, y \leftarrow w)$. By induction hypothesis, since $\emptyset; \Omega, y : \tau \vdash (s, (\mu, y \leftarrow w))$ is derivable, we obtain $dom(\mu, y \leftarrow w) = dom(\mu', y \leftarrow w')$ which implies $dom(\mu) = dom(\mu')$, and $\Omega, y : \tau \vdash (s', (\mu, y \leftarrow w'))$ which implies $\emptyset; \Omega \vdash \mathbf{var}\ y : \tau := w'\ ; s'$ and $\Omega, y : \tau \vdash (\mu', y \leftarrow w')$. This last assertion trivially implies $\Omega \vdash \mu'$ by definition of store typing, and $\emptyset; \emptyset \vdash w' : \tau$. We can then build the following typing derivation to conclude:
$$\frac{\emptyset; \Omega \vdash w' : \tau \quad \emptyset; \Omega, y : \tau \vdash s' : \mathbf{seq}}{\emptyset; \Omega \vdash \mathbf{var}\ y : \tau := w';\ s' : \mathbf{seq}}$$
- (S.ASSIGN): we have $\Omega \vdash \mu$ and
$$\frac{\dfrac{y : \tau \in \Omega \quad \emptyset; \Omega \vdash e : \tau}{\emptyset; \Omega \vdash y := e : \mathbf{comm}} \qquad \emptyset; \Omega \vdash s : \mathbf{seq}}{\emptyset; \Omega \vdash y := e; s : \mathbf{seq}}$$
then we get $\emptyset; \Omega \vdash s : \mathbf{seq}$. By Lemma B.5, we have $\emptyset; \emptyset \vdash w : \tau$, and since $y : \tau \in \Omega$ we get $\Omega \vdash \mu[y \leftarrow w]$ and $dom(\mu) = dom(\mu[y \leftarrow w])$.
- (S.INC): we have $\Omega \vdash \mu$ and
$$\frac{y : \mathbf{int} \in \Omega}{\emptyset; \Omega \vdash \mathbf{inc}(y) : \mathbf{comm}}$$
then we easily get $\Omega \vdash \mu$ and
$$\frac{y : \mathbf{int} \in \Omega \quad \emptyset; \Omega \vdash \overline{q+1} : \mathbf{int}}{\emptyset; \Omega \vdash y := \overline{q+1} : \mathbf{comm}}$$
- (S.DEC): similar to above.
- (S.CALL): we have $\Omega \vdash \mu$ and
$$\frac{\emptyset; \Omega \vdash p : \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau}) \quad \emptyset; \Omega \vdash e_i : \sigma_i \quad r_j : \tau_j \in \Omega}{\emptyset; \Omega \vdash p(\vec{e}\,, \vec{r}\,) : \mathbf{comm}}$$

By Lemma B.5, $\emptyset;\Omega\vdash e_i\colon\sigma_i$, $\Omega\vdash\mu$ and $\vec{e}=_\mu\vec{w}$ implies $\emptyset;\emptyset\vdash w_i\colon\sigma_i$. Still by Lemma B.5, $\emptyset;\Omega\vdash p\colon\mathbf{proc}(\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})$, $\Omega\vdash\mu$ and $p=_\mu\mathbf{proc}(\mathbf{in}\ \vec{y}\colon\vec{\sigma};\mathbf{out}\ \vec{z}\colon\vec{\tau})\{s\}_{\vec{z}}$ implies $\emptyset;\emptyset\vdash\mathbf{proc}(\mathbf{in}\ \vec{y}\colon\vec{\sigma};\mathbf{out}\ \vec{z}\colon\vec{\tau})\{s\}_{\vec{z}}\colon\mathbf{proc}(\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})$, that is

$$\frac{\vec{z}\neq\emptyset\qquad\emptyset;\vec{y}\colon\vec{\sigma};\vec{z}\colon\vec{\tau}\vdash s\colon\mathbf{seq}}{\emptyset;\emptyset\vdash\mathbf{proc}\ (\mathbf{in}\ \vec{y}\colon\vec{\sigma};\mathbf{out}\ \vec{z}\colon\vec{\tau})\{s\}_{\vec{z}}\colon\mathbf{proc}(\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})}$$

By Lemmas B.3 and B.4, $\vec{y}\colon\vec{\sigma};\vec{z}\colon\vec{\tau}\vdash s\colon\mathbf{comm}$ and $\emptyset;\emptyset\vdash w_i\colon\sigma_i$ implies $\emptyset;\vec{r}\colon\vec{\tau}\vdash s[\vec{y}\leftarrow\vec{w}][\vec{z}\hookleftarrow\vec{r}]\colon\mathbf{comm}$, and since $r_j\colon\tau_j\in\Omega$, the typing rule (T.COMM) gives us $\emptyset;\Omega\vdash\{s[\vec{y}\leftarrow\vec{w}][\vec{z}\hookleftarrow\vec{r}]\}_{\vec{r}}\colon\mathbf{comm}$. By Lemma 3.11, we have $\emptyset;\emptyset\vdash\epsilon(\tau_j)\colon\tau_j$, and since $r_j\colon\tau_j\in\Omega$, we obtain $\Omega\vdash\mu[\vec{r}\leftarrow\epsilon(\vec{\tau})]$.

- (S.CST): we have $\Omega\vdash\mu$ and

$$\frac{\emptyset;\Omega\vdash e\colon\tau\quad y\colon\tau;\Omega\vdash s\colon\mathbf{seq}}{\emptyset;\Omega\vdash\mathbf{cst}\ y=e;\ s\colon\mathbf{seq}}$$

  We trivially have $\Omega\vdash\mu$. By Lemma B.5, $\emptyset;\Omega\vdash e\colon\tau$, $\Omega\vdash\mu$ and $e=_\mu w$ implies $\emptyset;\emptyset\vdash w\colon\tau$. By Lemma B.3, $y\colon\tau;\Omega\vdash s\colon\mathbf{seq}$ and $\emptyset;\emptyset\vdash w\colon\tau$ implies $\emptyset;\Omega\vdash s[y\leftarrow w]\colon\mathbf{seq}$.

- (S.FOR-I): we have $\Omega\vdash\mu$ and

$$\frac{\vec{z}\subset\Omega\qquad\emptyset;\Omega\vdash e\colon int\quad y\colon int;\vec{z}\colon\vec{\sigma}\vdash s\colon\mathbf{seq}}{\emptyset;\Omega\vdash\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}}\colon\mathbf{comm}}$$

  We clearly have $\Omega\vdash\mu$ and $\emptyset;\Omega\vdash\{\}_{\vec{z}}\colon\mathbf{comm}$.

- (S.FOR-II): we have $\Omega\vdash\mu$ and

$$\frac{\vec{z}\subset\Omega\qquad\emptyset;\Omega\vdash e\colon int\quad y\colon int;\vec{z}\colon\vec{\sigma}\vdash s\colon\mathbf{seq}}{\emptyset;\Omega\vdash\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{z}}\colon\mathbf{comm}}$$

  By Lemmas B.3, $y\colon int;\vec{z}\colon\vec{\tau}\vdash s\colon\mathbf{seq}$ and $\emptyset;\emptyset\vdash\overline{q+1}\colon int$ implies $\emptyset;\vec{z}\colon\vec{\tau}\vdash s[y\leftarrow\overline{q+1}]\colon\mathbf{seq}$. We clearly have $\Omega\vdash\mu$ and the following rule is easily derivable:

$$\frac{\vec{z}\subset\Omega\qquad\dfrac{\dfrac{\vec{z}\subset\vec{z}\colon\vec{\sigma}\quad\emptyset;\Omega\vdash\bar{q}\colon int\quad y\colon int;\vec{z}\colon\vec{\sigma}\vdash s\colon\mathbf{seq}}{\emptyset;\vec{z}\colon\vec{\sigma}\vdash\mathbf{for}\ y:=0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{z}}\colon\mathbf{comm}}\qquad\dfrac{}{\emptyset;\vec{z}\colon\vec{\sigma}\vdash s[y\leftarrow\overline{q+1}]\colon\mathbf{seq}}}{\emptyset;\vec{z}\colon\vec{\sigma}\vdash\mathbf{for}\ y:=0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{z}};\ s[y\leftarrow\overline{q+1}]\colon\mathbf{seq}}}{\emptyset;\Omega\vdash\{\mathbf{for}\ y:=0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{z}};\ s[y\leftarrow\overline{q+1}]\}_{\vec{z}}\colon\mathbf{comm}}$$

□

Appendix C Translation $(\ )^\star$ is type-preserving

C.1 Preliminary lemmas

Lemma C.1. *For any environment Γ, variable y, and terms t and e of System T, if $\Gamma,y\colon\tau\vdash t\colon\tau'$ and $\Gamma,\Omega\vdash e\colon\tau$ then $\Gamma,\Omega\vdash t[e/y]\colon\tau'$.*

Proof. By induction on t. □

Lemma C.2. *For any environment Γ, variable y, type τ' and term t of System T, if $y\notin FId(t)$ then $\Gamma,y\colon\tau'\vdash t\colon\tau$ implies $\Gamma\vdash t\colon\tau$.*

Proof. By induction on the typing judgment of t. □

Notation C.3. *For any environment $\Gamma=(x_1\colon\tau_1,\ldots,x_n\colon\tau_n)$, we write $\Gamma^\times$ for $\tau_1\times\ldots\times\tau_n$.*

Lemma C.4. *For any environment Γ, $dom(\Gamma)=dom(\Gamma^\star)$ and $x\colon\tau\in\Gamma$ implies $x\colon\tau^\star\in\Gamma^\star$.*

C.2 Proof of Theorem 4.7

THEOREM 4.7. *For any environments Γ and Ω, any expression e and any sequence s and any block $\{s\}_{\vec{x}}$ we have:*

- $\Gamma;\Omega \vdash e:\tau$ *implies* $\Gamma^\star,\Omega^\star \vdash e^\star:\tau^\star$
- $\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash s:\mathbf{seq}$ implies $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{x}}:(\vec{\sigma}^\star)^\times$
- $\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}}:\mathbf{comm}$ *implies* $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star \vdash \{s\}^\star_{\vec{x}}:\vec{\sigma}^\times$

Proof. By induction on e, s, and $\{s\}_{\vec{x}}$.

- $e=\bar{n}$. We have:

  $$\Gamma;\Omega \vdash \bar{n}:int$$

  and then

  $$\Gamma^\star,\Omega^\star \vdash S^n(0):int$$

  is clearly derivable.

- $e=y$. We have:

  $$\frac{y:\tau \in \Gamma,\Omega}{\Gamma;\Omega \vdash y:\tau}$$

  and then

  $$\Gamma^\star,\Omega^\star \vdash y:\tau^\star$$

  since by Lemma C.4, $y:\tau \in \Gamma,\Omega$ implies $y:\tau^\star \in \Gamma^\star,\Omega^\star$.

- $e=\mathbf{proc}\ (\mathbf{in}\ \vec{y}:\vec{\sigma};\mathbf{out}\ \vec{z}:\vec{\tau})\,\{s\}_{\vec{z}}$. We have:

  $$\frac{\vec{z}\neq\emptyset \qquad \Gamma,\vec{y}:\vec{\sigma};\vec{z}:\vec{\tau} \vdash s:\mathbf{seq}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\vec{y}:\mathbf{in}\ \vec{\sigma};\vec{z}:\mathbf{out}\ \vec{\tau})\{s\}_{\vec{z}}:\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})}$$

  Since $\Gamma,\vec{y}:\vec{\sigma};\vec{z}:\vec{\tau} \vdash \{s\}_{\vec{z}}:\mathbf{comm}$ is derivable, we obtain by induction hypothesis $\Gamma^\star,\vec{y}:\vec{\sigma}^\star,\vec{z}:\vec{\tau}^\star \vdash \{s\}^\star_{\vec{z}}:(\vec{\tau}^\star)^\times$, and by Lemma 4.2 $\vdash \delta(\tau_i):\tau_i^\star$. By Lemma C.1, we then get $\Gamma^\star,\vec{y}:\vec{\sigma}^\star,\Omega^\star \vdash \{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}]:(\vec{\tau}^\star)^\times$. We build the following typing derivation:

  $$\frac{\Gamma^\star,\vec{y}:\vec{\sigma}^\star,\Omega^\star \vdash \{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}]:(\vec{\tau}^\star)^\times}{\Gamma^\star,\Omega^\star \vdash \mathbf{fn}\ (\vec{y}:\vec{\sigma}^\star) \Rightarrow \{s\}^\star_{\vec{z}}[\delta(\vec{\tau})/\vec{z}]:(\vec{\sigma}^\star)^\times \to (\vec{\tau}^\star)^\times}$$

- $\{s\}_{\vec{x}}$. We have:

  $$\frac{\vec{x}\subset\Omega,\vec{x}:\vec{\sigma} \qquad \Gamma;\vec{x}:\vec{\sigma} \vdash s:\mathbf{seq}}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}}:\mathbf{comm}}$$

  Since $\Gamma;\vec{x}:\vec{\sigma} \vdash s:\mathbf{seq}$ is derivable, we obtain by induction hypothesis the required typing derivation $\Gamma^\star;\vec{x}:\vec{\sigma}^\star \vdash (s)^\star_{\vec{x}}:(\vec{\sigma}^\star)^\times$.

- $s=\varepsilon$. We have:

  $$\frac{}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \varepsilon:\mathbf{seq}}$$

  and then $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star \vdash \vec{x}:(\vec{\sigma}^\star)^\times$ by Lemma C.4.

- $s=\mathbf{var}\ y:\tau:=e;\ s'$ with $s'\neq\varepsilon$. We have:

  $$\frac{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash e:\tau \quad \Gamma;\Omega,\vec{x}:\vec{\sigma},y:\tau \vdash s':\mathbf{seq}}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \mathbf{var}\ y:\tau:=e;\ s':\mathbf{seq}}$$

  By induction hypothesis, we obtain $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star,y:\tau^\star \vdash (s')^\star_{\vec{x}}:(\vec{\sigma}^\star)^\times$ and $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star \vdash e^\star:\tau^\star$. By Lemma C.1, we obtain $\Gamma^\star,\Omega^\star,\vec{x}:\vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}[e^\star/y]:(\vec{\sigma}^\star)^\times$.

- $s = \mathbf{cst}\ y = e;\ s'$. We have:

$$\frac{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash e: \tau \quad \Gamma, y: \tau; \Omega, \vec{x}: \vec{\sigma} \vdash s': \mathbf{seq}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash \mathbf{cst}\ y = e;\ s': \mathbf{seq}}$$

By induction hypothesis, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star, y: \tau^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times$ and $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash e^\star: \tau^\star$. We build then the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash e^\star: \tau^\star \qquad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star, y: \tau^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}$$

- $s = y := e;\ s'$. We have:

$$\frac{\dfrac{y: \tau \in \Omega, \vec{x}: \vec{\sigma} \quad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash e: \tau}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash y := e: \mathbf{comm}} \qquad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash s': \mathbf{seq}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash y := e,\ s': \mathbf{seq}}$$

By induction hypothesis on $\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash e: \tau$, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash e^\star: \tau^\star$ and by induction hypothesis on $\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash s': \mathbf{seq}$, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times$. We build then the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash e^\star: \tau^\star \qquad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}$$

- $s = \mathbf{inc}(y);\ s'$. We have:

$$\frac{\dfrac{y: int \in \Omega, \vec{x}: \vec{\sigma}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash \mathbf{inc}(y): \mathbf{comm}} \qquad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash s': \mathbf{seq}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash \mathbf{inc}(y);\ s': \mathbf{seq}}$$

We remark that since $y: int \in \Omega, \vec{x}: \vec{\sigma}$, by Lemma C.4, we get $(y: int) \in \Omega^\star, \vec{x}: \vec{\sigma}^\star$ and we have $\Omega^\star, \vec{x}: \vec{\sigma}^\star$, $y: int = \Omega^\star, \vec{x}: \vec{\sigma}^\star$. By induction hypothesis, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times$. We can then build the following typing derivation :

$$\frac{\dfrac{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash y: int}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{succ}(y): int} \qquad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}$$

- $s = \mathbf{dec}(y);\ s'$. Similar to the previous case.
- $s = p(\vec{e}\,; \vec{z}\,);\ s'$. We have:

$$\frac{\dfrac{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash p: \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\tau}\,') \quad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash e_i: \tau_i \quad z_j: \tau'_j \in \Omega, \vec{x}: \vec{\sigma}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash p(\vec{e}\,; \vec{z}\,): \mathbf{comm}} \qquad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash s': \mathbf{seq}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash p(\vec{e}\,; \vec{z}\,);\ s': \mathbf{seq}}$$

By induction hypothesis, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash e_i^\star: \tau_i^\star$ and $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash p^\star: (\vec{\tau}^\star)^\times \to (\vec{\tau}\,'^\star)^\times$ and $\Gamma^\star, \Omega^\star$, $\vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times$. Note that since $z_j: \tau'_j \in \Omega, \vec{x}: \vec{\sigma}^\star$, by Lemma C.4, we get $\vec{z}: \vec{\tau}\,'^\star \subset \Omega^\star, \vec{x}: \vec{\sigma}^\star$ and then $\Omega^\star, \vec{x}: \vec{\sigma}^\star, \vec{z}: (\vec{\tau}\,'^\star)^\times = \Omega^\star, \vec{x}: \vec{\sigma}^\star$. We can then build the following typing derivation :

$$\frac{\dfrac{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash p^\star: (\vec{\tau}^\star)^\times \to (\vec{\tau}\,'^\star)^\times \quad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \vec{e}^\star: \vec{\tau}^\star}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash p^\star\ \vec{e}^\star: (\vec{\tau}\,'^\star)^\times} \qquad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{let}\ \vec{z} = p^\star\ \vec{e}^\star\ \mathbf{in}\ (s')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}$$

- $s = \{s'\}^\star_{\vec{z}};\ s''$. We have:

$$\frac{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash \{s'\}_{\vec{z}}: \mathbf{comm} \quad \Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash s'': \mathbf{seq}}{\Gamma; \Omega, \vec{x}: \vec{\sigma} \vdash \{s'\}^\star_{\vec{z}};\ s'': \mathbf{seq}}$$

By induction hypothesis, we obtain $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \{s'\}^\star_{\vec{z}}: (\vec{\tau}^\star)^\times$ where $\vec{z}: \vec{\tau} \subset \Omega, \vec{x}: \vec{\sigma}$ (and then $\Omega^\star, \vec{x}: \vec{\sigma}^\star$, $\vec{z}: (\vec{\tau}^\star)^\times = \Omega^\star, \vec{x}: \vec{\sigma}^\star$) and $\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s'')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times$. We can then build the following typing derivation :

$$\frac{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \{s'\}^\star_{\vec{z}}: (\vec{\tau}^\star)^\times \qquad \Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash (s'')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x}: \vec{\sigma}^\star \vdash \mathbf{let}\ \vec{z} = \{s'\}^\star_{\vec{z}}\ \mathbf{in}\ (s'')^\star_{\vec{x}}: (\vec{\sigma}^\star)^\times}$$

- $s = \textbf{for } y := 0 \textbf{ until } e \ \{s'\}_{\vec{z}};\ s''$. We have:

$$\dfrac{\dfrac{\vec{z} \subset \Omega, \vec{x} : \vec{\sigma} \quad \Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash e : \tau \quad \Gamma, y : \tau; \vec{z} : \vec{\tau}\,' \vdash s' : \textbf{seq}}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \textbf{for } y := 0 \textbf{ until } e \ \{s'\}_{\vec{z}} : \textbf{comm}} \quad \Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash s'' : \textbf{seq}}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \textbf{for } y := 0 \textbf{ until } e \ \{s'\}_{\vec{z}};\ s'' : \textbf{seq}}$$

where $\vec{z} : \vec{\tau}\,' \subset \Omega, \vec{x} : \vec{\sigma}$ (and then $\Omega^\star, \vec{x} : \vec{\sigma}^\star, \vec{z} : \vec{\tau}\,'^\star = \Omega^\star, \vec{x} : \vec{\sigma}^\star$). By induction hypothesis, we obtain $\Gamma^\star, y : \tau^\star, \vec{z} : \vec{\tau}\,'^\star \vdash \{s'\}^\star_{\vec{z}} : (\vec{\tau}\,'^\star)^\times$ and $\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash e^\star : \tau^\star$ and $\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash (s'')^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times$. We build then the following typing derivation :

$$\dfrac{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash e^\star : \tau^\star \quad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash \vec{z} : (\vec{\tau}\,'^\star)^\times \quad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star, y : \tau^\star \vdash \{s\}^\star_{\vec{z}} : (\vec{\tau}\,'^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash \textbf{rec}(e^\star, \vec{z}, \lambda y.\lambda \vec{z}.\{s'\}^\star_{\vec{z}}) : (\vec{\tau}\,'^\star)^\times}$$

and then:

$$\dfrac{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash \textbf{rec}(e^\star, \vec{z}, \lambda y.\lambda \vec{z}.\{s'\}^\star_{\vec{z}}) : (\vec{\tau}\,'^\star)^\times \quad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash (s'')^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times}{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star \vdash \textbf{let } \vec{z} = \textbf{rec}(e^\star, \vec{z}, \lambda y.\lambda \vec{z}.\{s'\}^\star_{\vec{z}}) \textbf{ in } (s'')^\star_{\vec{x}} : (\vec{\sigma}^\star)^\times}$$

□

Acknowledgments

We are grateful to Olivier Danvy for many fruitful discussions on the topics presented in this article and to Neil Jones for his early support. We also would like to thank the anonymous referees for their helpful comments.

Bibliography

[Appel, 1998] Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, UK.

[Avigad and Feferman, 1998] Avigad, J. and Feferman, S. (1998). Gödel's functional ("dialectica") interpretation. In Buss, S. R., editor, *Handbook of Proof Theory*, pages 337–405. Elsevier Science Publishers, Amsterdam.

[Beckmann and Weiermann, 2000] Beckmann, A. and Weiermann, A. (2000). Characterizing the elementary recursive functions by a fragment of Gödel's T. *Archive for Mathematical Logic*, V39:475–491.

[Calude, 1988] Calude, C. (1988). *Theories of computational complexity*. Elsevier Science Inc., New York, NY, USA.

[Colson, 1991] Colson, L. (1991). About primitive recursive algorithms. *Theoretical Computer Science*, 83:57–69.

[Colson and Fredholm, 1998] Colson, L. and Fredholm, D. (1998). System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315.

[Crolard et al., 2006] Crolard, T., Lacas, S., and Valarcher, P. (2006). On the Expressive Power of the Loop Language. *Nordic Journal of Computing*, 13(1-2):46–57.

[Davis and Weyuker, 1983] Davis, M. and Weyuker, E. (1983). *Computability, Complexity and Languages*. Academic Press.

[DOD, 1980] DOD (1980). *The Programming Language Ada. Reference Manual*. Springer, Berlin.

[Donahue, 1977] Donahue, J. E. (1977). Locations considered unnecessary. *Acta Inf.*, 8:221–242.

[Felleisen and Friedman, 1987] Felleisen, M. and Friedman, D. P. (1987). A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, page 314, New York, NY, USA. ACM Press.

[Filinski, 1994] Filinski, A. (1994). Representing monads. In *Conference Record of the Twenty-First Annual Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon.

[Filliâtre, 2003] Filliâtre, J.-C. (2003). Verification of non-functional programs using interpretations in type theory. *J. Funct. Program*, 13(4):709–745.

[Filliâtre and Marché, 2004] Filliâtre, J.-C. and Marché, C. (2004). Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag.

[Gellerich and Plödereder, 2001] Gellerich, W. and Plödereder, E. (2001). Parameter-induced aliasing in ada. In Craeynest, D. and Strohmeier, A., editors, *Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference Leuven, Belgium, May 14-18, 2001, Proceedings*, volume 2043 of *Lecture Notes in Computer Science*, pages 88–99. Springer.

[Gifford and Lucassen, 1986] Gifford, D. and Lucassen, J. (1986). Integrating functional and imperative programming. In *ACM Symposium on Principles of Programming Languages*.

[Girard et al., 1989] Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*, volume 7. Cambridge Tracts in Theorical Comp. Sci.

[Gödel, 1958] Gödel, K. (1958). Über eine bisher noch nicht benützteerweiterung des finiten standpunktes. *Dialectica*, 12:280–287.

[Gödel, 1990] Gödel, K. (1990). *Collected Works, Volume 2*. Oxford University Press, Oxford.

[Hatcliff and Danvy, 1997] Hatcliff, J. and Danvy, O. (1997). Thunks and the lambda-calculus. *J. Funct. Program.*, 7(3):303–319.

[ISO, 2003] ISO (2003). C♯ language specification ISO/IEC 23270.

[Jones, 1997] Jones, N. D. (1997). *Computability and Complexity From a Programming Perspective*. The MIT Press.

[Kahn, 1987] Kahn, G. (1987). Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag.

[Kelsey et al., 1998] Kelsey, R., Clinger, W., and Rees, J. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[Kreisel, 1951] Kreisel, G. (1951). On the interpretation of non-finitist proofs - part I. *J. Symb. Log*, 16(4):241–267.

[Landin, 1964] Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195.

[Meyer and Ritchie, 1976] Meyer, A. R. and Ritchie, D. M. (1976). The complexity of loop programs. In *Proc. ACM Nat. Meeting*.

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML, Revised edition*. MIT Press.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1):55–92.

[O'Hearn and Tennent, 1997] O'Hearn, P. W. and Tennent, R. D., editors (1997). *Algol-like Languages*. Birkhaüser.

[Peter, 1968] Peter, R. (1968). *Recursive Functions*. Academic Press.

[Plotkin, 1981] Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

[Reynolds, 1978] Reynolds, J. C. (1978). Syntactic control of interference. In *POPL*, pages 39–46.

[Reynolds, 1981] Reynolds, J. C. (1981). The essence of Algol. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam. IFIP, North-Holland.

[Reynolds, 1998] Reynolds, J. C. (1998). *Theories of programming languages*. Cambridge University Press.

[Schmidt, 1994] Schmidt, D. A. (1994). *The Structure of Typed Programming Languages*. The MIT Press, Cambridge, Massachusetts.

[Schütte, 1967] Schütte, K. (1967). *Proof theory*. Addison Wesley.

[Stoy, 1977] Stoy, J. (1977). *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT.

[Talpin and Jouvelot, 1994] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

[Turner, 2004] Turner, D. A. (2004). Total functional programming. *J. UCS*, 10(7):751–768.

[Wadler, 1990] Wadler, P. (1990). Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France.

[Wadler, 1998] Wadler, P. (1998). The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, Baltimore. ACM.

Chapitre 3

Logiques de programmes et mécanismes de contrôle

∗. Les résultats présentés dans ce chapitre ont été obtenus en collaboration avec E. Polonowski et n'ont pas encore été publiés.

A program logic for higher-order procedural variables and non-local jumps

Abstract

Relying on the formulae-as-types paradigm for classical logic, we define a program logic for an imperative language with higher-order procedural variables and non-local jumps. Then, we show how to derive a sound program logic for this programming language. As a by-product, we obtain a non-dependent type system which is more permissive than what is usually found in statically typed imperative languages. As a generic example, we encode imperative versions of delimited continuations operators **shift** and **reset**.

1 Introduction

In his seminal series of papers [Landin, 1964, Landin, 1965a, Landin, 1965b], Landin proposed a direct translation of an idealized Algol into the λ-calculus. This translation required to extend the λ-calculus with a new operator **J** in order to handle non-local jumps in Algol. This operator, which was described in detail in [Landin, 1965c] (see also [Thielecke, 1998] for an introduction), is the father to all control operators in functional languages (such as the famous **call/cc** of Scheme [Kelsey et al., 1998] or Standard ML of New Jersey [Harper et al., 1993]). The syntactic theory of control has subsequently been explored thoroughly by Felleisen [Felleisen, 1987].

A type system for control operators which extends the so-called Curry-Howard correspondence [Curry and Feys, 1958, Howard, 1969] to classical logic first appeared in Griffin's pioneering work [Griffin, 1990], and was immediately generalized to Peano's arithmetic by Murthy in his thesis [Murthy, 1991a]. The following years, this extension of the formulas-as-types paradigm to classical logic has then studied by several researchers, for instance in [Barbanera and Berardi, 1994, Rehof and Sørensen, 1994, de Groote, 1995, Krivine, 1994, Parigot, 1993b] and many others since.

It is thus tempting to revisit Landin's work in the light of the formulas-as-types interpretation of control. Indeed, it is notoriously difficult to derive a sound program logic for an imperative language with procedures and non-local jumps [O'Donnell, 1982]. On the other hand, adding dependent types to such an imperative language, and translating type derivations into proof derivations seems more tractable. The difficult to obtain program logic is then mechanically derived.

As a stepping stone, we focus in this paper on Peano's arithmetic. The corresponding functional language (through the proofs-as-programs paradigm) is thus an extension of Gödel System T [Gödel, 1958] with control operators as described in [Murthy, 1991a]. We shall use instead a variant which was proposed by Leivant [Leivant, 1990, Leivant, 2002] (and rediscovered independently by Krivine and Parigot in the second-order framework [Krivine and Parigot, 1990]). The main advantage of this variant is that it requires no encoding in formulas (with Gödel numbers) to reason about functional programs. Moreover it can be extended to any other algebraic datatypes (such as lists or trees). In this paper, the control operators are given an indirect semantics through a call-by-value CPS transform (we do not consider any direct style semantics). As noticed in [Murthy, 1991a], this CPS transformation operates a variant of Kuroda's translation on dependent types [Kuroda, 1951].

The imperative counterpart of Gödel System T [Gödel, 1958] (called Loop$^\omega$) which was defined by the authors in [Crolard et al., 2009], is essentially an extension of Meyer and Ritchie's Loop language [Meyer and Ritchie, 1976] with higher-order procedural variables. Loop$^\omega$ is a genuine imperative language as opposed to functional languages with imperative features. However, Loop$^\omega$ is a "pure" imperative language: side-effects and aliasing are forbidden. These restrictions enable simple location-free operational semantics

[Donahue, 1977]. Moreover, the type system relies on the distinction between mutable and read-only variables to prevent procedure bodies to refer to non-local mutable variables. This property is crucial to guarantee that fix-points cannot be encoded using procedural variables. Since there is no recursivity and no unbounded loop construct in LOOP$^\omega$, one can prove that all LOOP$^\omega$ programs terminate (note that the expressive power of system T is still attained thanks to mutable higher-order procedural variables).

In this paper, we add to extend LOOP$^\omega$ with first-order dependent types. This led us in particular to relax the underlying static type system. Indeed, for instance, after the assignment $x := 0$, the type of x is $\mathbf{nat}(0)$. The type of x is thus changed by this assignment whenever the former value of x is different from 0. Moreover, the type of x before the assignment does not matter: there is no need to even require that x be a natural number. Pushing this idea to the limit, we obtain a type system for LOOP$^\omega$ where the type of any mutable variable can be changed by an assignment (or a procedure call). Although, this feature seems characteristic of a dynamic language, our type system is fully static. Moreover, since dealing with mutable variables is natural in imperative programming, global variables are easily simulated with in usual state-passing style. Besides, the logical meaning of this simulation is perfectly clear.

This above remark suggests that usual static type systems for imperative languages are overly restrictive. Indeed, a pseudo-dynamic type system is quite expressive: typing an imperative program in state-passing style amounts (up to curryfication) to typing its functional image with a parameterised state monad [Atkey, 2006]. To capture this expressivity would usually require an effect system on the imperative side [Gifford and Lucassen, 1986]. Moreover, a pseudo-dynamic type system provides an elegant way to deal with uninitialized variables. Indeed, in a logical type system, a type is not necessarily inhabited and there are thus no default values for arbitrary types. Although it is possible to design a type system which track uninitialized variables, it would be awkward (and meaningless from a logical standpoint). On the other hand, in a pseudo-dynamic type system any mutable variable can be initialized to a default inhabited type with a chosen default value.

Let us summarize the main developments of this paper. We rephrase Landin's translation for a total imperative language featuring higher-order procedures and non-local jumps and then we rely on the Curry-Howard correspondence for classical logic to derive a program logic for this language. To be more specific, we define a framework which includes an imperative language **I**, a call-by-value functional language **F** and a retraction between **I** and **F** as follows:

- The functional language **F**, which is our formulation of Gödel System T, is equipped with two usual type systems, a simple type system **IS** and a dependent type system **ID** which is akin to Leivant's **M1LP** [Leivant, 1990]. In particular, dependent types include arbitrary formulas of first-order arithmetic.
- The imperative language **I** (essentially LOOP$^\omega$ from [Crolard et al., 2009]) is an extension of Meyer and Ritchie's LOOP language [Meyer and Ritchie, 1976] with higher-order procedural variables. Language **I** is also equipped with two (unusual) type systems, a pseudo-dynamic simple type system **IS** and a dependent type system **ID**.
- A compositional translation * from **I** to **F** is definable [Crolard et al., 2009]. This translation actually provides a simulation: each evaluation step of an imperative program is simulated by a bounded number of reduction step of its functional image. In this paper, we show that this translation is type-preserving in both the pseudo-dynamic and dependent frameworks.
- We characterize the shape of the functional image of an imperative program by *: these functional terms are monadic normal forms [Hatcliff and Danvy, 1994] (also called A-normal forms [Flanagan et al., 1993]). A reverse translation $^\diamond$ from monadic normal forms of **F** to **I** is then defined, which is also compositional and type-preserving in both the pseudo-dynamic and dependent frameworks.
- We show that $\langle {}^\diamond, {}^\star \rangle$ forms a retraction. Consequently, from any dependently-typed functional program (and thus from any proof in Heyting arithmetic) we can derive an imperative program which implements the corresponding dependent type.

- $\mathbf{F}^c$ is then defined as an extension of **F** with control operators **callcc** and **throw** (taken from [Harper et al., 1993]). The semantics of $\mathbf{F}^c$ is given by a call-by-value CPS-transformation into **F**. Following [Hatcliff and Danvy, 1994], since the functional image of an imperative program is in monadic normal form, we factor the CPS transformation through Moggi's computational meta-language [Moggi, 1990, Moggi, 1991].
- From $\mathbf{F}^c$ we derive $\mathbf{I}^c$ which extends **I** with two primitive procedures **callcc** and **throw**. Although we do not pretend that these control operators are natural in an imperative language, they can be used to define more conventional statements which have to interact with the control flow. It is of course not possible to encode arbitrary **goto** statements since our programming language is total.
- Finally, as a generic example, by combining a simulated global state with **callcc** and **throw**, we show how to encode **shift** and **reset** [Danvy and Filinski, 1989] (and thus any representable monad) using Filinski's decomposition [Filinski, 1994]. As a consequence, we obtain an indirect formulas-as-types interpretation of delimited continuations in a dependently-typed framework.

Related works

Although there has been several attempts to design program logics for higher-order procedures or non-local jumps, we are not aware of any work which combines both in an imperative setting.

Of course, there has been much research on Floyd-Hoare logics [Floyd, 1967, Hoare, 1969, Hoare, 1971] (see the surveys [Apt, 1981] and [Cousot, 1990]). Such program logics for higher-order procedures have been defined instance in [Damm and Josko, 1983] (for Clarke's language L4 [Clarke, 1979]) or more recently for stored parameterless procedures in [Reus and Streicher, 2005]. Program logics for jumps exists since [Clint and Hoare, 1972], and although designing such a logic is error-prone [O'Donnell, 1982], there have been successfully used recently for proving properties in low-level languages [Feng et al., 2006, Tan and Appel, 2006].

A dependent type system for an imperative programming language is defined in [Xi, 2000], where the dependent types are restricted to ensure that type checking remains decidable. They also made the observation that imperative dependent types requires to allow the type of variables to change during evaluation. However they chose to restrict the type system in order to guarantee that the extracted program is typable in some usual static (non-dependent) type systems. On the contrary, we believe that a dynamically-flavoured static type system should be advocated.

Proofs-as-Imperative-Program [Poernomo, 2003, Poernomo and Crossley, 2003] adapts the proofs-as-programs paradigm for the synthesis of imperative SML programs with side-effect free return values. The type theory is however intrinsically constructive: it requires a strong existential quantifier which is not compatible with classical logic [Herbelin, 2005].

The Dependent Hoare Type Theory [Nanevski et al., 2006] and the Imperative Hoare Logic [Honda et al., 2005, Honda et al., 2006] are frameworks for reasoning about effectful higher-order functions. The dynamic semantics of those systems are much more complicated (since aliasing is allowed) than our location-free semantics. Although the Dependent Hoare Type Theory contains control expressions and enjoys a formulas-as-types interpretation, it is not clear whether programs correspond to proofs in some deduction system for classical logic.

Plan of the paper. In Section 2, we present the untyped functional language **F**, the untyped imperative language **I** and and their dynamic semantics. We define also the retraction $\langle \diamond, \star \rangle$ between programs of **I** and monadic normal forms of **F**. Section 3 is devoted to the definition of the pseudo-dynamic type system **IS** and its properties. Section 4 contains the definitions of the dependently-typed systems **ID** and **FD** and the various proofs of type preservation. In Section 5, we extend language **F** with control operators and its type system is raised to classical arithmetic $\mathbf{FD}^c$. Finally, in Section 6, we extend **I** with non-local jumps and we derive a corresponding program logic $\mathbf{ID}^c$.

2 Dynamic semantics of I and F

In this section, we present the untyped functional language **F** (which is a variant of Gödel System T) and the untyped imperative language **I** (which is an extension of Meyer and Ritchie's LOOP language [Meyer and Ritchie, 1976] with higher-order procedural variables studied in [Crolard et al., 2009]). We define also the dynamic semantics of both languages and the retraction $\langle \diamond, \star \rangle$ between programs of **I** and monadic normal forms of **F**.

2.1 Language F

Gödel System T may be defined as the simply typed λ-calculus extended with a type of natural numbers and with primitive recursion at all types [Girard et al., 1989]. The language **F** we consider in this paper a variant of System T with product types (n-ary tuples actually) and a constant-time predecessor operation (since any definition of this function as a term of System T is at least linear under the call-by-value evaluation strategy [Colson and Fredholm, 1998]). Moreover, we formulate this system directly as a context semantics (a set of reduction rules together with an inductive definition of evaluation contexts). As usual, we consider terms up to α-conversion and the set $\mathrm{FV}(t)$ of free variables of a term t is defined in the standard way. The rewriting system is summarized in Figure 2.1, where variables $x, x_1, ..., x_n, y$ range over a set of identifiers and $t[v_1/x_1, ...,$

(terms)

$$
\begin{array}{rcl}
t & ::= & x \\
& | & 0 \\
& | & S(t) \\
& | & \mathbf{pred}(t) \\
& | & t_1\ t_2 \\
& | & \lambda x.t \\
& | & (t_1, ..., t_n) \\
& | & \mathbf{let}\ (x_1, ..., x_n) = t_1\ \mathbf{in}\ t_2 \\
& | & \mathbf{rec}(t_1, t_2, t_3)
\end{array}
$$

(values)

$$
\begin{array}{rcl}
v & ::= & x \\
& | & 0 \\
& | & S(v) \\
& | & (v_1, ..., v_n) \\
& | & \lambda x.t
\end{array}
$$

(contexts)

$$
\begin{array}{rcl}
C[\,] & ::= & [\,] \\
& | & C[\,]\ t \\
& | & v\ C[\,] \\
& | & S(C[\,]) \\
& | & \mathbf{pred}(C[\,]) \\
& | & \mathbf{rec}(C[\,], t_2, t_3) \\
& | & \mathbf{rec}(v_1, C[\,], t_3) \\
& | & \mathbf{rec}(v_1, v_2, C[\,]) \\
& | & (v_1, ...v_{i-1}, C[\,], t_{i+1}..., t_n) \\
& | & \mathbf{let}\ (x_1, ..., x_n) = C[\,]\ \mathbf{in}\ t
\end{array}
$$

(evaluation rules)

$$
\begin{array}{rcl}
C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\
C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\
C[\mathbf{rec}(0, v_2, \lambda x.\lambda \vec{y}\,.t)] & \rightsquigarrow & C[v_2] \\
C[\mathbf{rec}(S(v_1), v_2, \lambda x.\lambda \vec{y}\,.t)] & \rightsquigarrow & C[\lambda x.\lambda \vec{y}\,.t\ v_1\ \mathbf{rec}(v_1, v_2, \lambda x.\lambda \vec{y}\,.t)] \\
C[\lambda x.t\ v] & \rightsquigarrow & C[t[v/x]] \\
C[\mathbf{let}\ (x_1, ..., x_n) = (v_1, ..., v_n)\ \mathbf{in}\ t] & \rightsquigarrow & C[t[v_1/x_1, ..., v_n/x_n]]
\end{array}
$$

Figure 2.1. Syntax and context semantics of Language **F**

$v_n/x_n]$ denotes the usual capture-avoiding substitution.

Remark 2.1. In order to distinguish the successor S (which is a constructor) from the successor seen as an operation (whose evaluation should imply a reduction step), we use the keyword **succ** as an abbreviation for $\lambda x.S(x)$.

Remark 2.2. We write $\lambda(x_1, ..., x_n).t$ (or $\lambda \vec{x}\,.t$) as an abbreviation for $\lambda z.\mathbf{let}\ (x_1, ..., x_n) = z\ \mathbf{in}\ t$ where z is a fresh variable. Similarly, we write $\lambda().t$ as an abbreviation for $\lambda z.\mathbf{let}\ () = z\ \mathbf{in}\ t$ where z is a fresh variable.

2.1.1 Example: the Ackermann function

The Ackermann function is an example of function known not to be primitive recursive [Peter, 1968] but which can be represented in System T. Here follows an example of a slightly modified version of the function defined by the following equations [Leivant, 2002]:

$$\begin{array}{lll} (1) & \mathbf{a}(0,n) & = \mathbf{s}(n) \\ (2) & \mathbf{a}(\mathbf{s}(z),0) & = \mathbf{s}(\mathbf{s}(0)) \\ (3) & \mathbf{a}(\mathbf{s}(z),\mathbf{s}(u)) & = \mathbf{a}(z,\mathbf{a}(\mathbf{s}(z),u)) \end{array}$$

$$ack(m,n) = \mathbf{rec}(m, \lambda y.S(y), \lambda i.\lambda f.\lambda y.\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k)))\ n$$

2.2 Language I

The untyped language **I** is essentially the LOOP$^\omega$ language presented in [Crolard et al., 2009] except that LOOP$^\omega$ was explicitly typed. Moreover the loop syntax is now **for** $y := 0$ **until** e $\{s\}$ where the bound e is excluded from the range (since this new syntax corresponds more closely to reasoning by induction). The location-free transition semantics [Donahue, 1977] is also the same as in [Crolard et al., 2009] except that we consider only sequences. Although it is somewhat more verbose, both semantics are clearly equivalent.

2.2.1 Syntax

The raw syntax of imperative programs is given below. There is nothing particular to this syntax except that we annotate each block $\{s\}_{\vec{x}}$ with a list of variables $\vec{x}$ (which corresponds to the mutable variables which may occur in the block). In the following grammar, x, y, z range over a set of identifiers, $\bar{q}$ ranges over natural numbers (*i.e.* constant literals), ε denotes the empty sequence and $*$ denotes the singleton value. Free identifiers are defined in the standard way (see Appendix A).

$$\begin{array}{lrcl} (\textit{command}) & c & ::= & \{s\}_{\vec{x}} \\ & & | & \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}} \\ & & | & y := e \ \mid\ \mathbf{inc}(y) \ \mid\ \mathbf{dec}(y) \\ & & | & e(\vec{e}\,;\vec{y}\,) \\ \\ (\textit{sequence}) & s & ::= & \varepsilon \\ & & | & c\,;\ s \\ & & | & \mathbf{cst}\ y = e;\ s \\ & & | & \mathbf{var}\ y := e;\ s \\ \\ (\textit{expression}) & e & ::= & y \ \mid\ * \ \mid\ \bar{q} \\ & & | & \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,;\mathbf{out}\ \vec{z}\,)\ \{s\}_{\vec{z}} \end{array}$$

Remark 2.3. *(no aliasing).* In order to avoid parameter-induced aliasing problems, we assume that all y_i are pairwise distinct in a procedure call $p(\vec{e}\,;\vec{y}\,)$.

Remark 2.4. *(annotations).* In a block $\{s\}_{\vec{x}}$, the variables in $\vec{x}$ must be visible (i.e. declared as enclosing local variables or as procedure out parameters). The list $\vec{x}$ must also contain all the free mutable variables occurring in the sequence. Such annotations can automatically be inferred by taking, for instance, all the visible mutable variables.

2.2.2 Example: the addition procedure

Here follows a procedure that computes the addition of two natural numbers:

```
cst add = proc (in X, Y; out Z) {
    Z := X;
    for I := 0 until Y {
        inc(Z);
    }_Z;
}_Z
```

2.2.3 Operational semantics

The operational semantics is given as transition system [Plotkin, 1981] which defines inductively a binary relation between states. A state is a pair (s, μ) consisting of a sequence s and a store μ, where a store is a finite ordered mapping from (mutable) variables to closed imperative values (i.e. integer literals, procedures and $*$).

Note that expressions do not require any evaluation since they are either variables or values. We introduce thus the following notation which allows us to treat uniformly values and variables in the semantics:

Notation 2.5. *Given a store μ, let φ_μ be the trivial extension of μ to expressions defined as follows $\varphi_\mu(x) = \mu(x)$ if x is a variable and $\varphi_\mu(w) = w$ otherwise. In the sequel, we write $e =_\mu w$ for $\varphi_\mu(e) = w$.*

Notation 2.6. *Let s be a sequence. We write $s[x \leftarrow w]$ for the substitution of a read-only variable x by a closed imperative value w and $s[y \hookleftarrow z]$ for the renaming of a mutable variable y by a mutable variable z. The formal definitions are similar to those given in [Crolard et al., 2009].*

Notation 2.7. *Let μ be a store. We write $(\mu[y \leftarrow w])$ for the store update,* i.e. *$\mu[y \leftarrow w](x) = \mu(x)$ if $x \neq y$ and $\mu[y \leftarrow w](y) = \mu(y)$. We write $(\mu, y \leftarrow w)$ for the store extension with the new variable y mapped to w.*

This definition of the transition system is summarized in Figure 2.2.

$$((\{\}_{\vec{z}}; s), \mu) \mapsto (s, \mu) \qquad \text{(S.BLOCK-I)}$$

$$\frac{(s_1, \mu) \mapsto (s_1', \mu')}{((\{s_1\}_{\vec{z}};\ s_2), \mu) \mapsto ((\{s_1'\}_{\vec{z}};\ s_2), \mu')} \qquad \text{(S.BLOCK-II)}$$

$$((\textbf{var}\ y{:=}e;\ \varepsilon), \mu) \mapsto (\varepsilon, \mu) \qquad \text{(S.VAR-I)}$$

$$\frac{e =_\mu w \qquad (s, (\mu, y \leftarrow w)) \mapsto (s', (\mu', y \leftarrow w'))}{((\textbf{var}\ y{:=}e;\ s), \mu) \mapsto ((\textbf{var}\ y{:=}w';\ s'), \mu')} \qquad \text{(S.VAR-II)}$$

$$\frac{e =_\mu w}{((y := e;\ s), \mu) \mapsto (s, \mu[y \leftarrow w])} \qquad \text{(S.ASSIGN)}$$

$$\frac{\mu(y) = \bar{q}}{((\textbf{inc}(y);\ s), \mu) \mapsto ((y := \overline{q+1};\ s), \mu)} \qquad \text{(S.INC)}$$

$$\frac{\mu(y) = \bar{q}}{((\textbf{dec}(y);\ s), \mu) \mapsto ((y := \overline{q \dot{-} 1};\ s), \mu)} \qquad \text{(S.DEC)}$$

$$\frac{\vec{e} =_\mu \vec{w} \qquad p =_\mu \textbf{proc}\ (\textbf{in}\ \vec{y}; \textbf{out}\ \vec{z})\ \{s'\}_{\vec{z}}}{((p(\vec{e}; \vec{r});\ s), \mu) \mapsto ((\{s'[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}]\}_{\vec{r}};\ s), \mu[\vec{r} \leftarrow *])} \qquad \text{(S.CALL)}$$

$$\frac{e =_\mu w}{((\textbf{cst}\ y = e;\ s), \mu) \mapsto (s[y \leftarrow w], \mu)} \qquad \text{(S.CST)}$$

$$\frac{e =_\mu \bar{0}}{((\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s\}_{\vec{z}};\ s'), \mu) \mapsto (s', \mu)} \qquad \text{(S.FOR-I)}$$

$$\frac{e =_\mu \overline{q+1}}{((\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s\}_{\vec{z}};\ s'), \mu) \mapsto ((\{\textbf{for}\ y := 0\ \textbf{until}\ \bar{q}\ \{s\}_{\vec{z}};\ s[y \leftarrow \bar{q}]\}_{\vec{z}};\ s'), \mu)} \qquad \text{(S.FOR-II)}$$

Figure 2.2. Transition semantics

Remark 2.8. This semantics is clearly deterministic since there is always at most one rule which can be applied (depending on the content of the store and the shape of the command).

2.3 Translation from I to F and simulation

We recall the translation, similar in spirit to Landin's translation of Algol-like languages, described in [Crolard et al., 2009]. The intuition behind this translation of imperative programs into functional programs is the following: a sequence $(c_1;...;c_n;)_{\vec{x}}$ is translated into:

$$\textbf{let } \vec{x}_1 = c_1^\star \textbf{ in } ... \textbf{ let } \vec{x}_n = c_n^\star \textbf{ in } \vec{x}$$

where each $\vec{x}_i \subseteq \vec{x}$ corresponds to the "output" of command c_i and $\vec{x}$ is the output of the sequence.

Definition 2.9. *For any expression e, sequence s and variables $\vec{x}$, the translations $e^\star$ and $(s)^\star_{\vec{x}}$ into terms of language* **F** *are defined by mutual induction as follows:*

- $\bar{n}^\star = S^n(0)$
- $y^\star = y$
- $*^\star = ()$
- $(\textbf{proc } (\textbf{in } \vec{y}; \textbf{out } \vec{z})\, \{s\}_{\vec{z}})^\star = \lambda \vec{y}.(s)^\star_{\vec{z}}\, [\vec{()}/\vec{z}]$
- $(\varepsilon)^\star_{\vec{x}} = \vec{x}$
- $(\textbf{var } y := e;\ s)^\star_{\vec{x}} = (s)^\star_{\vec{x}}[e^\star/y]$
- $(\textbf{cst } y = e;\ s)^\star_{\vec{x}} = \textbf{let } y = e^\star \textbf{ in } (s)^\star_{\vec{x}}$
- $(y := e;\ s)^\star_{\vec{x}} = \textbf{let } y = e^\star \textbf{ in } (s)^\star_{\vec{x}}$
- $(\textbf{inc}(y);\ s)^\star_{\vec{x}} = \textbf{let } y = \textbf{succ}(y) \textbf{ in } (s)^\star_{\vec{x}}$
- $(\textbf{dec}(y);\ s)^\star_{\vec{x}} = \textbf{let } y = \textbf{pred}(y) \textbf{ in } (s)^\star_{\vec{x}}$
- $(p(\vec{e}\,;\vec{z});\ s)^\star_{\vec{x}} = \textbf{let } \vec{z} = p^\star\ (\vec{e}^{\,\star}) \textbf{ in } (s)^\star_{\vec{x}}$
- $(\{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \textbf{let } \vec{z} = (s_1)^\star_{\vec{z}} \textbf{ in } (s_2)^\star_{\vec{x}}$
- $(\textbf{for } y := 0 \textbf{ until } e\ \{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \textbf{let } \vec{z} = \textbf{rec}(e^\star, \vec{z}, \lambda y.\lambda \vec{z}.(s_1)^\star_{\vec{z}}) \textbf{ in } (s_2)^\star_{\vec{x}}$

2.3.1 Simulation

We recall the simulation theorem from [Crolard et al., 2009] which states that for any sequence s, the evaluation of s is simulated by the reduction of $(s)^\star_{\vec{z}}$.

Proposition 2.10. *For any state (s, μ), if $\vec{x} = dom(\mu)$ and $\vec{z} \subseteq \vec{x}$ we have:*

$$(s, \mu) \mapsto (s', \mu') \textit{ implies } (s)^\star_{\vec{z}}[\mu(\vec{x})^\star/\vec{x}] \rightsquigarrow^* (s')^\star_{\vec{z}}[\mu'(\vec{x})^\star/\vec{x}]$$

2.4 Translation from F to I and retraction

In this section, we show how to translate a functional program of **F** into an imperative program of **I**. However, this translation is only defined for a sub-language $\mathcal{L}$ of monadic normal forms (terms where any non-trivial intermediate computation is named [Hatcliff and Danvy, 1994, Flanagan et al., 1993]). This sub-language $\mathcal{L}$ characterize the image of imperative programs by *. We show in appendix C.5 how to transform any term of language **F** into a monadic normal form of $\mathcal{L}$.

Definition 2.11. *We define inductively $\mathcal{W}$, $\mathcal{V} = \cup_{n\in\mathbb{N}}\mathcal{V}_n$ and $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$, where $\mathcal{L}_n$ (resp. $\mathcal{V}_n$) are indexed families of terms (resp. values) of* **F**, *as follows:*

- $x \in \mathcal{W}$
- $() \in \mathcal{W}$
- $S^n(0) \in \mathcal{W}$
- $\lambda\vec{x}.t \in \mathcal{W}$ *if* $t \in \mathcal{L}_p$
- $(w_1, ..., w_n) \in \mathcal{V}_n$ *if* $w_1, ..., w_n \in \mathcal{W}$
- $v \in \mathcal{L}_n$ *if* $v \in \mathcal{V}_n$
- **let** $x = w$ **in** $u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- **let** $x = \mathbf{succ}(w)$ **in** $u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- **let** $x = \mathbf{pred}(w)$ **in** $u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- **let** $\vec{x} = w\ v$ **in** $u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$, $v \in \mathcal{V}_p$ *and* $u \in \mathcal{L}_n$
- **let** $(x_1, ..., x_p) = \mathbf{rec}(w, v, \lambda y.\lambda(z_1, ..., z_p).t)$ **in** $u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$, $v \in \mathcal{V}_p$ *and* $u \in \mathcal{L}_n$
- **let** $(x_1, ..., x_p) = t$ **in** $u \in \mathcal{L}_n$ *if* $t \in \mathcal{L}_p$ *and* $u \in \mathcal{L}_n$

Proposition 2.12. *For any sequence s, any expression e and any variables $\vec{x} = (x_1, ..., x_n)$, $(s)^\star_{\vec{x}} \in \mathcal{L}_n$ and $e^\star \in \mathcal{W}$.*

Proof. Straightforward mutual induction on s and e. □

Notation 2.13. *In the sequel, we shall use the following abbreviations:*

$$\begin{aligned}
\mathbf{var}\ \vec{y};\ s &= \mathbf{var}\ y_1 := *;\ ...;\ \mathbf{var}\ y_n := *;\ s \\
\mathbf{var}\ \vec{y} := \vec{w};\ s &= \mathbf{var}\ y_1 := w_1;\ ...;\ \mathbf{var}\ y_n := w_n;\ s \\
\mathbf{cst}\ \vec{y} = \vec{z};\ s &= \mathbf{cst}\ y_1 = z_1;\ ...;\ \mathbf{cst}\ y_n = z_n;\ s \\
\vec{y} := \vec{w};\ s &= y_1 := w_1;\ ...;\ y_n := w_n;\ s
\end{aligned}$$

Definition 2.14. *For any value $w \in \mathcal{W}$ and any term $t \in \mathcal{L}_n$, the translation $w^\diamond$ and $t^\diamond_{\vec{r}}$ are defined by mutual induction, where $\vec{r} = (r_1, ..., r_n)$, z and $\vec{z} = (z_1, ..., z_n)$ are fresh variables, as follows:*

- $()^\diamond = *$
- $S^n(0)^\diamond = \bar{n}$
- $y^\diamond = y$
- $(\lambda\vec{x}.t)^\diamond = \mathbf{proc}\ (\mathbf{in}\ \vec{x}; \mathbf{out}\ \vec{z})\ \{t^\diamond_{\vec{z}}\}_{\vec{z}}$ *where* $\vec{z} = (z_1, ..., z_m)$ *and* $t \in \mathcal{L}_m$
- $(\vec{w})^\diamond_{\vec{r}} = \vec{r} := \vec{w};\ \varepsilon$
- $(\mathbf{let}\ y = w\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{cst}\ y = w^\diamond;\ (u)^\diamond_{\vec{r}}$
- $(\mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ z := w^\diamond;\ \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}}$
- $(\mathbf{let}\ y = \mathbf{pred}(w)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ z := w^\diamond;\ \mathbf{dec}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}}$
- $(\mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z};\ w^\diamond(\vec{w}^\diamond; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$
- $(\mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda\vec{y}.t)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z} := \vec{w};\ \mathbf{for}\ i := 0\ \mathbf{until}\ w^\diamond\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ t^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$
- $(\mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z};\ \{(t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$

Remark 2.15. Note that all identifiers of the source term are mapped to read-only variables. Indeed, mutable are introduced locally, assigned and then only used to initialize local read-only variables. This property ensures that mutable variables do not occur in the body of procedures in the resulting LOOP$^\omega$ program: the only mutable variables are fresh variables introduced during the translation.

2.4.1 Retraction

We prove that for any term t of $\mathcal{L}$, the term $(t^\diamond_{\vec{r}})^\star_{\vec{r}}$ is convertible with t. Both terms are not equal in general since some "administrative" redices are introduced by the translations. However, equality holds for integer values.

Definition 2.16. *We define the reduction relation $\approx$ as the reflexive, symmetric, transitive and contextual closure of the reduction $\rightsquigarrow$ for arbitrary contexts.*

Proposition 2.17.

- *Given a term $t \in \mathcal{L}$ and a fresh mutable variable tuple $\vec{r}$ we have $(t^{\diamond}_{\vec{r}})^{\star}_{\vec{r}} \approx t$.*
- *Given a value $w \in \mathcal{W}$, if $w = S^n(0)$ or $w = *$ then $w^{\diamond\star} = w$ else $w^{\diamond\star} \approx w$.*

Proof. See Appendix-A. □

Proposition 2.18. *For any value w, if $w = \bar{q}$ or $w = *$ then $w^{\star\diamond} = w$.*

Proof. By Proposition 2.17, if $w = \bar{q}$ or $w = *$ then $w^{\star\diamond\star} = w^{\star}$. □

3 Pseudo-dynamic Type System

In this section, we present the simple type system for language **F** and the pseudo-dynamic type system for language **I**. Then we show that both translation * and $\diamond$ preserve typability and that the transition semantics of **I** enjoys the usual "type preservation" and "progress" properties.

3.1 Functional simple type system FS

The functional simple type system **FS** is defined as usual for a simply typed λ-calculus extended with tuples, natural numbers and with primitive recursion at all types. The set $\Sigma_{\mathbf{FS}}$ of simple functional types is defined by the following grammar:

$$\begin{array}{rcl} \sigma & ::= & \mathbf{nat} \\ & | & \mathbf{unit} \\ & | & \sigma_1 \to \sigma_2 \\ & | & \sigma_1 \times \ldots \times \sigma_n \end{array}$$

The type system is summarized in Figure 3.1.

3.2 Pseudo-dynamic imperative type system IS

The static type system described in this section is called "pseudo-dynamic" since the type of a mutable variable is allowed to change during execution. For instance, the following sequence is well-typed, with the type of the variable x changing three times, from **unit** to **proc**(**in nat**; **out nat**) and finally to **nat**:

$$x := *;\ x := \mathbf{proc}(\mathbf{in}\ y;\ \mathbf{out}\ z)\{z := y\}_z;\ x(6; x);$$

It is however fully static in the sense that it guarantees statically that no type error can occur at run-time. As a side benefit, we obtain a convenient way to address the issue of uninitialized variables: any mutable variables can be initialized with the $*$ (which denotes the single value of type **unit**) and its type shall change later (when assigned its first relevant value).

The pseudo-dynamic type system may also be seen as a simple effect system [Gifford and Lucassen, 1986, Talpin and Jouvelot, 1994] since it is able to guarantee the absence of side-effects, aliasing and fix-points in well-typed programs. Its key feature which enable this property is the distinction between mutable variables and read-only variables. More formally, the set $\Sigma_{\mathbf{IS}}$ of imperative types is defined by the following grammar:

$$\sigma, \tau\ ::=\ \mathbf{nat} \mid \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \mid \mathbf{unit}$$

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \quad (\textsc{ident})$$

$$\Gamma \vdash 0:\mathbf{nat} \quad (\textsc{zero})$$

$$\frac{\Gamma \vdash t:\mathbf{nat}}{\Gamma \vdash S(t):\mathbf{nat}} \quad (\textsc{succ})$$

$$\frac{\Gamma \vdash t:\mathbf{nat}}{\Gamma \vdash \mathbf{pred}(t):\mathbf{nat}} \quad (\textsc{pred})$$

$$\frac{\Gamma \vdash t_1:\tau_1 \quad ... \quad \Gamma \vdash t_n:\tau_n}{\Gamma \vdash (t_1,...,t_n):\tau_1 \times ... \times \tau_n} \quad (\textsc{tuple})$$

$$\frac{}{\Gamma \vdash ():\mathbf{unit}} \quad (\textsc{unit})$$

$$\frac{\Gamma, x_1:\tau_1,...,x_n:\tau_n \vdash t:\tau \qquad \Gamma \vdash u:\tau_1 \times ... \times \tau_n}{\Gamma \vdash \mathbf{let}\ (x_1,...,x_n) = u\ \mathbf{in}\ t:\tau} \quad (\textsc{let})$$

$$\frac{\Gamma, x:\tau \vdash t:\sigma}{\Gamma \vdash \lambda x.t\ :\tau \to \sigma} \quad (\textsc{abs})$$

$$\frac{\Gamma \vdash t_1:\sigma \to \tau \quad \Gamma \vdash t_2:\sigma}{\Gamma \vdash t_1\ t_2:\tau} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t_1:\mathbf{nat} \quad \Gamma \vdash t_2:\tau \quad \Gamma, x:\mathbf{nat}, y:\tau \vdash t_3:\tau}{\Gamma \vdash \mathbf{rec}(t_1, t_2, \lambda x.\lambda y.t_3):\tau} \quad (\textsc{rec})$$

Figure 3.1. Functional type system **FS**

A typing environment has the form $\Gamma;\Omega$ where Γ and Ω are (possibly empty) lists of pairs $x:\tau$ (x ranges over variables and τ over types). Γ stands for read-only variables (constants and **in** parameters) and Ω stands for mutable variables (local variables and **out** parameters). We use two typing judgments, one for expressions and one for sequences: $\Gamma;\Omega \vdash e:\tau$ has the usual meaning "In the environment $\Gamma;\Omega$ the expression e is of type τ" ; the meaning of $\Gamma;\Omega \vdash s \rhd \Omega'$ is "In the environment $\Gamma;\Omega$ the sequence s has as final types for its variables those given in Ω'". The type system is given in Figure B.1. As usual, we consider programs up to renaming of bound variables, where the notion of free variable of a command is defined in the standard way.

Remark 3.1. Let us recall important features of this pseudo-dynamic type system shared with the static type system described in [Crolard et al., 2009]:

- *(scoping rules).* As usual for C-like languages, the scope of a constant (rule T.CST) or a variable (rule T.VAR) extends from the point of declaration to the end of the block containing the declaration.
- *(no side-effects).* Rule (T.PROC) implies that the only mutable variables which may occur inside the body of a procedure are its **out** parameters and its local mutable variables. This is enough to guarantee the absence of side-effects. However, side-effects can still be simulated by passing the non-local variable as an explicit **in out** parameter (see section-3.5).
- *(no fix-points).* Rule (T.PROC) also forbids the reading of non-local mutable variables: this is necessary to prevents the definition of fix-points in the language.

Let us define formally the notions of well-typed stores and states.

Definition 3.2. (store typing). *We say that a store μ is typable of output typing environment $\Omega = z_1:\tau_1,...,z_n:\tau_n$, denoted $\mu \rhd \Omega$ if and only if $\vec{z} \in dom(\mu)$ and for all $(z_i, w_i) \in \mu$ we have $\emptyset;\emptyset \vdash w_i:\tau_i$.*

$$\frac{x:\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x:\tau} \quad (\textsc{t.env})$$

$$\frac{}{\Gamma;\Omega \vdash \bar{q}:\mathbf{nat}} \quad (\textsc{t.num})$$

$$\frac{}{\Gamma;\Omega \vdash *:\mathbf{unit}} \quad (\textsc{t.unit})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}:\vec{\sigma}; \vec{z}:\overrightarrow{\mathbf{unit}} \vdash s \rhd \vec{z}:\vec{\tau}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y};\ \mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}}:\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\ \mathbf{out}\ \vec{\tau})} \quad (\textsc{t.proc})$$

$$\frac{}{\Gamma;\Omega,\Omega' \vdash \varepsilon \rhd \Omega'} \quad (\textsc{t.empty})$$

$$\frac{\Gamma;\Omega \vdash e:\tau \qquad \Gamma, y:\tau;\Omega \vdash s \rhd \Omega'}{\Gamma;\Omega \vdash \mathbf{cst}\ y = e;\ s \rhd \Omega'} \quad (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega \vdash e:\tau \qquad \Gamma;\Omega, y:\tau \vdash s \rhd \Omega' \qquad y \notin \Omega'}{\Gamma;\Omega \vdash \mathbf{var}\ y := e;\ s \rhd \Omega'} \quad (\textsc{t.var})$$

$$\frac{\Gamma;\Omega \vdash c \rhd \Omega' \qquad \Gamma;\Omega' \vdash s \rhd \Omega''}{\Gamma;\Omega \vdash c;\ s \rhd \Omega''} \quad (\textsc{t.seq})$$

$$\frac{\Gamma;\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\tau}}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}} \rhd \Omega,\vec{x}:\vec{\tau}} \quad (\textsc{t.block})$$

$$\frac{}{\Gamma;\Omega, y:\mathbf{nat} \vdash \mathbf{inc}(y) \rhd \Omega, y:\mathbf{nat}} \quad (\textsc{t.inc})$$

$$\frac{}{\Gamma;\Omega, y:\mathbf{nat} \vdash \mathbf{dec}(y) \rhd \Omega, y:\mathbf{nat}} \quad (\textsc{t.dec})$$

$$\frac{\Gamma;\Omega, y:\sigma \vdash e:\tau}{\Gamma;\Omega, y:\sigma \vdash y := e \rhd \Omega, y:\tau} \quad (\textsc{t.assign})$$

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash e:\mathbf{nat} \quad \Gamma, y:\mathbf{nat};\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\sigma}}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}} \rhd \Omega,\vec{x}:\vec{\sigma}} \quad (\textsc{t.for})$$

$$\frac{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p:\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\ \mathbf{out}\ \vec{\tau}) \quad \Gamma;\Omega,\vec{r}:\vec{\omega} \vdash \vec{e}:\vec{\sigma}}{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p(\vec{e};\vec{r}) \rhd \Omega,\vec{r}:\vec{\tau}} \quad (\textsc{t.call})$$

Figure 3.2. Imperative type system

Definition 3.3. (state typing). *We say that a state (s, μ) is typable of output typing environment Ω' for a restriction of the store to the variables $\vec{z}:\vec{\tau}$, which we write as $\vec{z}:\vec{\tau} \vdash (s,\mu) \rhd \Omega'$, if and only if $\mu \rhd \vec{z}:\vec{\tau}$ and $\emptyset; \vec{z}:\vec{\tau} \vdash s \rhd \Omega'$.*

3.3 Translations between IS and FS

We define translations * and $\diamond$ for simple types (which also form retraction at the type level) and we show that both translations preserve typing.

Definition 3.4. *For any type $\tau \in \Sigma_{\mathbf{IS}}$, the corresponding type $\tau^\star \in \Sigma_{\mathbf{FS}}$ is defined inductively as follows:*

- $\mathbf{unit}^\star = \mathbf{unit}$
- $\mathbf{nat}^\star = \mathbf{nat}$
- $\mathbf{proc}\ (\mathbf{in}\ \vec{\tau};\mathbf{out}\ \vec{\sigma})^\star = \vec{\tau}^\star \to \vec{\sigma}^\star$

Let us call $\Sigma^\star_{\mathbf{FS}}$ the image of $\Sigma_{\mathbf{IS}}$ by *. The sub-language $\Sigma^\star_{\mathbf{FS}}$ may clearly be characterized by the following grammar:

$$\sigma, \tau \ ::= \ \mathbf{nat} \ \mid \ \mathbf{unit} \ \mid \ \vec{\sigma} \to \vec{\tau}$$

Definition 3.5. *For any type $\sigma \in \Sigma^{\star}_{\mathbf{FS}}$ the translation $\sigma^{\diamond}$ is defined as follows:*

- $\mathbf{unit}^{\diamond} = \mathbf{unit}$
- $\mathbf{nat}^{\diamond} = \mathbf{nat}$
- $(\vec{\sigma} \to \vec{\tau})^{\diamond} = \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}^{\diamond}; \mathbf{out}\ \vec{\tau}^{\diamond})$

Proposition 3.6. *(retraction at the type level).*

1. *For any type $\sigma \in \Sigma_{\mathbf{IS}}$, we have $\sigma^{\star\diamond} = \sigma$.*
2. *For any type $\sigma \in \Sigma^{\star}_{\mathbf{FS}}$, we have $\sigma^{\diamond\star} = \sigma$.*

Proof. Straightforward induction on the translations $\sigma^{\diamond}$ and $\sigma^{\star}$. □

Theorem 3.7. *For any environments Γ and Ω, any expression e, any sequence s we have:*

- $\Gamma; \Omega \vdash e : \tau$ in **IS** implies $\Gamma^{\star}, \Omega^{\star} \vdash e^{\star} : \tau^{\star}$ in **FS**.
- $\Gamma; \Omega \vdash s \rhd \Omega'$ in **IS** implies $\Gamma^{\star}, \Omega^{\star} \vdash (s)^{\star}_{\vec{z}} : \vec{\sigma}^{\star}$ for any $\vec{z} : \vec{\sigma}^{\star} \subset \Omega'^{\star}$ in **FS**.

Proof. By induction on the typing derivation. □

Theorem 3.8. *For any state (s, μ), if $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ in* **IS** *with $\vec{z} : \vec{\sigma} \subset \Omega$, then $\vdash (s)^{\star}_{\vec{z}}[\mu(\vec{x})^{\star}/\vec{x}] : \vec{\sigma}^{\star}$ in* **FS**.

Proof. By definition of state typing, $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ implies $\emptyset; \vec{z} : \vec{\tau} \vdash s \rhd \Omega$ and for all $(z_i, \mu(z_i)) \in \mu$, $\emptyset; \emptyset \vdash \mu(z_i) : \tau_i$. By Theorem 3.7, on one hand $\emptyset; \vec{z} : \vec{\tau} \vdash s \rhd \Omega$ implies $\vec{z} : \vec{\tau}^{\star} \vdash (s)^{\star}_{\vec{z}} : \vec{\sigma}^{\star}$, and on the other hand $\emptyset; \emptyset \vdash \mu(z_i) : \tau_i$ implies $\vdash \mu(z_i)^{\star} : \tau_i^{\star}$. Since $(s)^{\star}_{\vec{z}}$ is well typed in the environment $\vec{z} : \vec{\tau}^{\star}$, the variables in $\vec{x}$ which are not in $\vec{z}$ are not free in $(s)^{\star}_{\vec{z}}$. Hence, by the substitution lemma, $\vdash (s)^{\star}_{\vec{z}}[\mu(\vec{x})^{\star}/\vec{x}] : \vec{\sigma}^{\star}$. □

Theorem 3.9.

- *Given a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \vec{\sigma}$ in* **FS** *with $\Gamma, \vec{\sigma} \in \Sigma^{\star}_{\mathbf{FS}}$ and a fresh mutable variable tuple $\vec{r}$ of any type $\vec{\sigma}' \in \Sigma_{\mathbf{ID}}$ we have $\Gamma^{\diamond}; \vec{r} : \vec{\sigma}' \vdash t^{\diamond}_{\vec{r}} \rhd \vec{r} : \vec{\sigma}^{\diamond}$ in* **IS**.
- *Given a value $v \in \mathcal{V}$ such that $\Gamma \vdash v : \sigma$ in* **FS** *with $\Gamma, \sigma \in \Sigma^{\star}_{\mathbf{FS}}$, we have $\Gamma^{\diamond}; \vdash v^{\diamond} : \sigma^{\diamond}$ in* **IS**.

Proof. By induction on the typing derivation. □

3.4 Properties of the pseudo-dynamic type system

As expected, the "type preservation" and "progress" properties hold for the transition semantics.

Theorem 3.10. *(type preservation). For any state (s, μ), if $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ in* **IS** *and $(s, \mu) \mapsto (s', \mu')$ then there exists $\vec{\tau}'$ such that $\vec{z} : \vec{\tau}' \vdash (s', \mu') \rhd \Omega$, in the simple type system.*

Proof. See Appendix B.3. □

Lemma 3.11. *(progress). For any state (s, μ), if $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ in* **IS** *then either $s = \varepsilon$ and no more evaluation step can occur, or there is a unique state (s', μ') such that $(s, \mu) \mapsto (s', \mu')$.*

Proof. See Appendix B.4. □

Lemma 3.12. *(termination). For any state (s, μ), if $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ in* **IS** *then the evaluation of (s, μ) terminates.*

Proof. By contradiction, let us assume that there is an infinite sequence of evaluation steps of (s, μ). By Proposition 2.10, with the fact that there cannot be an infinite sequence of evaluation steps using only rule (S.VAR-I), we have an infinite sequence of evaluation steps of $(s)^{\star}_{\vec{z}}[\mu(\vec{x})^{\star}/\vec{x}]$. By Theorem 3.8, $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ implies $\vdash (s)^{\star}_{\vec{z}}[\mu(\vec{x})^{\star}/\vec{x}] : \vec{\sigma}^{\star}$ and since typable terms of system T are strongly normalizing, we have a contradiction. □

Proposition 3.13. *For any* (s,μ), Ω *and* $\vec{z}$, *if* $\vec{z}:\vec{\tau}\vdash(s,\mu)\rhd\Omega$ *in* **IS**, *then there is a unique store* μ' *such that* $(s,\mu)\mapsto^n(\varepsilon,\mu')$ *for some* n.

Proof. Since, by Lemma 3.12, no infinite evaluation of (s,μ) can occur, we prove the property by induction on the length n of the longest sequence of evaluation steps from (s,μ):

- $n=0$: we have (s,μ) as normal form; by Lemma 3.11, $s=\varepsilon$, hence we conclude with $\mu'=\mu$.
- $n>0$: by Lemma 3.11, $\vec{z}:\vec{\tau}\vdash(s,\mu)\rhd\Omega$ implies that there is a unique state (s',μ') such that $(s,\mu)\mapsto(s',\nu)$; by Theorem 3.10, $\vec{z}:\vec{\tau}\vdash(s,\mu)\rhd\Omega$ and $(s,\mu)\mapsto(s',\nu)$ implies $\vec{z}:\vec{\tau}'\vdash(s',\nu)\rhd\Omega$ for some $\vec{\tau}'$; by induction hypothesis, there is a unique store μ' such that $(s',\nu)\mapsto^{n-1}(\varepsilon,\mu')$; hence $(s,\mu)\mapsto(s',\nu)\mapsto^{n-1}(\varepsilon,\mu')$. □

3.5 Global variables

Recall that the imperative type systems **IS** and **ID** forbid any access to global mutable variables. It is straightforward to address this restriction by passing the global variable as an explicit **in out** parameter to each procedure declaration. The same variable is then given as argument for each procedure call. Moreover, an **in out** parameter can be encoded with one **in** parameter and one **out** parameter, where each procedure initialize the variable with its input value before executing its body. To handle more conveniently a list of global variables $\vec{z}$ we introduce the following abbreviations:

$$\begin{array}{rcl}\mathbf{proc}(\mathbf{in}\ \vec{x};\mathbf{out}\ \vec{y})_{\vec{z}}\{s\}_{\vec{y},\vec{z}} & = & \mathbf{proc}(\mathbf{in}\ \vec{x},\vec{z}\,';\mathbf{out}\ \vec{y},\vec{z})\{\vec{z}:=\vec{z}\,';s\}_{\vec{y},\vec{z}}\\ p(\vec{e};\vec{y})_{\vec{z}} & = & p(\vec{e},\vec{z};\vec{y},\vec{z})\end{array}$$

This transformation corresponds to the usual state-passing style transform in functional programming. Up to curryfication, we also obtain a state monad [Liang et al., 1995]. At the type level, however, since the type of a mutable variable can be changed by an assignment, this transform do not correspond to the usual state monad $\tau\ ST=\sigma\to(\tau\times\sigma)$ where σ is the fixed type of the global state. We obtain instead a parameterized state monad [Atkey, 2006], $(\sigma,\tau,\sigma')\ ST=\sigma\to(\tau\times\sigma')$ where σ is the input type of the global state and σ' is its output state.

This remark shows that the pseudo-dynamic type system is quite expressive and enables to type programs which would usually require an ad-hoc effect system [Talpin and Jouvelot, 1994].

4 Dependent Type Systems

In this section, we present the dependently-typed systems for languages **F** and **I**. As in the non-dependent case, we show that both translation * and $\diamond$ preserve typability. As a corollary, we obtain a soundness result (theorem 4.8) and a representation theorem (proposition 4.11) for dependently-typed imperative programs.

4.1 Functional dependent type system FD

Following the definition of **ML1P** [Leivant, 1990] (or similarly **IT**($\mathbb{N}$) in [Leivant, 2002]), we enrich language **F** with dependent types. The type system is parameterized by an equational system $\mathcal{E}$ which defines a set of functions in the style of Herbrand-Gödel. We consider only the sort **nat** (with constructors 0 and **s**), and we assume that $\mathcal{E}$ contains at least the usual defining equations for addition, multiplication and a predecessor function **p** (which is essential to derive all axioms of Peano's arithmetic [Leivant, 2002]). The syntax of formulas is the following:

$$\begin{array}{rcl}\tau & ::= & \mathbf{nat}(n)\\ & | & (n=m)\\ & | & \forall\vec{\imath}\,(\tau_1\Rightarrow\tau_2)\\ & | & \exists\vec{\imath}\,(\tau_1\wedge\ldots\wedge\tau_n)\end{array}$$

Note that first-order quantifiers are provided in the form of dependent products and dependent sums. As usual, implication and conjunction are recovered as special non-dependent cases (when $\vec{\imath}$ is empty). Similarly, relativized quantification $\forall x(\mathbf{nat}(x)\Rightarrow\varphi)$ and $\exists x(\mathbf{nat}(x)\wedge\varphi)$ are also obtained as special cases.

The functional dependent type system is summarized in Figure 4.1 (where $\top$ denotes $n = n$ for some n and $\vdash_{\mathcal{E}} n = m$ means that either $n = m$ or $m = n$ is an instance of $\mathcal{E}$).

The main difference between this type system and the deduction system **ML1P** described in [Leivant, 1990] comes from the fact that a derived sequent is directly annotated by a realizer (a functional term), whereas in [Leivant, 1990] an extraction function (or forgetful map) κ needs to be applied to the derivation to obtain the realizer. In other words, if Π is a derivation of a sequent $\Gamma \vdash \sigma$ in **ML1P**, then $\Gamma \vdash \kappa(\Pi) : \sigma$ is derivable in **FD**. Conversely, if Π is a derivation of $\Gamma \vdash t : \sigma$ in **FD**, then Π is also derivation of $\Gamma \vdash \sigma$ in **ML1P** (just remove the realizers from the derivation). Let us recall the subject reduction property of **ML1P** [Leivant, 1990] and derive the same property for **FS** as a corollary.

Theorem 4.1. *(subject reduction for* **ML1P***).*

- *If* Π *Prawitz-reduces to* Π'*, then* $\kappa\Pi$ *reduces to* $\kappa\Pi'$*.*
- *If* $t = \kappa\Pi$ *reduces to* t' *then* $t' = \kappa\Pi'$ *for some* Π' *such that* Π *Prawitz-reduces to* Π'*.*

Corollary 4.2. *(subject reduction for* **FD***). If* $\Gamma \vdash t : \sigma$ *in* **FD** *and* $t \rightsquigarrow t'$ *then* $\Gamma \vdash t' : \sigma$*.*

Proof. Let Π be a derivation of $\Gamma \vdash t : \sigma$ in **FD**, then $\kappa(\Pi) = t$ and Π is also a derivation of $\Gamma \vdash \sigma$ in **ML1P**. By the above theorem, if $t \rightsquigarrow t'$ then $t' = \kappa\Pi'$ for some derivation Π' of the same sequent $\Gamma \vdash \sigma$ in **ML1P**. Consequently, $\Gamma \vdash t' : \sigma$ is derivable since $t' = \kappa\Pi'$. □

Similarly, we obtain the representation theorem for **FD** as a corollary of the same property for **ML1P** [Leivant, 1990, Leivant, 2002].

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad (\textsc{ident})$$

$$\Gamma \vdash 0 : \mathbf{nat}(0) \quad (\textsc{zero})$$

$$\frac{\Gamma \vdash t : \mathbf{nat}(n)}{\Gamma \vdash S(t) : \mathbf{nat}(\mathbf{s}(n))} \quad (\textsc{succ})$$

$$\frac{\Gamma \vdash t : \mathbf{nat}(n)}{\Gamma \vdash \mathbf{pred}(t) : \mathbf{nat}(\mathbf{p}(n))} \quad (\textsc{pred})$$

$$\frac{\Gamma \vdash t_1 : \tau_1[\vec{m}/\vec{\imath}] \quad \ldots \quad \Gamma \vdash t_k : \tau_k[\vec{m}/\vec{\imath}]}{\Gamma \vdash (t_1, \ldots, t_k) : \exists \vec{\imath}\, (\tau_1 \wedge \ldots \wedge \tau_k)} \quad (\textsc{tuple})$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_k : \tau_k \vdash t : \tau \qquad \Gamma \vdash u : \exists \vec{\imath}\, (\tau_1 \wedge \ldots \wedge \tau_k)}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_k) = u\ \mathbf{in}\ t : \tau} \quad (\textsc{let})^\star$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \forall \vec{\imath}\, (\tau \Rightarrow \sigma)} \quad (\textsc{abs})^\star$$

$$\frac{\Gamma \vdash t_1 : \forall \vec{\imath}\, (\sigma \Rightarrow \tau) \quad \Gamma \vdash t_2 : \sigma[\vec{n}/\vec{\imath}]}{\Gamma \vdash t_1\ t_2 : \tau[\vec{n}/\vec{\imath}]} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat}(n) \quad \Gamma \vdash t_2 : \tau[\mathbf{0}/i] \quad \Gamma, x : \mathbf{nat}(i), y : \tau \vdash t_3 : \tau[\mathbf{s}(i)/i]}{\Gamma \vdash \mathbf{rec}(t_1, t_2, \lambda x.\lambda y.t_3) : \tau[n/i]} \quad (\textsc{rec})^\star$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma \vdash () : (n = m)} \quad (\textsc{equal})$$

$$\frac{\Gamma \vdash t : \tau[n/i] \quad \Gamma \vdash v : (n = m)}{\Gamma \vdash t : \tau[m/i]} \quad (\textsc{subst})$$

*where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$ in (ABS), $\vec{\imath} \notin \mathcal{FV}(\Gamma, \tau)$ in (LET) and $i \notin \mathcal{FV}(\Gamma)$ in (REC)

Figure 4.1. Functional dependent type system

Proposition 4.3. *(representation theorem for* **FD***) Given an equational system* $\mathcal{E}$ *and an n-ary function symbol* f*, if* $\vdash_{\mathcal{E}} t : \forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \mathbf{nat}(f(\vec{n}))$ *is derivable in* **FD** *then* t *represents* f*.*

Definition 4.4. *(forgetful map). For any functional dependent type* τ*, the computational content* $\kappa\tau$ *of* τ *is defined inductively as follows:*

- $\kappa(n = m) = \mathbf{unit}$
- $\kappa(\mathbf{nat}(n)) = \mathbf{nat}$
- $\kappa(\forall \vec{i}\,(\sigma \Rightarrow \tau)) = \kappa\sigma \to \kappa\tau$
- $\kappa(\exists \vec{i}\,(\tau_1 \wedge \ldots \wedge \tau_n) = \kappa\tau_1 \times \ldots \times \kappa\tau_n$

4.1.1 Example: the addition function

Recall the usual Peano's axiom for addition (see in appendix D the conventions we use in the examples):

$$\begin{array}{lrcl} (1) & x + 0 & = & x \\ (2) & x + \mathbf{s}(i) & = & \mathbf{s}(x + i) \end{array}$$

The proof of $\forall n(\mathbf{nat}(n) \Rightarrow \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n+m)))$ gives us a term of **F** that computes the addition of two natural numbers. Here follows, in a "pure" natural deduction style, the proof annotated by the terms of **F**.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : \mathbf{nat}(m)} \quad \dfrac{\dfrac{}{x : \mathbf{nat}(n)}}{x : \mathbf{nat}(n+0)}\text{by (1)} \quad \dfrac{\dfrac{\dfrac{}{z : \mathbf{nat}(n+u)}}{S(z) : \mathbf{nat}(\mathbf{s}(n+u))}}{S(z) : \mathbf{nat}(n + \mathbf{s}(u))}\text{by (2)}}{\mathbf{rec}(y, x, \lambda i.\lambda z.S(z)) : \mathbf{nat}(n+m)}}{\lambda y.\mathbf{rec}(y, x, \lambda i.\lambda z.S(z)) : \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n+m))}}{\lambda x.\lambda y.\mathbf{rec}(y, x, \lambda i.\lambda z.S(z)) : \forall n(\mathbf{nat}(n) \Rightarrow \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n+m)))}$$

4.2 Imperative dependent type system ID

As in the functional case, the type system is parameterized by equational system $\mathcal{E}$. The syntax of imperative dependent types is the following:

$$\sigma, \tau \ ::= \ \mathbf{nat}(n) \mid \mathbf{proc}\ \forall \vec{i}\,(\mathbf{in}\ \vec{\tau}; \exists \vec{j}\ \ \mathbf{out}\ \vec{\sigma}) \mid n = m$$

The dependent type system is summarized in Figure C.1 (where $\top$ denotes $n = n$ for some n and $\vdash_{\mathcal{E}} n = m$ means that either $n = m$ or $m = n$ is an instance of $\mathcal{E}$).

The store typing and the state typing are defined in the same way as for the pseudo-dynamic type system.

Definition. (store typing). *We say that a store* μ *is typable of output typing environment* $\Omega = z_1 : \tau_1, \ldots, z_n : \tau_n$*, denoted* $\mu \rhd \Omega$ *if and only if* $\vec{z} \in dom(\mu)$ *and for all* $(z_i, w_i) \in \mu$ *we have* $\emptyset; \emptyset \vdash w_i : \tau_i$*.*

Definition. (state typing). *We say that a state* $(s,\ \mu)$ *is typable of output typing environment* Ω' *for a restriction of the store to the variables* $\vec{z} : \vec{\tau}$*, which we write as* $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \exists \vec{\kappa}.\Omega'$*, if and only if* $\mu \rhd \vec{z} : \vec{\tau}$ *and* $\emptyset; \vec{z} : \vec{\tau} \vdash s \rhd \exists \vec{\kappa}.\Omega'$*.*

Definition 4.5. *(forgetful map). For any imperative dependent type* τ*, the computational content* $\kappa\tau$ *of* τ *is defined inductively as follows:*

- $\kappa(n = m) = \mathbf{unit}$
- $\kappa(\mathbf{nat}(n)) = \mathbf{nat}$
- $\kappa(\mathbf{proc}\ \forall \vec{i}\,(\mathbf{in}\ \vec{\sigma}; \exists \vec{j}\ \ \mathbf{out}\ \vec{\tau})) = \mathbf{proc}\ (\mathbf{in}\ \kappa\vec{\tau}; \mathbf{out}\ \kappa\vec{\sigma})$

$$\frac{x{:}\,\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x{:}\,\tau} \quad (\textsc{t.env})$$

$$\frac{}{\Gamma;\Omega \vdash \bar{q}{:}\,\mathbf{nat}(\mathbf{s}^q(\mathbf{0}))} \quad (\textsc{t.num})$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma;\Omega \vdash *{:}\,n = m} \quad (\textsc{t.equal})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}{:}\,\vec{\sigma}; \vec{z}{:}\,\vec{\top} \vdash s \rhd \exists \vec{\jmath}\,.\vec{z}{:}\,\vec{\tau}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y};\ \mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}}{:}\,\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\tau})} \quad (\textsc{t.proc})^*$$

$$\frac{\Gamma;\Omega \vdash e'{:}\,\tau[n/i] \qquad \Gamma;\Omega \vdash e{:}\,n = m}{\Gamma;\Omega \vdash e'{:}\,\tau[m/i]} \quad (\textsc{t.subst-i})$$

$$\frac{\Gamma;\Omega \vdash s \rhd \exists \vec{\kappa}\,.\Omega'[n/i] \qquad \Gamma;\Omega \vdash e{:}\,n = m}{\Gamma;\Omega \vdash s \rhd \exists \vec{\kappa}\,.\Omega'[m/i]} \quad (\textsc{t.subst-ii})$$

$$\frac{}{\Gamma;\Omega,\Omega'[\vec{n}/\vec{\kappa}] \vdash \varepsilon \rhd \exists \vec{\kappa}\,.\Omega'} \quad (\textsc{t.empty})$$

$$\frac{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash c \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\tau} \qquad \Gamma;\Omega,\vec{x}{:}\,\vec{\tau} \vdash s \rhd \exists \vec{\kappa}\,.\Omega'}{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash c;\ s \rhd \exists \vec{\kappa}\,.\Omega'} \quad (\textsc{t.seq})^*$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma, y{:}\,\tau;\Omega \vdash s \rhd \exists \vec{\kappa}\,.\Omega'}{\Gamma;\Omega \vdash \mathbf{cst}\ y = e;\ s \rhd \exists \vec{\kappa}\,.\Omega'} \quad (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma;\Omega, y{:}\,\tau \vdash s \rhd \exists \vec{\kappa}\,.\Omega' \qquad y \notin \Omega'}{\Gamma;\Omega \vdash \mathbf{var}\ y := e;\ s \rhd \exists \vec{\kappa}\,.\Omega'} \quad (\textsc{t.var})$$

$$\frac{\Gamma;\vec{x}{:}\,\vec{\tau} \vdash s \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\sigma}}{\Gamma;\Omega,\vec{x}{:}\,\vec{\tau} \vdash \{s\}_{\vec{x}} \rhd \exists \vec{\jmath}\,.\vec{x}{:}\,\vec{\sigma}} \quad (\textsc{t.block})$$

$$\frac{}{\Gamma;\Omega, y{:}\,\mathbf{nat}(n) \vdash \mathbf{inc}(y) \rhd y{:}\,\mathbf{nat}(\mathbf{s}(n))} \quad (\textsc{t.inc})$$

$$\frac{}{\Gamma;\Omega, y{:}\,\mathbf{nat}(n) \vdash \mathbf{dec}(y) \rhd y{:}\,\mathbf{nat}(\mathbf{p}(n))} \quad (\textsc{t.dec})$$

$$\frac{\Gamma;\Omega, y{:}\,\sigma \vdash e{:}\,\tau}{\Gamma;\Omega, y{:}\,\sigma \vdash y := e \rhd \Omega, y{:}\,\tau} \quad (\textsc{t.assign})$$

$$\frac{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma}[\mathbf{0}/i] \vdash e{:}\,\mathbf{nat}(n) \qquad \Gamma, y{:}\,\mathbf{nat}(i);\vec{x}{:}\,\vec{\sigma} \vdash s \rhd \vec{x}{:}\,\vec{\sigma}[\mathbf{s}(i)/i]}{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma}[\mathbf{0}/i] \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}} \rhd \vec{x}{:}\,\vec{\sigma}[n/i]} \quad (\textsc{t.for})^*$$

$$\frac{\Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash p{:}\,\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\tau}) \qquad \Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash \vec{e}{:}\,\vec{\sigma}[\vec{u}/\vec{\imath}]}{\Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash p(\vec{e};\vec{r}) \rhd \exists \vec{\jmath}\,.\vec{r}{:}\,\vec{\tau}[\vec{u}/\vec{\imath}]} \quad (\textsc{t.call})$$

*where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$ in (T.PROC) and $i \notin \mathcal{FV}(\Gamma)$ in (T.FOR)
and $\vec{\jmath} \notin \mathcal{FV}(\Gamma,\Omega)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega')$ in (T.SEQ)

Figure 4.2. Imperative dependent type system

Proposition 4.6. *(erasure). If* $\Gamma;\Omega \vdash s \rhd \exists \vec{\kappa}\,.\Omega'$ *is derivable in* **ID** *then* $\kappa\Gamma;\kappa\Omega \vdash s \rhd \kappa\Omega'$ *is derivable in* **IS**.

Proof. By induction on the typing derivation of $\Gamma;\Omega \vdash s \rhd \exists \vec{\kappa}\,.\Omega'$. □

4.2.1 Example: the addition procedure

Complete type derivations in **ID** are tedious. In the following examples, we prefer instead to provide only some type annotations on the right-hand side of the program. Although we did not formally define this syntax, we believe that it is self-explanatory and contains enough information to reconstruct the complete type derivation in **ID**. For instance, here is the procedure *add* which computes the addition together with the sketch of its type derivation:

```
cst add = proc (in X, Y; out Z) {          −(X: nat(x), Y: nat(y))[Z: ⊤]
    Z := X;                                 |   [Z: nat(x + 0)]        by (1)
    for I := 0 until Y {                    |   −(I: nat(i))[Z: nat(x + i)]
        inc(Z);                             |   |   [Z: nat(x + s(i))]      by (2)
    }_Z;                                    |   [Z: nat(x + y)]
}_Z                                        (add: proc ∀x, y(in nat(x), nat(y); out nat(x + y)))
```

4.2.2 Example: the Ackermann procedure

We recall the equations which define a variant the Ackermann function [Leivant, 2002]:

$$\begin{array}{lrcl} (1) & \mathbf{a}(0,n) & = & \mathbf{s}(n) \\ (2) & \mathbf{a}(\mathbf{s}(z),0) & = & \mathbf{s}(\mathbf{s}(0)) \\ (3) & \mathbf{a}(\mathbf{s}(z),\mathbf{s}(u)) & = & \mathbf{a}(z,\mathbf{a}(\mathbf{s}(z),u)) \end{array}$$

Similarly, from a proof of $\forall m, n(\mathbf{nat}(m) \wedge \mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(m,n)))$ in **FD** in monadic normal form, by applying translation $^\diamond$ by hand, we obtain a procedure which computes $\mathbf{a}(m,n)$ (the functional typing derivation is in Appendix D.3.1). Here is the definition of the procedure *ack* with its typing annotations.

```
cst ack = proc (in M, N; out Z) {          −(M: nat(m), N: nat(n))[Z: ⊤]
                                            |
    var G := proc (in Y; out P) {           |   −(Y: nat(y))[P: ⊤]
        P := Y;                             |   |   [P: nat(y)]
        inc(P);                             |   |   [P: nat(s(y))]
    }_P;                                    |   [G: proc ∀y(in nat(y); out nat(a(0, y)))]      by (1)
                                            |
    for I := 0 until M {                    |   −(I: nat(i))[G: proc ∀y(in nat(y); out nat(a(i, y)))]
        cst H = G;                          |   |   (H: proc ∀y(in nat(y); out nat(a(i, y))))
                                            |   |
        G := proc (in Y; out P) {           |   |   −(Y: nat(y))[P: ⊤]
            P := 2;                         |   |   |   [P: nat(a(s(i), 0))]      by (2)
            for J := 0 until Y {            |   |   |   −(J: nat(j))[P: nat(a(s(i), j))]
                H(P; P);                    |   |   |   |   [P: nat(a(s(i), s(j)))]      by (3)
            }_P;                            |   |   |   [P: nat(a(s(i), y))]
        }_P;                                |   |   [G: proc ∀y(in nat(y); out nat(a(s(i), y)))]
                                            |   |
    }_G;                                    |   [G: proc ∀y(in nat(y); out nat(a(m, y)))]
                                            |
    G(N; Z);                                |   [Z: a(m, n)]
}_Z                                        (ack: proc ∀m, n(in nat(m), nat(n); out nat(a(m, n))))
```

4.3 Translation from ID to FD

We show that translation * preserves dependent types.

Definition 4.7. *(translation of dependent types). For any imperative dependent type τ, the corresponding functional dependent type $\tau^\star$ is defined inductively as follows:*

- $(t = u)^\star = (t = u)$
- $(\mathbf{nat}(u))^\star = \mathbf{nat}(u)$
- $(\mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}))^\star = \forall \vec{\imath}\,(\vec{\tau}^\star \Rightarrow \exists \vec{\jmath}\,(\vec{\sigma}^\star))$

Theorem 4.8. *(Soundness for* **ID***). For any environments* Γ *and* Ω*, any expression* e*, any sequence* s *we have:*

- $\Gamma;\Omega\vdash e:\tau$ *in* **ID** *implies* $\Gamma^\star,\Omega^\star\vdash e^\star:\tau^\star$ *in* **FD**.
- $\Gamma;\Omega\vdash s\rhd\exists\vec{\jmath}.\vec{z}:\vec{\sigma}$ *in* **ID** *implies* $\Gamma^\star,\Omega^\star\vdash(s)^\star_{\vec{z}}:\exists\vec{\jmath}.\vec{\sigma}^\star$ *in* **FD**.

Notation 4.9. *If* $\vec{z}:\vec{\sigma}^\star=\Omega^\star$ *we write* $(\Omega^\star)_{\vec{z}}$ *for* $\sigma^\star_1\wedge\ldots\wedge\sigma^\star_n$.

Proof. See Appendix C.3. □

Theorem 4.10. *For any state* (s,μ)*, if* $\vec{z}:\vec{\tau}\vdash(s,\mu)\rhd\exists\vec{\kappa}.\Omega$ *in* **ID** *with* $\vec{z}:\vec{\sigma}\subset\Omega$ *then* $\vdash(s)^\star_{\vec{z}}[\mu(\vec{x})^\star/\vec{x}]:\exists\vec{\kappa}.\vec{\sigma}^\star$ *in* **FD**.

Proof. By definition of state typing, $\vec{z}:\vec{\tau}\vdash(s,\mu)\rhd\Omega$ implies $\emptyset;\vec{z}:\vec{\tau}\vdash s\rhd\exists\vec{\kappa}.\Omega$ and for all $(z_i,\mu(z_i))\in\mu$, $\emptyset;\emptyset\vdash\mu(z_i):\tau_i$. By theorem 4.8, on one hand $\emptyset;\vec{z}:\vec{\tau}\vdash s\rhd\exists\vec{\kappa}.\Omega$ implies $\vec{z}:\vec{\tau}^\star\vdash(s)^\star_{\vec{z}}:\exists\vec{\kappa}.\vec{\sigma}^\star$, and on the other hand $\emptyset;\emptyset\vdash\mu(z_i):\tau_i$ implies $\vdash\mu(z_i)^\star:\tau_i^\star$. Since $(s)^\star_{\vec{z}}$ is well typed in the environment $\vec{z}:\vec{\tau}^\star$, the variables in $\vec{x}$ which are not in $\vec{z}$ are not free in $(s)^\star_{\vec{z}}$. Hence, by the substitution lemma, $\vdash(s)^\star_{\vec{z}}[\mu(\vec{x})^\star/\vec{x}]:\exists\vec{\kappa}.\vec{\sigma}^\star$. □

4.4 Properties of dependently-typed imperative programs

We are now ready to state and prove the representation theorem for dependently-typed imperative program. This theorem is a corollary of the representation theorem for **FD** and the simulation theorem.

Corollary 4.11. *(representation theorem for* **ID***). Given an equational system* $\mathcal{E}$ *and an n-ary function symbol* f*, if* $\vdash p:\mathbf{proc}\ \forall\vec{n}(\mathbf{in}\ \mathbf{nat}(\vec{n});\mathbf{out}\ \mathbf{nat}(f(\vec{n})))$ *is derivable in* **ID** *then* p *represents* f.

Proof. Indeed, $\vdash p^\star:\forall\vec{n}.\mathbf{nat}(\vec{n})\Rightarrow\mathbf{nat}(f(\vec{n}))$ is derivable in **FD**, and thus $p^\star$ represents f by proposition 4.3. Since by Proposition 4.6,$\vdash p:\mathbf{proc}(\mathbf{in}\ \mathbf{nat};\mathbf{out}\ \mathbf{nat})$ is derivable in **IS**, we know that p always terminates by lemma 3.12 and computes $p^\star$ by proposition 2.10. □

4.5 Translation from FD to ID

We close this section by some properties of translation $^\diamond$. As in the non-dependent case, let us call $\Sigma^\star_{\mathbf{FD}}$ the image of $\Sigma_{\mathbf{ID}}$ by *. The sub-language $\Sigma^\star_{\mathbf{FD}}$ may clearly be characterized by the following grammar:

$$\sigma,\tau ::= \quad n=m \quad | \quad \mathbf{nat}(n) \quad | \quad \forall\vec{\imath}\,(\vec{\tau}\Rightarrow\exists\vec{\jmath}\,(\vec{\sigma}))$$

Definition 4.12. *For any type* $\sigma\in\Sigma^\star_{\mathbf{FD}}$ *the translation* $\sigma^\diamond$ *is defined as follows:*

- $(n=m)^\diamond=(n=m)$
- $(\mathbf{nat}(n))^\diamond=\mathbf{nat}(n)$
- $(\forall\vec{\imath}\,(\vec{\tau}\Rightarrow\exists\vec{\jmath}\,(\vec{\sigma})))^\diamond=\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond;\exists\vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}^\diamond)$

As expected, Proposition 3.6 is extended as follows.

Proposition 4.13. *(retraction).*

1. *For any type* $\sigma\in\Sigma_{\mathbf{ID}}$*, we have* $\sigma^{\star\diamond}=\sigma$.
2. *For any type* $\sigma\in\Sigma^\star_{\mathbf{FD}}$*, we have* $\sigma^{\diamond\star}=\sigma$.

Proof. Straightforward induction on translations $\sigma^\diamond$ and $\sigma^\star$. □

Proposition 4.14. *(erasure and translation commute). For any imperative dependent type* σ *we have* $\kappa(\sigma^\star)=(\kappa\sigma)^\star$.

Proof. Straightforward induction on types. □

Theorem 4.15.

- *Given a term $t \in \mathcal{L}$ such that $\Gamma \vdash t : \exists \vec{\kappa}.\vec{\sigma}$ in* **FD** *with $\Gamma, \vec{\sigma} \in \Sigma^{\star}_{\mathbf{FD}}$ and a fresh mutable variable tuple $\vec{r}$ of any type $\vec{\sigma}\,' \in \Sigma_{\mathbf{ID}}$ we have $\Gamma^{\diamond}; \vec{r} : \vec{\sigma}\,' \vdash t^{\diamond}_{\vec{r}} \rhd \exists \vec{\kappa}.\vec{r} : \vec{\sigma}^{\diamond}$ in* **ID**.
- *Given a value $v \in \mathcal{V}$ such that $\Gamma \vdash v : \sigma$ in* **FD** *with $\Gamma, \sigma \in \Sigma^{\star}_{\mathbf{FD}}$, we have $\Gamma^{\diamond}; \vdash v^{\diamond} : \sigma^{\diamond}$ in* **ID**.

Proof. See Appendix C.4. □

Remark 4.16. Translation $^{\diamond}$ is only defined above for term of $\mathcal{L}$. Translating an arbitrary term (typable in **FD**) into an imperative program (typable in **ID**), just requires to put the term in monadic normal form (and to encode tuples as thunks [Hatcliff and Danvy, 1997]). More details are given in Appendix C.5.

5 Control operators

In order to extend the imperative language **I** with non-local jumps, we first extend the functional language **F** with control operators. The resulting dependent type system $\mathbf{FD}^c$ corresponds thus to classical logic [Griffin, 1990] (Peano's arithmetic in fact). In this section, we rephrase known results from [Murthy, 1990, Murthy, 1991b] in our setting. However, since **FD** is based on Leivant's **ML1P**, our variant may seem closer to Parigot's type system for the $\lambda\mu$-calculus [Parigot, 1993a] (albeit in the second-order framework).

5.1 Functional dependent type system for control $\mathbf{FD}^c$

In order to extend **FD** to $\mathbf{FD}^c$, we assume the existence of a propositional constant "absurd" written $\bot$, we define the negation $\neg\varphi$ as an abbreviation for $\varphi \Rightarrow \bot$ and we add two constants **callcc** and **throw** with the following types:

$$\begin{aligned} \mathbf{callcc} &: (\neg\varphi \Rightarrow \varphi) \Rightarrow \varphi \\ \mathbf{throw} &: (\neg\varphi \wedge \varphi) \Rightarrow \psi \end{aligned}$$

This choice of control operators is taken from [Harper et al., 1993] but it would be equivalent to take for instance $\mathcal{A}$ and $\mathcal{C}$ from [Felleisen, 1987] as in [Murthy, 1990, Murthy, 1991b]. Note that do not consider any direct style semantics of these operators in this paper. Instead, we give an indirect semantics as a CPS-transformation [Plotkin, 1975].

5.2 CPS translation

As is well-known [Hatcliff and Danvy, 1994], it is natural to factor a CPS-transformation through Moggi's computational meta-language [Moggi, 1990, Moggi, 1991]. Since we are interested in providing a semantics for imperative programs and since the output of translation * is already a term in monadic normal form, the CPS-transformation needed is almost straightforward. We still have to be careful since in a dependent type system a monad is actually a modality [Coquand, 1996, Benton et al., 1998], and we have to deal with first-order quantifiers.

Following [Coquand, 1996], we write $\neg_o\varphi$ for $\varphi \Rightarrow o$ where o is a fixed propositional variable. The continuation monad ∇ is then defined as $\nabla\varphi = \neg_o\neg_o\varphi$ together with the following two abbreviations (which corresponds to *unit* and *bind*):

$$\begin{aligned} \mathbf{val}\ u &= \lambda z.(z\ u) \\ \mathbf{let\ val}\ x = u\ \mathbf{in}\ t &= \lambda z.(u\ \lambda x.(t\ z)) \end{aligned}$$

Moreover, in the continuation monad, control operators *callcc* and *throw* are definable as the following abbreviations [Nielsen, 2001]:

$$\begin{array}{rcl} \mathit{callcc} & = & \lambda h.\lambda k.(h\ k\ k) \\ \mathit{throw} & = & \lambda(k,a).\lambda k'.(k\ a) \end{array}$$

Let us now prove that for any monadic normal form (possibly containing **callcc** and **throw**) typable in $\mathbf{FD}^c$, its call-by-value CPS-transform is typable in **FD**. The translation of dependent types is defined as follows:

Definition 5.1. *(translation of dependent types from* $\mathbf{FD}^c$ *to* **FD***)*

$$\begin{array}{rcl} \mathbf{nat}(n)^\circ & = & \mathbf{nat}(n) \\ (n=m)^\circ & = & (n=m) \\ (\exists\vec{n}\,(\varphi_1\wedge\ldots\wedge\varphi_n))^\circ & = & \exists\vec{n}\,(\varphi_1^\circ\wedge\ldots\wedge\varphi_n^\circ) \\ (\forall\vec{n}\,(\varphi\Rightarrow\psi))^\circ & = & \forall\vec{n}\,(\varphi^\circ\Rightarrow\nabla\psi^\circ) \\ (\neg\varphi)^\circ & = & \neg_o\varphi^\circ \\ \bot^\circ & = & o \end{array}$$

Remark 5.2. If we instantiate the monad, and restrict ourselves to relativized quantifiers we obtain as expected Murthy's variant [Murthy, 1990, Murthy, 1991b] of Kuroda's translation [Kuroda, 1951].

Definition 5.3. *For any value* $v\in\mathcal{V}$ *and any term* $t\in\mathcal{L}$ *possibly containing* **callcc** *and* **throw***, the call-by-value CPS-transform* $v^\bullet$ *and* t° *are defined by mutual induction as follows:*

$$\begin{array}{rcl} ()^\bullet & = & () \\ x^\bullet & = & x \\ 0^\bullet & = & 0 \\ S(v)^\bullet & = & S(v^\bullet) \\ (\lambda x.u)^\bullet & = & (\lambda x.u^\circ) \\ (v_1,\ldots,v_k)^\bullet & = & (v_1^\bullet,\ldots,v_k^\bullet) \\ (\mathbf{callcc})^\bullet & = & \mathit{callcc} \\ (\mathbf{throw})^\bullet & = & \mathit{throw} \end{array}$$

$$\begin{array}{rcl} (v)^\circ & = & \mathbf{val}\ (v^\bullet) \\ (v_1\ v_2)^\circ & = & (v_1^\bullet\ v_2^\bullet) \\ (\mathbf{let}\ (x_1,\ldots,x_n)=t\ \mathbf{in}\ u)^\circ & = & \mathbf{let\ val}\ y=t^\circ\ \mathbf{in\ let}\ (x_1,\ldots,x_n)=y\ \mathbf{in}\ u^\circ \\ \mathbf{rec}(v,u,\lambda x.\lambda y.t)^\circ & = & \mathbf{rec}(v^\bullet,u^\circ,\lambda x.\lambda r.\mathbf{let\ val}\ y=r\ \mathbf{in}\ t^\circ) \\ \mathbf{pred}(v)^\circ & = & \mathbf{val\ pred}(v^\bullet) \end{array}$$

Remark 5.4. The translation above is defined for a syntax slightly more general than $\mathcal{L}$ since we only need here to distinguish values from computations. It is however straightforward to check that any term of $\mathcal{L}$ belongs to $\mathit{dom}(^\circ)$ and any value of $\mathcal{V}$ belongs to $\mathit{dom}(^\bullet)$.

Lemma 5.5. *The following typing rules are derivable in* **FD***:*

$$\frac{\Gamma\vdash u\colon\varphi}{\Gamma\vdash\mathbf{val}\ u\colon\nabla\varphi} \qquad\qquad \frac{\Gamma\vdash u\colon\nabla\varphi \qquad \Gamma,x\colon\varphi\vdash t\colon\nabla\psi}{\Gamma\vdash\mathbf{let\ val}\ x=u\ \mathbf{in}\ t\colon\nabla\psi}$$

Proof. See Appendix B. □

Lemma 5.6. *Abbreviations callcc and throw are typable in* **FD** *as follows:*

$$\begin{array}{rcl} \mathit{callcc} & : & ((\varphi^\circ\Rightarrow o)\Rightarrow\nabla\varphi^\circ)\Rightarrow\nabla\varphi^\circ \\ \mathit{throw} & : & ((\varphi^\circ\Rightarrow o)\wedge\varphi^\circ)\Rightarrow\nabla\psi^\circ \end{array}$$

Proof. See Appendix B. □

Lemma 5.7. *For any term t of $\mathcal{L}$ (resp. any value v of $\mathcal{V}$) possibly containing* **callcc** *and* **throw**, *if $\Gamma \vdash t : \varphi$ (resp. $\Gamma \vdash v : \varphi$) is derivable in* $\mathbf{FD}^c$ *then $\Gamma^\circ \vdash t^\circ : \nabla\varphi^\circ$ (resp. $\Gamma^\circ \vdash v^\bullet : \varphi^\circ$) is derivable in* **FD**.

Proof. By induction on the typing derivation where the basic cases for *callcc* and *throw* are obtained by Lemma 5.6:

- (IDENT)

$$\Gamma, x : \varphi \vdash x : \varphi$$

  Indeed,

$$\Gamma^\circ, x : \varphi^\circ \vdash x : \varphi^\circ$$

- (EQUAL)

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma \vdash () : (n = m)}$$

  Indeed,

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma^\circ \vdash () : (n = m)}$$

- (SUBST)

$$\frac{\Gamma \vdash t : \varphi[n/i] \qquad \Gamma \vdash v : (n = m)}{\Gamma \vdash t : \varphi[m/i]}$$

  Indeed,

$$\frac{\Gamma^\circ \vdash t^\circ : \nabla\varphi^\circ[n/i] \qquad \Gamma^\circ \vdash v^\bullet : (n = m)}{\Gamma^\circ \vdash t^\circ : \nabla\varphi^\circ[m/i]}$$

- (ZERO)

$$\Gamma \vdash 0 : \mathbf{nat}(0)$$

  Indeed,

$$\Gamma^\circ \vdash 0 : \mathbf{nat}(0)$$

- (SUCC)

$$\frac{\Gamma \vdash v : \mathbf{nat}(n)}{\Gamma \vdash S(v) : \mathbf{nat}(\mathbf{s}n)}$$

  Indeed,

$$\frac{\Gamma^\circ \vdash v^\bullet : \mathbf{nat}(n)}{\Gamma^\circ \vdash S(v^\bullet) : \mathbf{nat}(\mathbf{s}n)}$$

- (ABS) where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$

$$\frac{\Gamma, x : \varphi \vdash u : \psi}{\Gamma \vdash \lambda x.u : \forall\vec{\imath}\,(\varphi \Rightarrow \psi)}$$

  Indeed,

$$\frac{\Gamma^\circ, x : \varphi^\circ \vdash u^\circ : \nabla\psi^\circ}{\Gamma^\circ \vdash \lambda x.u^\circ : \forall\vec{\imath}\,(\varphi^\circ \Rightarrow \nabla\psi^\circ)}$$

- (APP)

$$\frac{\Gamma \vdash v_1 : \forall\vec{\imath}\,(\varphi \Rightarrow \psi) \qquad \Gamma \vdash v_2 : \varphi[\vec{n}/\vec{\imath}\,]}{\Gamma \vdash (v_1\ v_2) : \psi[\vec{n}/\vec{\imath}\,]}$$

  Indeed,

$$\frac{\Gamma \vdash v_1^\bullet : \forall\vec{\imath}\,(\varphi^\circ \Rightarrow \nabla\psi^\circ) \qquad \Gamma^\circ \vdash v_2^\bullet : \varphi^\circ[\vec{n}/\vec{\imath}\,]}{\Gamma \vdash (v_1^\bullet\ v_2^\bullet) : \nabla\psi^\circ[\vec{n}/\vec{\imath}\,]}$$

- (TUPLE)

$$\frac{\Gamma \vdash v_1 : \varphi_1[\vec{n}/\vec{\imath}\,] \quad \ldots \quad \Gamma \vdash v_k : \varphi_k[\vec{n}/\vec{\imath}\,]}{\Gamma \vdash (v_1, \ldots, v_k) : \exists\vec{\imath}\,(\varphi_1 \wedge \ldots \wedge \varphi_k)}$$

Indeed,

$$\frac{\Gamma^\circ \vdash v_1^\bullet : \varphi_1^\circ[\vec{n}/\vec{\imath}] \quad \ldots \quad \Gamma^\circ \vdash v_k^\bullet : \varphi_k^\circ[\vec{n}/\vec{\imath}]}{\Gamma^\circ \vdash (v_1^\bullet, \ldots, v_k^\bullet) : \exists \vec{\imath}\,(\varphi_1^\circ \wedge \ldots \wedge \varphi_k^\circ)}$$

- (LET) where $\vec{n} \notin \mathcal{FV}(\Gamma, \psi)$

$$\frac{\Gamma \vdash t : \exists \vec{\imath}\,(\varphi_1 \wedge \ldots \wedge \varphi_k) \quad \Gamma, x_1 : \varphi_1[\vec{n}/\vec{\imath}], \ldots, x_k : \varphi_k[\vec{n}/\vec{\imath}] \vdash u : \psi}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_k) = t\ \mathbf{in}\ u : \psi}$$

Indeed, since $\vec{n} \notin \mathcal{FV}(\Gamma^\circ, \psi^\circ)$

$$\frac{\Gamma^\circ \vdash t : \nabla(\exists \vec{\imath}\,(\varphi_1^\circ \wedge \ldots \wedge \varphi_k^\circ)) \qquad \dfrac{\Gamma^\circ \vdash t : \exists \vec{\imath}\,(\varphi_1^\circ \wedge \ldots \wedge \varphi_k^\circ) \quad \Gamma^\circ, x_1 : \varphi_1^\circ[\vec{n}/\vec{\imath}], \ldots, x_k : \varphi_k^\circ[\vec{n}/\vec{\imath}] \vdash u^\circ : \nabla\psi^\circ}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_k) = y\ \mathbf{in}\ u^\circ : \nabla\psi^\circ}}{\Gamma^\circ \vdash \mathbf{let\ val}\ y = t^\circ\ \mathbf{in\ let}\ (x_1, \ldots, x_k) = y\ \mathbf{in}\ u^\circ : \nabla\psi^\circ}$$

- (REC) where $i \notin \mathcal{FV}(\Gamma)$

$$\frac{\Gamma \vdash v : \mathbf{nat}(n) \quad \Gamma \vdash u : \varphi[0/i] \quad \Gamma, x : \mathbf{nat}(i), y : \varphi \vdash t : \varphi[\mathbf{s}(i)/i]}{\Gamma \vdash \mathbf{rec}(v, u, \lambda x.\lambda y.t) : \varphi[n/i]}$$

Indeed, since $z : \mathbf{nat}(n) \in \Gamma^\circ$ and $i \notin \mathcal{FV}(\Gamma^\circ)$

$$\frac{\Gamma^\circ \vdash v^\bullet : \mathbf{nat}(n) \quad \Gamma^\circ \vdash u^\circ : \nabla\varphi^\circ[0/i] \qquad \dfrac{\dfrac{\Gamma^\circ, x : \mathbf{nat}(i), r : \nabla\varphi^\circ \vdash r : \nabla\varphi^\circ \quad \Gamma^\circ, x : \mathbf{nat}(i), y : \varphi^\circ \vdash\ t^\circ : \nabla\varphi^\circ[\mathbf{s}(i)/i]}{\Gamma^\circ, x : \mathbf{nat}(i), r : \nabla\varphi^\circ \vdash \mathbf{let\ val}\ y = r\ \mathbf{in}\ t^\circ : \nabla\varphi^\circ[\mathbf{s}(i)/i]}}{\Gamma^\circ, x : \mathbf{nat}(i) \vdash \lambda r.\mathbf{let\ val}\ y = r\ \mathbf{in}\ t^\circ : \nabla\varphi^\circ \Rightarrow \nabla\varphi^\circ[\mathbf{s}(i)/i]}}{\Gamma^\circ \vdash \mathbf{rec}(v^\bullet, u^\circ, \lambda x.\lambda r.\mathbf{let\ val}\ y = r\ \mathbf{in}\ t^\circ) : \nabla\varphi^\circ[n/i]}$$

- (PRED)

$$\frac{\Gamma \vdash v : \mathbf{nat}(n)}{\Gamma \vdash \mathbf{pred}(v) : \mathbf{nat}(\mathbf{p}n)}$$

Indeed,

$$\frac{\dfrac{\Gamma^\circ \vdash v^\bullet : \mathbf{nat}(n)}{\Gamma^\circ \vdash \mathbf{pred}(v^\bullet) : \mathbf{nat}(\mathbf{p}n)}}{\Gamma^\circ \vdash \mathbf{val\ pred}(v^\bullet) : \nabla\mathbf{nat}(\mathbf{p}n)}$$

□

As a corollary of Lemma 5.7, we obtain a representation theorem for $\mathbf{FD}^c$.

Theorem 5.8. *(representation theorem for $\mathbf{FD}^c$). Given an equational system $\mathcal{E}$ and an n-ary function symbol f, if $\vdash t : \forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \mathbf{nat}(f(\vec{n}))$ is derivable in $\mathbf{FD}^c$ then t represents f.*

Proof. By Lemma 5.7 $\vdash t^\circ : \forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \nabla\mathbf{nat}(f(\vec{n})))$ is derivable in **FD**. Then, using Friedman's top level trick [Friedman, 1978, Murthy, 1990], we replace o by $\mathbf{nat}(f(\vec{n}))$ in the derivation, we obtain that $\vdash \lambda\vec{x}.(t^\circ\ \vec{x}\ id) : \forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \mathbf{nat}(f(\vec{n}))$ is also derivable in **FD**, and thus $\lambda\vec{x}.(t^\circ\ \vec{x}\ id)$ represents f. □

6 Non-local jumps

In this section we extend language **I** with control. Since control in imperative language are usually given in the form of several ad-hoc statements (such as exits from loops, exception handling, generators), there is no natural primitive statements. Consequently, we chose to retrofit operators **callcc** and **throw** to language **I**. We do not claim that these are natural control statement in an imperative language, but they primitive constructs which can be used to encode other statements as derived forms. This main advantage of this approach is that we derive immediately a sound program logic for imperative programs with control.

6.1 Dependent imperative type system with control $\mathbf{ID}^c$

Similarly to the functional case, we extend type system **ID** with a propositional type constant $\bot$, we define $\neg\vec{\sigma}$ as an abbreviation for **proc** (**in** $\vec{\sigma}$; **out** $\bot$), and we add to **ID** two primitive procedures **callcc** and **throw** with the following types:

$$\begin{aligned}\textbf{callcc} &: \textbf{proc}\,(\textbf{in proc}\,(\textbf{in}\,\neg\vec{\sigma};\textbf{out}\,\vec{\sigma});\textbf{out}\,\vec{\sigma})\\ \textbf{throw} &: \textbf{proc}\,(\textbf{in}\,\neg\vec{\sigma},\vec{\sigma};\textbf{out}\,\vec{\tau})\end{aligned}$$

Note that the type of **callcc** is exactly $((\neg\vec{\sigma}\Rightarrow\vec{\sigma})\Rightarrow\vec{\sigma})^\diamond$ and the type of **throw** is exactly $((\neg\vec{\sigma}\wedge\vec{\sigma})\Rightarrow\vec{\tau})^\diamond$. If we assume that **callcc** and **throw** are mapped by * to their functional counterpart, we have the following properties by construction:

Proposition 6.1. *For any environments Γ and Ω, any expression e, any sequence s, possibly containing procedures* **callcc** *and* **throw**, *we have:*

- $\Gamma;\Omega\vdash e:\tau$ *in* $\mathbf{ID}^c$ *implies* $\Gamma^\star,\Omega^\star\vdash e^\star:\tau^\star$ *in* $\mathbf{FD}^c$.
- $\Gamma;\Omega\vdash s\rhd\exists\vec{\jmath}.\vec{z}:\vec{\sigma}$ *in* $\mathbf{ID}^c$ *implies* $\Gamma^\star,\Omega^\star\vdash(s)^\star_{\vec{z}}:\exists\vec{\jmath}.\vec{\sigma}^\star$ *in* $\mathbf{FD}^c$.

Proposition 6.2.

- *Given a term $t\in\mathcal{L}$ possibly containing* **callcc** *and* **throw** *such that $\Gamma\vdash t:\exists\vec{\jmath}.\vec{\sigma}$ in* $\mathbf{FD}^c$ *and a fresh mutable variable tuple $\vec{r}$ of any type $\vec{\sigma}'\in\Sigma_{\mathbf{ID}}$ we have $\Gamma^\diamond;\vec{r}:\vec{\sigma}'\vdash t^\diamond_{\vec{r}}\rhd\exists\vec{\jmath}.\vec{r}:\vec{\sigma}^\diamond$ in* $\mathbf{ID}^c$.
- *Given a value $v\in\mathcal{V}$ possibly containing* **callcc** *and* **throw** *such that $\Gamma\vdash v:\sigma$ in* $\mathbf{FD}^c$ *for any environment Ω we have $\Gamma^\diamond;\Omega\vdash v^\diamond:\sigma^\diamond$ in* $\mathbf{ID}^c$.

Since our semantics of $\mathbf{ID}^c$ is indirect, no representation theorem for $\mathbf{ID}^c$ can be claimed. However, we still have the following corollary:

Corollary 6.3. *Given an equational system $\mathcal{E}$ and an n-ary function symbol f, if* $\vdash p:\textbf{proc}(\{\vec{n}\}\textbf{in nat}(\vec{n});\textbf{out nat}(f(\vec{n})))$ *is derivable in* $\mathbf{ID}^c$ *then $p^\star$ represents f.*

Proof. Since $\vdash p^\star:\forall\vec{n}.\mathbf{nat}(\vec{n})\Rightarrow\mathbf{nat}(f(\vec{n})))$ is derivable in $\mathbf{FD}^c$ and by Theorem 5.8, $p^\star$ represents f. □

6.2 Syntax and typing extensions with control operators

In order to get closer to some usual syntax for jumps in imperative language, we introduce the following two abbreviations:

$$\begin{aligned}k:\{s\}_{\vec{z}} &= \textbf{cst}\ \vec{z}\,'=\vec{z};\ \textbf{callcc}(\textbf{proc}(\textbf{in}\ k;\textbf{out}\ \vec{z})\{\vec{z}:=\vec{z}\,';\ s\}_{\vec{z}};\vec{z})\\ \textbf{jump}(k,\vec{e}\,)_{\vec{z}} &= \textbf{throw}(k,\vec{e};\vec{z})\end{aligned}$$

The first abbreviation corresponds to the declaration of a (first-class) label. Recall that our type systems requires that the current mutable variables be explicitly passed inside the body of the procedure, hence the constants declaration. The second abbreviation is a "jump with parameters" to *the end* of the block annotated with the label given as argument. Note that the output variables are important only for typing purpose (since the **jump** never returns), they are thus written as a subscript.

Proposition 6.4. *The following typing rules are derivable in* $\mathbf{ID}^c$.

$$\frac{\Gamma,k:\neg\vec{\sigma};\vec{z}:\vec{\tau}\vdash s\rhd\vec{z}:\vec{\sigma}\qquad\Gamma;\Omega,\vec{z}:\vec{\sigma}\vdash s'\rhd\Omega',\vec{z}:\vec{\sigma}\,'}{\Gamma;\Omega,\vec{z}:\vec{\tau}\vdash k:\{s\}_{\vec{z}};\ \ s'\rhd\Omega',\vec{z}:\vec{\sigma}\,'}$$

$$\frac{\Gamma;\Omega,\vec{z}:\vec{\tau}\vdash k:\neg\vec{\sigma}\qquad\Gamma;\Omega,\vec{z}:\vec{\tau}\vdash\vec{e}:\vec{\sigma}}{\Gamma;\Omega,\vec{z}:\vec{\tau}\vdash\textbf{jump}(k,\vec{e}\,)_{\vec{z}}\rhd\vec{z}:\vec{\tau}\,'}$$

Proof. See Appendix C.7. □

6.3 A generic example: encoding shift and reset

As a concluding example we show how to encode delimited continuation operators **shift** and **reset** [Danvy and Filinski, 1989] in $\mathbf{ID}^c$. This example is generic since it was shown by Filinski [Filinski, 1994, Filinski, 1999] that any representable monad can be encoded using **shift** and **reset**. [Filinski, 1994] also contains the proof that **shift** and **reset** can themselves be implemented using **callcc**, **throw** and one global mutable variable storing the meta-continuation. Filinski's implementation can thus be almost mechanically translated in $\mathbf{ID}^c$ (the type derivations are given in Appendix D.4):

$$
\begin{array}{lcl}
\textbf{reset} & : & \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \neg\alpha; \textbf{out}\ \beta, \neg\beta), \neg\gamma; \textbf{out}\ \alpha, \neg\gamma) \\
\textbf{reset} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{ \\
 & & \quad k\colon\{ \\
 & & \quad\quad \textbf{cst}\ m = mk; \\
 & & \quad\quad mk := \textbf{proc}(\textbf{in}\ r; \textbf{out}\ z)\{\textbf{jump}\ (k, r, m)_z;\}_z; \\
 & & \quad\quad \textbf{var}\ y;\ p(; y)_{mk}; \\
 & & \quad\quad \textbf{jump}\ (mk, y)_{r,mk}; \\
 & & \quad\}_{r,mk}; \\
 & & \}_{r,mk};
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{shift} & : & \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \textbf{proc}(\textbf{in}\ \alpha, \neg\beta; \textbf{out}\ \gamma, \neg\beta), \neg\delta; \textbf{out}\ \epsilon, \neg\epsilon), \neg\delta; \textbf{out}\ \alpha, \neg\gamma) \\
\textbf{shift} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{ \\
 & & \quad k\colon\{ \\
 & & \quad\quad \textbf{proc}\ q(\textbf{in}\ v; \textbf{out}\ r)_{mk}\{ \\
 & & \quad\quad\quad \textbf{reset}\ (\textbf{proc}(\textbf{out}\ z)_{mk}\{\textbf{jump}\ (k, v, mk)_{z,mk};\}_{z,mk}; r)_{mk}; \\
 & & \quad\quad \}_{r,mk}; \\
 & & \quad\quad \textbf{var}\ y;\ p(q; y)_{mk}; \\
 & & \quad\quad \textbf{jump}\ (mk, y)_{r,mk}; \\
 & & \quad\}_{r,mk}; \\
 & & \}_{r,mk}
\end{array}
$$

Of course, the image of those procedures by translation * yields functional terms typable in $\mathbf{FD}^c$. Those terms are given in Appendix E in Standard ML syntax [Milner et al., 1997]. The SML signature *CONT* is slightly different from [Harper et al., 1993] but they are equivalent (see [Filinski, 1994] for an implementation of a similar signature in SML/NJ [Appel and MacQueen, 1991]). Their functional types are reproduced here:

$$
\begin{array}{lcl}
\mathit{reset} & : & (\neg\alpha \Rightarrow \beta \wedge \neg\beta) \wedge \neg\gamma \Rightarrow \alpha \wedge \neg\gamma \\
\mathit{shift} & : & ((\alpha \wedge \neg\beta \Rightarrow \gamma \wedge \neg\beta) \wedge \neg\delta \Rightarrow \varepsilon \wedge \neg\varepsilon) \wedge \neg\delta \Rightarrow \alpha \wedge \neg\gamma
\end{array}
$$

These types could be made a little more readable by using a parameterised state monad. However, we recognize the type of **shift** and **reset** from [Danvy and Filinski, 1989] where $(\alpha \wedge \neg\sigma) \Rightarrow (\beta \wedge \neg\tau)$ is written in the form $\alpha/\tau \to \beta/\sigma$. We also refer the reader to [Wadler, 1994] for a detailed analysis of various type systems for **shift** and **reset** in the monadic framework and to [Ariola et al., 2007] for a type-theoretic study of delimited continuations.

Appendix A Properties of I and F untyped languages

A.1 Basic properties of I

Definition. *The sets* $\mathrm{FI}(s)$, $\mathrm{FI}(c)$ *and* $\mathrm{FI}(e)$ *of free identifiers (including both variable and constant identifiers) of a sequence, a command and an expression are defined by mutual induction as follows:*

- $\mathrm{FI}(y) = \{y\}$
- $\mathrm{FI}(*) = \mathrm{FI}(\bar{q}\,) = \emptyset$
- $\mathrm{FI}(\mathbf{proc}\ (\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{z}\,)\ \{s\}_{\vec{z}}) = \mathrm{FI}(s) \setminus (\vec{y} \cup \vec{z}\,)$

- $\mathrm{FI}(\mathbf{inc}(y)) = \mathrm{FI}(\mathbf{dec}(y)) = \{y\}$
- $\mathrm{FI}(\{s\}_{\vec{x}}) = \mathrm{FI}(s) \cup \vec{x}$
- $\mathrm{FI}(y := e) = \{y\} \cup \mathrm{FI}(e)$
- $\mathrm{FI}(p(\vec{e}\,; \vec{y}\,)) = \vec{y} \cup \mathrm{FI}(\vec{e}\,) \cup \mathrm{FI}(p)$
- $\mathrm{FI}(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}}) = \mathrm{FI}(\vec{e}\,) \cup (\mathrm{FI}(s) \setminus \{y\}) \cup \vec{x}$

- $\mathrm{FI}(\varepsilon) = \emptyset$
- $\mathrm{FI}(c\,;\ s) = \mathrm{FI}(c) \cup \mathrm{FI}(s)$
- $\mathrm{FI}(\mathbf{cst}\ y = e;\ s) = \mathrm{FI}(\mathbf{var}\ y := e;\ s) = \mathrm{FI}(\vec{e}\,) \cup (\mathrm{FI}(s) \setminus \{y\})$

A.2 Translation from F to I and retraction

Definition A.1. *We define inductively* $\mathcal{W}$, $\mathcal{V} = \cup_{n\in\mathbb{N}}\mathcal{V}_n$ *and* $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$, *where* $\mathcal{L}_n$ *(resp.* $\mathcal{V}_n$*) are indexed families of terms (resp. values) of* **F**, *as follows:*

- $x \in \mathcal{W}$
- $() \in \mathcal{W}$
- $S^n(0) \in \mathcal{W}$
- $\lambda\vec{x}.t \in \mathcal{W}$ *if* $t \in \mathcal{L}$
- $(w_1, ..., w_n) \in \mathcal{V}_n$ *if* $w_1, ..., w_n \in \mathcal{W}$
- $v \in \mathcal{L}_n$ *if* $v \in \mathcal{V}_n$
- $\mathbf{let}\ x = w\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- $\mathbf{let}\ x = \mathbf{succ}(w)\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- $\mathbf{let}\ x = \mathbf{pred}(w)\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$ *and* $u \in \mathcal{L}_n$
- $\mathbf{let}\ \vec{x} = w\ v\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$, $v \in \mathcal{V}_m$ *and* $u \in \mathcal{L}_n$
- $\mathbf{let}\ (x_1, ..., x_p) = \mathbf{rec}(w, v, \lambda y.\lambda(z_1, ..., z_p).t)\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $w \in \mathcal{W}$, $v \in \mathcal{V}_p$, $t \in \mathcal{L}_p$ *and* $u \in \mathcal{L}_n$
- $\mathbf{let}\ (x_1, ..., x_p) = t\ \mathbf{in}\ u \in \mathcal{L}_n$ *if* $t \in \mathcal{L}_p$ *and* $u \in \mathcal{L}_n$

Definition A.2. *We define the translation of any term* t *of* **F** *into a term* $t^\natural$ *of* $\mathcal{L}$ *by the following equations:*

$$\begin{aligned} x^\natural &= x \\ ()^\natural &= () \\ S^n(0)^\natural &= S^n(0) \\ S^n(t)^\natural &= \mathbf{let}\ x = t^\natural\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ ...\ \mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}\ x \end{aligned}$$

$$
\begin{array}{rcl}
(\lambda x.t)^\natural & = & \lambda x.t^\natural \\
\\
\mathbf{pred}(t)^\natural & = & \mathbf{let}\ x = t^\natural\ \mathbf{in\ let}\ x = \mathbf{pred}(x)\ \mathbf{in}\ x \\
\mathbf{rec}(t_1,t_2,t_3)^\natural & = & \mathbf{let}\ a = t_1^\natural\ \mathbf{in\ let}\ b = t_2^\natural\ \mathbf{in\ let}\ c = t_3^\natural\ \mathbf{in} \\
 & & \mathbf{let}\ z = \mathbf{rec}(a, b, \lambda x.\lambda y.\mathbf{let}\ d = c\ x\ \mathbf{in\ let}\ e = d\ y\ \mathbf{in}\ e)\ \mathbf{in}\ z \\
(t\ u)^\natural & = & \mathbf{let}\ x = t^\natural\ \mathbf{in\ let}\ y = u^\natural\ \mathbf{in\ let}\ r = x\ y\ \mathbf{in}\ r \\
(\mathbf{let}\ \vec{x} = u\ \mathbf{in}\ t)^\natural & = & \mathbf{let}\ y = u^\natural\ \mathbf{in\ let}\ \vec{x} = y\ ()\ \mathbf{in}\ t^\natural \\
(t_1, ..., t_n)^\natural & = & \mathbf{let}\ x_1 = t_1^\natural\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = t_n^\natural\ \mathbf{in}\ \lambda().(x_1, ..., x_n)
\end{array}
$$

Proposition A.3. *For any term t of* **F**, *we have $t^\natural \in \mathcal{L}$.*

Proof. Straightforward induction on t. □

Lemma A.4. *Given a term $t \in \mathcal{L}$ and a fresh mutable variable tuple $\vec{r}$ we have $\vec{r} \notin \mathrm{FV}(((t)^\diamond_{\vec{r}})^\star_{\vec{r}})$.*

Proof. By induction on t.

- $((\vec{w})^\diamond_r)^\star_r$
  $= (\vec{r} := \vec{w};)^\star_r$
  $= \mathbf{let}\ r_1 = (w_1^\diamond)^\star\ \mathbf{in}\ ...\ \mathbf{let}\ r_n = (w_n^\diamond)^\star\ \mathbf{in}\ \vec{r}$
  We easily conclude since $\vec{r}$ does not occur in $(\vec{w}^\diamond)^\star$.
- $((\mathbf{let}\ y = w\ \mathbf{in}\ u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{cst}\ y = w^\diamond;\ (u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= \mathbf{let}\ y = (w^\diamond)^\star\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  By induction hypothesis, $\vec{r} \notin \mathrm{FV}(((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$, and $\vec{r}$ does not occur in $(w^\diamond)^\star$.
- $((\mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{var}\ z := w^\diamond;\ \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{let}\ z = \mathbf{succ}(z)\ \mathbf{in\ let}\ y = z\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})[(w^\diamond)^\star/z]$
  $= (\mathbf{let}\ z = \mathbf{succ}((w^\diamond)^\star)\ \mathbf{in\ let}\ y = z\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$
  By induction hypothesis, $\vec{r} \notin \mathrm{FV}(((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$, and $\vec{r}$ does not occur in $(w^\diamond)^\star$.
- The case of **pred** is similar to **succ**.
- $((\mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda \vec{y}.t)\ \mathbf{in}\ u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z} := \vec{w};\ \mathbf{for}\ i := 0\ \mathbf{until}\ w^\diamond\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ (t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= \mathbf{let}\ \vec{z} = \mathbf{rec}((w^\diamond)^\star, \vec{z}, \lambda i.\lambda \vec{z}.\mathbf{let}\ \overrightarrow{y = z}\ \mathbf{in}\ ((t)^\diamond_{\vec{z}})^\star_{\vec{z}})\ \mathbf{in\ let}\ \overrightarrow{x = z}\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})[(\vec{w}^\diamond)^\star/\vec{z}]$
  $= \mathbf{let}\ \vec{z} = \mathbf{rec}((w^\diamond)^\star, (\vec{w}^\diamond)^\star, \lambda i.\lambda \vec{z}.\mathbf{let}\ \overrightarrow{y = z}\ \mathbf{in}\ ((t)^\diamond_{\vec{z}})^\star_{\vec{z}})\ \mathbf{in\ let}\ \overrightarrow{x = z}\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$
  By induction hypothesis, $\vec{r} \notin \mathrm{FV}(((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$, and $\vec{r}$ does not occur in $(w^\diamond)^\star$, $(\vec{w}^\diamond)^\star$ and $((t)^\diamond_{\vec{z}})^\star_{\vec{z}})$.
- $((\mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z};\ w^\diamond(\vec{w}^\diamond; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{let}\ \vec{z} = (w^\diamond)^\star\ (\vec{w}^\diamond)^\star\ \mathbf{in\ let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})[\vec{()}/\vec{z}]$
  $= \mathbf{let}\ \vec{z} = (w^\diamond)^\star\ (\vec{w}^\diamond)^\star\ \mathbf{in\ let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  By induction hypothesis, $\vec{r} \notin \mathrm{FV}(((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$, and $\vec{r}$ does not occur in $(w^\diamond)^\star$ and $(\vec{w}^\diamond)^\star$.
- $((\mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z};\ \{(t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}))^\star_{\vec{r}}$
  $= (\mathbf{let}\ \vec{z} = ((t)^\diamond_{\vec{z}})^\star_{\vec{z}}\ \mathbf{in\ let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}})[\vec{()}/\vec{z}]$
  $= \mathbf{let}\ \vec{z} = ((t)^\diamond_{\vec{z}})^\star_{\vec{z}}\ \mathbf{in\ let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^\diamond_{\vec{r}})^\star_{\vec{r}}$
  By induction hypothesis, $\vec{r} \notin \mathrm{FV}(((u)^\diamond_{\vec{r}})^\star_{\vec{r}})$, and $\vec{r}$ does not occur in $((t)^\diamond_{\vec{z}})^\star_{\vec{z}})$.

□

Definition A.5. *We define the reduction relation $\twoheadrightarrow$ as the reflexive, transitive and contextual closure of the reduction $\rightsquigarrow$ for arbitrary contexts.*

Proposition. *We prove the following properties, which clearly implies $((t)^{\diamond}_{r})^{\star}_{r} \approx t$ and if $w = S^n(0)$ or $w = *$ then $w^{\star\diamond} = w$ else $w^{\star\diamond} \approx w$.*

- *Given a term $t \in \mathcal{L}$ and a fresh mutable variable r we have $((t)^{\diamond}_{r})^{\star}_{r} \twoheadrightarrow t$.*
- *Given a value $v \in \mathcal{W}$, if $w = S^n(0)$ or $w = *$ then $w^{\star\diamond} = w$ else $w^{\star\diamond} \twoheadrightarrow w$.*

Proof. By mutual induction.

- $(S^n(0)^{\diamond})^{\star} = \bar{n}^{\star} = S^n(0)$.
- $(y^{\diamond})^{\star} = y^{\star} = y$.
- $(()^{\diamond})^{\star} = *^{\star} = ()$.
- $((\lambda\vec{x}.t)^{\diamond})^{\star}$
  $= (\mathbf{proc}(\mathbf{in}\ \vec{x}; \mathbf{out}\ \vec{z})\ \{(t)^{\diamond}_{\vec{z}}\}_{\vec{z}})^{\star}$
  $= \lambda\vec{x}.((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}[()/\vec{z}]$
  $= \lambda\vec{x}.((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}$ since $\vec{z} \notin \mathrm{FV}(((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}})$ by Lemma A.4
  $\twoheadrightarrow \lambda\vec{x}.t$ by induction hypothesis.
- $((\vec{w})^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\vec{r} := \vec{w}^{\diamond};)^{\star}_{\vec{r}}$
  $= \mathbf{let}\ r_1 = (w_1^{\diamond})^{\star}\ \mathbf{in}\ \ldots\ \mathbf{let}\ r_n = (w_n^{\diamond})^{\star}\ \mathbf{in}\ \vec{r}$
  $\rightsquigarrow^n (\vec{w}^{\diamond})^{\star}$
  $\twoheadrightarrow \vec{w}$ by induction hypothesis.
- $((\mathbf{let}\ y = w\ \mathbf{in}\ u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{cst}\ y = w^{\diamond};\ (u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= \mathbf{let}\ y = (w^{\diamond})^{\star}\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $\twoheadrightarrow \mathbf{let}\ y = w\ \mathbf{in}\ u$ by induction hypothesis.
- $((\mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{var}\ z := w^{\diamond};\ \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{let}\ z = \mathbf{succ}(z)\ \mathbf{in}\ \mathbf{let}\ y = z\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})[(w^{\diamond})^{\star}/z]$
  $= (\mathbf{let}\ z = \mathbf{succ}((w^{\diamond})^{\star})\ \mathbf{in}\ \mathbf{let}\ y = z\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})$
  $\rightsquigarrow (\mathbf{let}\ y = \mathbf{succ}((w^{\diamond})^{\star})\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})$ since $z \notin \mathrm{FV}((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})$
  $\twoheadrightarrow \mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u$ by induction hypothesis.
- The case of **pred** is similar to **succ**.
- $((\mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda\vec{y}.t)\ \mathbf{in}\ u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z} := \vec{w};\ \mathbf{for}\ i := 0\ \mathbf{until}\ w^{\diamond}\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ (t)^{\diamond}_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= \mathbf{let}\ \vec{z} = \mathbf{rec}((w^{\diamond})^{\star}, \vec{z}, \lambda i.\lambda\vec{z}.\mathbf{let}\ \overrightarrow{y = z}\ \mathbf{in}\ ((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}})\ \mathbf{in}\ \mathbf{let}\ \overrightarrow{x = z}\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})[(\vec{w}^{\diamond})^{\star}/\vec{z}]$
  $= \mathbf{let}\ \vec{z} = \mathbf{rec}((w^{\diamond})^{\star}, (\vec{w}^{\diamond})^{\star}, \lambda i.\lambda\vec{z}.\mathbf{let}\ \overrightarrow{y = z}\ \mathbf{in}\ ((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}})\ \mathbf{in}\ \mathbf{let}\ \overrightarrow{x = z}\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})$
  $\twoheadrightarrow \mathbf{let}\ \vec{z} = \mathbf{rec}((w^{\diamond})^{\star}, (\vec{w}^{\diamond})^{\star}, \lambda i.\lambda\vec{z}.((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}[\vec{z}/\vec{y}])\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})[\vec{z}/\vec{x}]$
  $\twoheadrightarrow \mathbf{let}\ \vec{z} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda\vec{z}.t[\vec{z}/\vec{y}])\ \mathbf{in}\ u[\vec{z}/\vec{x}]$ by induction hypothesis
  $= \mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda\vec{y}.t)\ \mathbf{in}\ u$ modulo α-conversion.
- $((\mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z};\ w^{\diamond}(\vec{w}^{\diamond}; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{let}\ \vec{z} = (w^{\diamond})^{\star}\ (\vec{w}^{\diamond})^{\star}\ \mathbf{in}\ \mathbf{let}\ x_1 = z_1\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})[\vec{()}/\vec{z}]$
  $= \mathbf{let}\ \vec{z} = (w^{\diamond})^{\star}\ (\vec{w}^{\diamond})^{\star}\ \mathbf{in}\ \mathbf{let}\ x_1 = z_1\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $\twoheadrightarrow \mathbf{let}\ \vec{z} = (w^{\diamond})^{\star}\ (\vec{w}^{\diamond})^{\star}\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}[\vec{z}/\vec{x}]$

$\twoheadrightarrow \mathbf{let}\ \vec{z} = w\ \vec{w}\ \mathbf{in}\ u[\vec{z}/\vec{x}]$ by induction hypothesis
$= \mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u$ modulo α-conversion.

- $((\mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}$
  $= (\mathbf{var}\ \vec{z};\ \{(t)^{\diamond}_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^{\diamond}_{\vec{r}}))^{\star}_{\vec{r}}$
  $= (\mathbf{let}\ \vec{z} = ((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}\ \mathbf{in}\ \mathbf{let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})[\vec{()}/\vec{z}]$
  $= (\mathbf{let}\ \vec{z} = ((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}\ \mathbf{in}\ \mathbf{let}\ x_1 = z_1\ \mathbf{in}\ ...\ \mathbf{let}\ x_n = z_n\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}})$
  $\twoheadrightarrow \mathbf{let}\ \vec{z} = ((t)^{\diamond}_{\vec{z}})^{\star}_{\vec{z}}\ \mathbf{in}\ ((u)^{\diamond}_{\vec{r}})^{\star}_{\vec{r}}[\vec{z}/\vec{x}]$
  $\twoheadrightarrow \mathbf{let}\ \vec{z} = t\ \mathbf{in}\ u[\vec{z}/\vec{x}]$ by induction hypothesis
  $= \mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u$ modulo α-conversion.

□

Appendix B Properties of IS and FS

B.1 Sequence-directed pseudo-dynamic type system

We present first a different (but equivalent) formulation of the pseudo-dynamic type system which is easier to deal with when proving properties by induction on sequences:

$$\frac{x{:}\,\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x{:}\,\tau} \quad (\textsc{t.env})$$

$$\frac{}{\Gamma;\Omega \vdash \bar{q}{:}\,\mathbf{nat}} \quad (\textsc{t.num})$$

$$\frac{}{\Gamma;\Omega \vdash *{:}\,\mathbf{unit}} \quad (\textsc{t.unit})$$

$$\frac{}{\Gamma;\Omega,\Omega' \vdash \varepsilon \rhd \Omega'} \quad (\textsc{t.empty})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma, y{:}\,\tau;\Omega \vdash s \rhd \Omega'}{\Gamma;\Omega \vdash \mathbf{cst}\ y = e;\ s \rhd \Omega'} \quad (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega \vdash e{:}\,\tau \qquad \Gamma;\Omega, y{:}\,\tau \vdash s \rhd \Omega' \qquad y \notin \Omega'}{\Gamma;\Omega \vdash \mathbf{var}\ y := e;\ s \rhd \Omega'} \quad (\textsc{t.var})$$

$$\frac{\Gamma;\vec{x}{:}\,\vec{\sigma} \vdash s \rhd \vec{x}{:}\,\vec{\tau} \qquad \Gamma;\Omega,\vec{x}{:}\,\vec{\tau} \vdash s' \rhd \Omega'}{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash \{s\}_{\vec{x}};\ s' \rhd \Omega'} \quad (\textsc{t.block})$$

$$\frac{\Gamma;\Omega, y{:}\,\mathbf{nat} \vdash s \rhd \Omega'}{\Gamma;\Omega, y{:}\,\mathbf{nat} \vdash \mathbf{inc}(y);\ s \rhd \Omega'} \quad (\textsc{t.inc})$$

$$\frac{\Gamma;\Omega, y{:}\,\mathbf{nat} \vdash s \rhd \Omega'}{\Gamma;\Omega, y{:}\,\mathbf{nat} \vdash \mathbf{dec}(y);\ s \rhd \Omega'} \quad (\textsc{t.dec})$$

$$\frac{\Gamma;\Omega, y{:}\,\sigma \vdash e{:}\,\tau \qquad \Gamma;\Omega, y{:}\,\tau \vdash s \rhd \Omega'}{\Gamma;\Omega, y{:}\,\sigma \vdash y := e;\ s \rhd \Omega'} \quad (\textsc{t.assign})$$

$$\frac{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash e{:}\,\mathbf{nat} \quad \Gamma, y{:}\,\mathbf{nat};\vec{x}{:}\,\vec{\sigma} \vdash s \rhd \vec{x}{:}\,\vec{\sigma} \quad \Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash s' \rhd \Omega'}{\Gamma;\Omega,\vec{x}{:}\,\vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ s' \rhd \Omega'} \quad (\textsc{t.for})$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}{:}\,\vec{\sigma};\vec{z}{:}\,\overrightarrow{\mathbf{unit}} \vdash s \rhd \vec{z}{:}\,\vec{\tau}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}}{:}\,\mathbf{proc}\ (\mathbf{in}\ \vec{\sigma};\mathbf{out}\ \vec{\tau})} \quad (\textsc{t.proc})$$

$$\frac{\Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash p{:}\,\mathbf{proc}\ (\mathbf{in}\ \vec{\tau};\mathbf{out}\ \vec{\sigma}) \quad \Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash \vec{e}{:}\,\vec{\tau} \quad \Gamma;\Omega,\vec{r}{:}\,\vec{\sigma} \vdash s \rhd \Omega'}{\Gamma;\Omega,\vec{r}{:}\,\vec{\omega} \vdash p(\vec{e}\,;\vec{r}\,);\ s \rhd \Omega'} \quad (\textsc{t.call})$$

Figure B.1. Imperative type system

B.2 Preliminary properties for substitutions and typing

Lemma B.1. *If* $\Gamma, x{:}\,\tau;\Omega \vdash s \rhd \Omega'$ *and* $\emptyset;\emptyset \vdash e{:}\,\tau$ *in* **IS** *then* $\Gamma;\Omega \vdash s[x \leftarrow e] \rhd \Omega'$ *in* **IS**.

Proof. Straightforward induction on s. □

Lemma B.2. *If* $\Gamma;x{:}\,\tau,\Omega \vdash s \rhd x{:}\,\sigma,\Omega'$ *in* **IS** *then* $\Gamma;y{:}\,\tau,\Omega \vdash s[x \hookleftarrow y] \rhd y{:}\,\sigma,\Omega'$ *in* **IS**.

Proof. Straightforward induction on s. □

Lemma B.3. *If* $\Gamma;\Omega \vdash s \rhd \Omega'$ *in* **IS** *then for any* $x{:}\,\sigma$, $\Gamma, x{:}\,\sigma;\Omega \vdash s \rhd \Omega'$ *and* $\Gamma;\Omega, x{:}\,\sigma \vdash s \rhd \Omega'$ *in* **IS**.

Proof. Straightforward by induction on the typing derivation. □

Lemma B.4. *If $\mu \rhd \Omega$ and $\emptyset; \Omega \vdash e : \tau$ in* **IS** *and $e =_\mu w$, then we have $\emptyset; \emptyset \vdash w : \tau$ in* **IS**.

Proof. The case $e = w$ is trivial and if e is some variable $x \in \Omega$ then by definition of $\mu \rhd \Omega$, we have $\emptyset; \emptyset \vdash \mu(x) = w : \tau$. □

B.3 Typing preservation by reduction

Theorem. *For any state (s, μ), Ω and $\vec{z}$, if $\vec{z} : \vec{\tau} \vdash (s, \mu) \rhd \Omega$ in* **IS** *and $(s, \mu) \mapsto (s', \mu')$ then there exists $\vec{\tau}'$ such that $\vec{z} : \vec{\tau}' \vdash (s', \mu') \rhd \Omega$ in* **IS**.

Proof. By induction on the derivation of $(s, \mu) \mapsto (s', \mu')$, and then by analysis of the typing derivation.

- (S.BLOCK-I): we have $\mu \rhd \Delta, \vec{x} : \vec{\tau}$ and

$$\frac{\overline{\emptyset; \vec{x} : \vec{\tau} \vdash \varepsilon \rhd \vec{x} : \vec{\tau}} \quad \emptyset; \Delta, \vec{x} : \vec{\tau} \vdash s \rhd \Delta'}{\emptyset; \Delta, \vec{x} : \vec{\tau} \vdash \{\}_{\vec{x}};\ s \rhd \Delta'}$$

  then we get $\emptyset; \Delta, \vec{x} : \vec{\tau} \vdash s \rhd \Delta'$ hence $\mu \rhd \Delta, \vec{x} : \vec{\tau}$ and $\Delta, \vec{x} : \vec{\tau} \vdash (s, \mu) \rhd \Delta'$.

- (S.BLOCK-II): we have $\mu \rhd \Delta, \vec{x} : \vec{\sigma}$

$$\frac{\emptyset; \vec{x} : \vec{\sigma} \vdash s_1 \rhd \vec{x} : \vec{\tau} \quad \emptyset; \Delta, \vec{x} : \vec{\tau} \vdash s_2 \rhd \Delta'}{\emptyset; \Delta, \vec{x} : \vec{\sigma} \vdash \{s_1\}_{\vec{x}};\ s_2 \rhd \Delta'}$$

  By induction hypothesis on $\vec{x} : \vec{\sigma} \vdash (s_1, \mu) \rhd \vec{x} : \vec{\tau}$, we obtain $\vec{x} : \vec{\sigma}' \vdash (s_1', \mu') \rhd \vec{x} : \vec{\tau}$ which gives us $\emptyset; \vec{x} : \vec{\sigma}' \vdash s_1' \rhd \vec{x} : \vec{\tau}$ and $\mu' \rhd \Delta, \vec{x} : \vec{\sigma}'$. We can build the following typing derivation to conclude:

$$\frac{\emptyset; \vec{x} : \vec{\sigma}' \vdash s_1' \rhd \vec{x} : \vec{\tau} \quad \emptyset; \Delta, \vec{x} : \vec{\tau} \vdash s \rhd \Delta'}{\emptyset; \Delta, \vec{x} : \vec{\sigma}' \vdash \{s_1'\}_{\vec{x}};\ s_2 \rhd \Delta'}$$

- (S.VAR-I): we have $\mu \rhd \Omega$

$$\frac{\emptyset; \Omega \vdash e : \tau \qquad \overline{\emptyset; \Omega, y : \tau \vdash \varepsilon \rhd \Omega}}{\emptyset; \Omega \vdash \mathbf{var}\ y := e;\ \varepsilon \rhd \Omega}$$

  then we get $\emptyset; \Omega \vdash \varepsilon \rhd \Omega$.

- (S.VAR-II): we have $\mu \rhd \Delta$ and

$$\frac{\emptyset; \Delta \vdash e : \tau \quad \emptyset; \Delta, y : \tau \vdash s \rhd \Omega \quad y \notin \Omega}{\emptyset; \Delta \vdash \mathbf{var}\ y := e;\ s \rhd \Omega}$$

  By Lemma B.4, $\mu \rhd \Delta$ and $\emptyset; \Delta \vdash e : \tau$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w : \tau$. By definition of store typing, $(\mu, y \leftarrow w) \rhd \Delta, y : \tau$. By induction hypothesis, since $\Delta, y : \tau \vdash (s, (\mu, y \leftarrow w)) \rhd \Omega$ is derivable, we obtain $\Gamma, y : \sigma \vdash (s', (\mu', y \leftarrow w')) \rhd \Omega$ which implies $\emptyset; \Gamma, y : \sigma \vdash s' \rhd \Omega$ with $(\mu', y \leftarrow w') = \Gamma, y : \sigma$. This last assertion trivially implies $\emptyset; \Gamma \vdash w' : \sigma$ by definition of store typing. We can then build the following typing derivation to conclude:

$$\frac{\emptyset; \Gamma \vdash w' : \sigma \quad \emptyset; \Gamma, y : \sigma \vdash s' \rhd \Omega \quad y \notin \Omega}{\emptyset; \Gamma \vdash \mathbf{var}\ y := w';\ s' \rhd \Omega}$$

- (S.ASSIGN): we have $\mu \rhd \Delta, y : \sigma$ and

$$\frac{\emptyset; \Delta, y : \sigma \vdash e : \tau \quad \emptyset; \Delta, y : \tau \vdash s \rhd \Delta'}{\emptyset; \Delta, y : \sigma \vdash y := e;\ s \rhd \Delta'}$$

  then we get $\emptyset; \Delta, y : \tau \vdash s \rhd \Delta'$. By Lemma B.4, $\mu \rhd \Delta, y : \sigma$ and $\emptyset; \Delta, y : \sigma \vdash e : \tau$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w : \tau$. Then, by definition of store typing, we obtain $\mu[y \leftarrow w] \rhd \Delta, y : \tau$.

- (S.INC): we have $\mu \rhd \Delta, y\colon \mathbf{nat}$ and
$$\frac{\emptyset; \Delta, y\colon \mathbf{nat} \vdash s \rhd \Delta'}{\emptyset; \Delta, y\colon \mathbf{nat} \vdash \mathbf{inc}(y);\ s \rhd \Delta'}$$
then
$$\frac{\overline{\emptyset; \Delta, y\colon \mathbf{nat} \vdash \overline{q+1}\colon \mathbf{nat}} \qquad \emptyset; \Delta, y\colon \mathbf{nat} \vdash s \rhd \Delta'}{\emptyset; \Delta, y\colon \mathbf{nat} \vdash y := \overline{q+1};\ s \rhd \Delta'}$$
- (S.DEC): similar to above.
- (S.CALL): we have $\mu \rhd \Delta, \vec{r}\colon \vec{\omega}$ and
$$\frac{\emptyset; \Delta, \vec{r}\colon \vec{\omega} \vdash p\colon \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \quad \emptyset; \Delta, \vec{r}\colon \vec{\omega} \vdash \vec{e}\colon \vec{\tau} \quad \emptyset; \Delta, \vec{r}\colon \vec{\sigma} \vdash s \rhd \Delta'}{\emptyset; \Delta, \vec{r}\colon \vec{\omega} \vdash p(\vec{e}\,, \vec{r});\ s \rhd \Delta'}$$
By Lemma B.4, $\mu \rhd \Delta, \vec{r}\colon \vec{\omega}$ and $\emptyset; \Delta, \vec{r}\colon \vec{\omega} \vdash \vec{e}\colon \vec{\tau}$ and $\vec{e} =_\mu \vec{w}$ implies $\emptyset; \emptyset \vdash \vec{w}\colon \vec{\tau}$. Still by Lemma B.4, $\mu \rhd \Delta, \vec{r}\colon \vec{\omega}$ and $\emptyset; \Delta, \vec{r}\colon \vec{\omega} \vdash p\colon \mathbf{proc}(\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})$ and $p =_\mu \mathbf{proc}(\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{x})\{s'\}_{\vec{x}}$ implies $\emptyset; \emptyset \vdash \mathbf{proc}(\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{x})\{s\}_{\vec{x}}\colon \mathbf{proc}(\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})$, that is
$$\frac{\vec{z} \neq \emptyset \qquad \emptyset; \vec{y}\colon \vec{\sigma}; \vec{x}\colon \overrightarrow{\mathbf{unit}} \vdash s' \rhd \vec{x}\colon \vec{\sigma}}{\emptyset; \emptyset \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{x})\{s'\}_{\vec{x}}\colon \mathbf{proc}(\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})}$$
By Lemmas B.1 and B.2, $\emptyset; \vec{y}\colon \vec{\sigma}; \vec{x}\colon \overrightarrow{\mathbf{unit}} \vdash s' \rhd \vec{x}\colon \vec{\sigma}$ and $\emptyset; \emptyset \vdash \vec{w}\colon \vec{\tau}$ implies $\emptyset; \vec{r}\colon \overrightarrow{\mathbf{unit}} \vdash s'[\vec{y} \leftarrow \vec{w}][\vec{x} \hookleftarrow \vec{r}\,] \rhd \vec{r}\colon \vec{\sigma}$. By definition of store typing, we have $\mu[\vec{r} \leftarrow *] \rhd \Delta, \vec{r}\colon \overrightarrow{\mathbf{unit}}$ and we can then build the following typing derivation to conclude:
$$\frac{\emptyset; \vec{r}\colon \overrightarrow{\mathbf{unit}} \vdash s'[\vec{y} \leftarrow \vec{w}][\vec{x} \hookleftarrow \vec{r}\,] \rhd \vec{r}\colon \vec{\sigma} \quad \emptyset; \Delta, \vec{r}\colon \vec{\sigma} \vdash s \rhd \Delta'}{\emptyset; \Delta, \vec{r}\colon \overrightarrow{\mathbf{unit}} \vdash \{s'[\vec{y} \leftarrow \vec{w}][\vec{x} \hookleftarrow \vec{r}\,]\}_{\vec{x}};\ s \rhd \Delta'}$$
- (S.CST): we have $\mu \rhd \Delta$ and
$$\frac{\emptyset; \Delta \vdash e\colon \tau \quad y\colon \tau; \Delta \vdash s \rhd \Omega}{\emptyset; \Delta \vdash \mathbf{cst}\ y = e;\ s \rhd \Omega}$$
By Lemma B.4, $\mu \rhd \Delta$ and $\emptyset; \Delta \vdash e\colon \tau$ and $e =_\mu w$ implies $\emptyset; \emptyset \vdash w\colon \tau$. By Lemma B.1, $y\colon \tau; \Delta \vdash s \rhd \Omega$ and $\emptyset; \emptyset \vdash w\colon \tau$ implies $\emptyset; \Delta \vdash s[y \leftarrow w] \rhd \Omega$.
- (S.FOR-I): we have $\mu \rhd \Delta, \vec{x}\colon \vec{\sigma}$ and
$$\frac{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash e\colon \mathbf{nat} \quad y\colon \mathbf{nat}; \vec{x}\colon \vec{\sigma} \vdash s \rhd \vec{x}\colon \vec{\sigma} \quad \emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash s' \rhd \Delta'}{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ s' \rhd \Delta'}$$
We have immediately $\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash s' \rhd \Delta'$.
- (S.FOR-II): we have $\mu \rhd \Delta, \vec{x}\colon \vec{\sigma}$ and
$$\frac{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash e\colon \mathbf{nat} \quad y\colon \mathbf{nat}; \vec{x}\colon \vec{\sigma} \vdash s \rhd \vec{x}\colon \vec{\sigma} \quad \emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash s' \rhd \Delta'}{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ s' \rhd \Delta'}$$
By Lemma B.1, $y\colon \mathbf{nat}; \vec{x}\colon \vec{\sigma} \vdash s \rhd \vec{x}\colon \vec{\sigma}$ and $\emptyset; \emptyset \vdash \bar{q}\colon \mathbf{nat}$ implies $\emptyset; \vec{x}\colon \vec{\sigma} \vdash s[y \leftarrow \bar{q}\,] \rhd \vec{x}\colon \vec{\sigma}$. We can then build the following typing derivation to conclude:
$$\frac{\dfrac{\overline{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash \bar{q}\colon \mathbf{nat}} \qquad y\colon \mathbf{nat}; \vec{x}\colon \vec{\sigma} \vdash s \rhd \vec{x}\colon \vec{\sigma} \qquad \emptyset; \vec{x}\colon \vec{\sigma} \vdash s[y \leftarrow \bar{q}] \rhd \vec{x}\colon \vec{\sigma}}{\emptyset; \vec{x}\colon \vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{x}};\ s[y \leftarrow \bar{q}] \rhd \vec{x}\colon \vec{\sigma}} \qquad \emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash s' \rhd \Delta'}{\emptyset; \Delta, \vec{x}\colon \vec{\sigma} \vdash \{\mathbf{for}\ y := 0\ \mathbf{until}\ \bar{q}\ \{s\}_{\vec{x}};\ s[y \leftarrow \bar{q}]\}_{\vec{x}};\ s' \rhd \Delta'}$$

□

B.4 Progress

Proposition. *For any state* (s, μ)*,* Ω *and* $\vec{z}$*, if* $\vec{z}:\vec{\tau} \vdash (s, \mu) \rhd \Omega$ *in* **IS** *then either* $s = \varepsilon$ *and no more reduction can occur, or there is a state* (s', μ') *such that* $(s, \mu) \mapsto (s', \mu')$*.*

Proof. By induction on $\vec{z}:\vec{\tau} \vdash (s, \mu) \rhd \Omega$.

- $s \equiv \varepsilon$: then we are in the first case.
- $s \equiv (\textbf{cst}\ y = e;\ s_1)$: if $e \equiv x$ then by definition of state typing, $x \in dom(\mu)$; we then have $((\textbf{cst}\ y = e;\ s_1), \mu) \mapsto (s_1[y \leftarrow \varphi_\mu(e)], \mu))$.
- $s \equiv (\textbf{var}\ y := e;\ s_1)$: if $e \equiv x$ then by definition of state typing, $x \in dom(\mu)$; by induction hypothesis on $\vec{z}:\vec{\tau}, y:\tau \vdash (s_1, (\mu, y \leftarrow \varphi_\mu(e))) \rhd \Omega, y:\tau'$, we have either $s_1 \equiv \varepsilon$ or $(s_1, (\mu, y \leftarrow \varphi_\mu(e))) \mapsto (s_1', (\mu', y \leftarrow w'))$; in the first case, we have $((\textbf{var}\ y := e;\ \varepsilon), \mu) \mapsto (\varepsilon, \mu)$, and in the second case we have $((\textbf{var}\ y := e;\ s_1), \mu) \mapsto ((\textbf{var}\ y := w';\ s_1'), \mu')$.
- $s \equiv (\{s_1\}_{\vec{z}\,'};\ s_2)$: by induction hypothesis on $\vec{z}\,':\vec{\sigma} \vdash (s_1, \mu) \rhd \vec{z}\,':\vec{\sigma}\,'$, we have either $s_1 \equiv \varepsilon$ or $(s_1, \mu) \mapsto (s_1', \mu')$; in the first case, we have $((\{\}_{\vec{z}\,'};\ s_2), \mu) \mapsto (s_2, \mu)$, and in the second case we have $((\{s_1\}_{\vec{z}\,'};\ s_2), \mu) \mapsto ((\{s_1'\}_{\vec{z}\,'};\ s_2), \mu')$.
- $s \equiv (\textbf{inc}(y);\ s_1)$: by definition of state typing, $y \in dom(\mu)$; we have $((\textbf{inc}(y);\ s_1), \mu) \mapsto ((y := \overline{q+1};\ s_1), \mu)$.
- the case for **dec** is similar to **inc**.
- $s \equiv (y := e;\ s_1)$: if $e \equiv x$ then by definition of state typing, $x \in dom(\mu)$; we have $((y := e;\ s_1), \mu) \mapsto (s_1, \mu[y \leftarrow w])$.
- $s \equiv (p(\vec{e}\,;\vec{r}\,);\ s_1)$: if $e_i \equiv x$ then by definition of state typing, $x \in dom(\mu)$, similarly for p ; we have $((p(\vec{e}\,;\vec{r}\,);\ s), \mu) \mapsto ((\{s'[\vec{y} \leftarrow \vec{w}][\vec{z} \leftrightarrow \vec{r}\,]\}_{\vec{r}};\ s), \mu[\vec{r} \leftarrow *])$.
- $s \equiv (\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s_1\}_{\vec{z}\,'};\ s_2)$: if $e \equiv x$ then by definition of state typing, $x \in dom(\mu)$; either $e =_\mu \bar{0}$ and $((\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s_1\}_{\vec{z}\,'};\ s_2), \mu) \mapsto (s_2, \mu)$, or $e \neq_\mu \bar{0}$ and $((\textbf{for}\ y := 0\ \textbf{until}\ e\ \{s_1\}_{\vec{z}\,'};\ s_2), \mu) \mapsto ((\{\textbf{for}\ y := 0\ \textbf{until}\ \bar{q}\ \ \{s_1\}_{\vec{z}\,'};\ s_1[y \leftarrow \bar{q}\,]\}_{\vec{z}\,'};\ s_2), \mu)$. □

B.5 Expressiveness

Definition B.5. *The translation of a type* $\tau \in \Sigma_{\mathbf{FS}}$ *into a type* $\tau^\natural \in \Sigma^\star_{\mathbf{FS}}$ *is defined by the following rules:*

$$\begin{aligned} \mathbf{nat}^\natural &= \mathbf{nat} \\ \mathbf{unit}^\natural &= \mathbf{unit} \\ (\sigma \to \tau)^\natural &= \sigma^\natural \to \tau^\natural \\ (\tau_1 \times \ldots \times \tau_n)^\natural &= \mathbf{unit} \to (\tau_1^\natural \times \ldots \times \tau_n^\natural) \end{aligned}$$

Proposition B.6. *For any functional term* t*, if* $\Gamma \vdash t:\tau$ *in* **FS** *then* $\Gamma^\natural \vdash t^\natural:\tau^\natural$ *in* **FS**.

Proof. Straightforward induction on t. □

Appendix C Properties of $\mathrm{ID}^{(c)}$ and $\mathrm{FD}^{(c)}$

C.1 Sequence-directed dependent type system

We present first a different (but equivalent) formulation of the imperative dependent type system which is easier to deal with when proving properties by induction on sequences:

$$\frac{x\colon\tau\in\Gamma;\Omega}{\Gamma;\Omega\vdash x\colon\tau}\ (\textsc{t.env})$$

$$\frac{}{\Gamma;\Omega\vdash \bar{q}\colon\mathbf{nat}(\mathbf{s}^q(\mathbf{0}))}\ (\textsc{t.num})$$

$$\frac{\vdash_{\mathcal{E}} n=m}{\Gamma;\Omega\vdash *\colon n=m}\ (\textsc{t.equal})$$

$$\frac{\Gamma;\Omega\vdash e'\colon\tau[n/i]\qquad\Gamma;\Omega\vdash e\colon n=m}{\Gamma;\Omega\vdash e'\colon\tau[m/i]}\ (\textsc{t.subst-i})$$

$$\frac{\Gamma;\Omega\vdash s\rhd\exists\vec{\kappa}.\Omega'[n/i]\qquad\Gamma;\Omega\vdash e\colon n=m}{\Gamma;\Omega\vdash s\rhd\exists\vec{\kappa}.\Omega'[m/i]}\ (\textsc{t.subst-ii})$$

$$\frac{}{\Gamma;\Omega,\Omega'[\vec{n}/\vec{\kappa}]\vdash\varepsilon\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.empty})$$

$$\frac{\Gamma;\Omega\vdash e\colon\tau\qquad\Gamma,y\colon\tau;\Omega\vdash s\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega\vdash\mathbf{cst}\ y=e;\ s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.cst})$$

$$\frac{\Gamma;\Omega\vdash e\colon\tau\qquad\Gamma;\Omega,y\colon\tau\vdash s\rhd\exists\vec{\kappa}.\Omega'\qquad y\notin\Omega'}{\Gamma;\Omega\vdash\mathbf{var}\ y:=e;\ s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.var})$$

$$\frac{\Gamma;\vec{x}\colon\vec{\tau}\vdash s\rhd\exists\vec{\jmath}.\vec{x}\colon\vec{\sigma}\qquad\Gamma;\Omega,\vec{x}\colon\vec{\sigma}\vdash s'\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,\vec{x}\colon\vec{\tau}\vdash\{s\}_{\vec{x}};s'\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.block})^*$$

$$\frac{\Gamma;\Omega,y\colon\mathbf{nat}(\mathbf{s}(n))\vdash s\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,y\colon\mathbf{nat}(n)\vdash\mathbf{inc}(y);s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.inc})$$

$$\frac{\Gamma;\Omega,y\colon\mathbf{nat}(\mathbf{p}(n))\vdash s\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,y\colon\mathbf{nat}(n)\vdash\mathbf{dec}(y);s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.dec})$$

$$\frac{\Gamma;\Omega,y\colon\sigma\vdash e\colon\tau\qquad\Gamma;\Omega,y\colon\tau\vdash s\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,y\colon\sigma\vdash y:=e;s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.assign})$$

$$\frac{\Gamma;\Omega,\vec{x}\colon\vec{\sigma}[\mathbf{0}/i]\vdash e\colon\mathbf{nat}(n)\quad\Gamma,y\colon\mathbf{nat}(i);\vec{x}\colon\vec{\sigma}\vdash s\rhd\vec{x}\colon\vec{\sigma}[\mathbf{s}(i)/i]\quad\Gamma;\Omega,\vec{x}\colon\vec{\sigma}[n/i]\vdash s'\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,\vec{x}\colon\vec{\sigma}[\mathbf{0}/i]\vdash\mathbf{for}\ y:=0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};s'\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.for})^*$$

$$\frac{\vec{z}\neq\emptyset\qquad\Gamma,\vec{y}\colon\vec{\sigma};\vec{z}\colon\vec{\top}\vdash s\rhd\exists\vec{\jmath}.\vec{z}\colon\vec{\tau}}{\Gamma;\Omega\vdash\mathbf{proc}\ (\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}}\colon\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma};\exists\vec{\jmath}\ \ \mathbf{out}\ \vec{\tau})}\ (\textsc{t.proc})^*$$

$$\frac{\Gamma;\Omega,\vec{r}\colon\vec{\omega}\vdash p\colon\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\sigma};\exists\vec{\jmath}\ \ \mathbf{out}\ \vec{\tau})\quad\Gamma;\Omega,\vec{r}\colon\vec{\omega}\vdash\vec{e}\colon\vec{\sigma}[\vec{u}/\vec{\imath}]\quad\Gamma;\Omega,\vec{r}\colon\vec{\tau}[\vec{u}/\vec{\imath}]\vdash s\rhd\exists\vec{\kappa}.\Omega'}{\Gamma;\Omega,\vec{r}\colon\vec{\omega}\vdash p(\vec{e};\vec{r})\,;s\rhd\exists\vec{\kappa}.\Omega'}\ (\textsc{t.call})^*$$

*where $\vec{\imath}\notin\mathcal{FV}(\Gamma)$ in (T.PROC) and $i\notin\mathcal{FV}(\Gamma)$ in (T.FOR)
and $\vec{\jmath}\notin\mathcal{FV}(\Gamma,\Omega)$ and $\vec{\jmath}\setminus\vec{\kappa}\notin\mathcal{FV}(\Omega')$ in (T.BLOCK) and (T.CALL)

Figure C.1. Imperative dependent type system

C.2 Preliminary properties for substitutions and typing

Lemma C.1. *If* $\Gamma,x\colon\tau;\Omega\vdash s\rhd\exists\vec{\imath}.\Omega'$ *and* $\emptyset;\emptyset\vdash e\colon\tau$ *in* **ID** *then* $\Gamma;\Omega\vdash s[x\leftarrow e]\rhd\exists\vec{\imath}.\Omega'$ *in* **ID**.

Proof. Straightforward induction on s. □

Lemma C.2. *If* $\Gamma; x{:}\tau, \Omega \vdash s \rhd \exists \vec{\imath}\,.(x{:}\sigma, \Omega')$ *in* **ID** *then* $\Gamma; y{:}\tau, \Omega \vdash s[x \leftarrow\!\!\rightarrow y] \rhd \exists \vec{\imath}\,.(y{:}\sigma, \Omega')$ *in* **ID**.

Proof. Straightforward induction on s. □

Lemma C.3. *If* $\Gamma; \Omega \vdash s \rhd \exists \vec{\imath}\,.\Omega'$ *in* **ID** *then for any* $x{:}\sigma$, $\Gamma, x{:}\sigma; \Omega \vdash s \rhd \exists \vec{\imath}\,.\Omega'$ *and* $\Gamma; \Omega, x{:}\sigma \vdash s \rhd \exists \vec{\imath}\,.\Omega'$ *in* **ID**.

Proof. Straightforward by induction on the typing derivation. □

Lemma C.4. *If* $\mu \rhd \Omega$ *and* $\emptyset; \Omega \vdash e{:}\tau$ *in* **ID** *and* $e =_\mu w$, *then we have* $\emptyset; \emptyset \vdash w{:}\tau$ *in* **ID**.

Proof. The case $e = w$ is trivial and if e is some variable $x \in \Omega$ then by definition of $\mu \rhd \Omega$, we have $\emptyset; \emptyset \vdash \mu(x) = w{:}\tau$. □

C.3 Typing preservation by translation from ID to FD

Theorem. *(Soundness for* **ID***). For any environments* Γ *and* Ω, *any expression* e, *any sequence* s *we have:*

- $\Gamma; \Omega \vdash e{:}\tau$ *in* **ID** *implies* $\Gamma^\star, \Omega^\star \vdash e^\star{:}\tau^\star$ *in* **FD**.
- $\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\vec{z}{:}\vec{\sigma}$ *in* **ID** *implies* $\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}{:}\exists \vec{\jmath}\,.\vec{\sigma}^\star$ *in* **FD**.

Notation C.5. *If* $\vec{z}{:}\vec{\sigma}^\star = \Omega^\star$ *we write* $(\Omega^\star)_{\vec{z}}$ *for* $\sigma_1^\star \wedge \ldots \wedge \sigma_n^\star$.

Proof. We proceed by induction on the typing derivation:

- (T.ENV)

$$\frac{y{:}\tau \in \Gamma, \Omega}{\Gamma; \Omega \vdash y{:}\tau}$$

  Indeed,

$$\frac{y{:}\tau^\star \in \Gamma^\star, \Omega^\star}{\Gamma^\star, \Omega^\star \vdash y{:}\tau^\star}$$

- (T.NUM)

$$\frac{}{\Gamma; \Omega \vdash \bar{q} : \mathbf{nat}(\mathbf{s}^q(0))}$$

  Indeed,

$$\frac{\dfrac{}{\Gamma^\star, \Omega^\star \vdash 0{:}\,\mathbf{nat}(0)}}{\Gamma^\star, \Omega^\star \vdash S^q(0){:}\,\mathbf{nat}(\mathbf{s}^q(0))}\ \ldots$$

- (T.SUBST-I)

$$\frac{\Gamma; \Omega \vdash e'{:}\tau[n/i] \quad \Gamma; \Omega \vdash e{:}\, n = m}{\Gamma; \Omega \vdash e'{:}\tau[m/i]}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash e'^\star{:}\tau[n/i] \quad \Gamma^\star, \Omega^\star \vdash e^\star{:}\, n = m}{\Gamma^\star, \Omega^\star \vdash e'^\star{:}\tau[m/i]}$$

- (T.EQUAL)

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma; \Omega \vdash *{:}\, n = m}$$

  Indeed,

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma^\star, \Omega^\star \vdash ()\,{:}\, n = m}$$

- (T.SUBST-II)

$$\frac{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'[n/i] \quad \Gamma; \Omega \vdash e{:}\, n = m}{\Gamma; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'[m/i]}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}{:}\exists \vec{\jmath}\,.(\Omega'^\star[n/i])_{\vec{z}} \quad \Gamma; \Omega \vdash e^\star{:}\, n = m}{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}{:}\exists \vec{\jmath}\,.(\Omega'^\star[m/i])_{\vec{z}}}$$

- (T.EMPTY)

$$\frac{}{\Gamma; \Omega, \Omega'[\vec{n}/\vec{\jmath}] \vdash \varepsilon \rhd \exists \vec{\jmath}\,.\Omega'}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star, \Omega'^\star[\vec{n}/\vec{\jmath}] \vdash z_1 \colon (\Omega'^\star)_{z_1}[\vec{n}/\vec{\jmath}] \quad \ldots \quad \Gamma^\star, \Omega^\star, \Omega'^\star[\vec{n}/\vec{\jmath}] \vdash z_n \colon (\Omega'^\star)_{z_n}[\vec{n}/\vec{\jmath}]}{\Gamma^\star, \Omega^\star, \Omega'^\star[\vec{n}/\vec{\jmath}] \vdash (\varepsilon)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.CST)

$$\frac{\Gamma; \Omega \vdash e \colon \tau \qquad \Gamma, y \colon \tau; \Omega \vdash s \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega \vdash \mathbf{cst}\ y = e;\ s \rhd \exists \vec{\jmath}\,.\Omega'}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star \colon \tau^\star \qquad \Gamma^\star, y \colon \tau^\star, \Omega^\star \vdash (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{z}} : \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.VAR)

$$\frac{\Gamma; \Omega \vdash e \colon \tau \qquad \Gamma; \Omega, y \colon \tau \vdash s \rhd \exists \vec{\jmath}\,.\Omega' \qquad y \notin \Omega'}{\Gamma; \Omega \vdash \mathbf{var}\ y := e;\ s \rhd \exists \vec{\jmath}\,.\Omega'}$$

  Indeed, by the substitution lemma,

$$\frac{\Gamma^\star, \Omega^\star \vdash e^\star \colon \tau^\star \qquad \Gamma^\star, y \colon \tau^\star, \Omega^\star \vdash (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}}[e^\star/y] : \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.BLOCK)

$$\frac{\Gamma; \vec{x} \colon \vec{\tau} \vdash s \rhd \exists \vec{\jmath}\,.\vec{x} \colon \vec{\sigma} \qquad \Gamma; \Omega, \vec{x} \colon \vec{\sigma} \vdash s' \rhd \exists \vec{\kappa}\,.\Omega'}{\Gamma; \Omega, \vec{x} \colon \vec{\tau} \vdash \{s\}_{\vec{x}}; s' \rhd \exists \vec{\kappa}\,.\Omega'}$$

  with $\vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega')$. Indeed,

$$\frac{\Gamma^\star, \vec{x} \colon \vec{\tau}^\star \vdash (s)^\star_{\vec{x}} \colon \exists \vec{\jmath}\,.(\vec{\sigma}^\star) \qquad \Gamma^\star, \Omega^\star, \vec{x} \colon \vec{\sigma}^\star \vdash (s')^\star_{\vec{z}} \colon \exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, \vec{x} \colon \vec{\tau}^\star \vdash \mathbf{let}\ \vec{x} = (s)^\star_{\vec{x}}\ \mathbf{in}\ (s')^\star_{\vec{z}} \colon \exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}}}$$

  since $\vec{\jmath} \notin \mathcal{FV}(\Gamma^\star, \Omega^\star)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega'^\star)$ and thus $\vec{\jmath} \notin \mathcal{FV}(\exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}})$.

- (T.INC)

$$\frac{\Gamma; \Omega, y \colon \mathbf{nat}(\mathbf{s}(n)) \vdash s \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega, y \colon \mathbf{nat}(n) \vdash \mathbf{inc}(y);\ s \rhd \exists \vec{\jmath}\,.\Omega'}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{succ} \colon \forall x(\mathbf{nat}(x) \Rightarrow \mathbf{nat}(\mathbf{s}(x))) \quad \Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash y \colon \mathbf{nat}(n)}{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{succ}(y) \colon \mathbf{nat}(\mathbf{s}(n))}$$

  and

$$\frac{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{succ}(y) \colon \mathbf{nat}(\mathbf{s}(n)) \qquad \Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(\mathbf{s}(n)) \vdash (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.DEC)

$$\frac{\Gamma; \Omega, y \colon \mathbf{nat}(\mathbf{p}(n)) \vdash s \rhd \{\vec{\jmath}\,\}\Omega'}{\Gamma; \Omega, y \colon \mathbf{nat}(n) \vdash \mathbf{dec}(y);\ s \rhd \{\vec{\jmath}\,\}\Omega'}$$

  Indeed,

$$\frac{\dfrac{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash y \colon \mathbf{nat}(n)}{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{pred}(y) \colon \mathbf{nat}(\mathbf{p}(n))} \qquad \Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(\mathbf{p}(n)) \vdash (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, y \colon \mathbf{nat}(n) \vdash \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.ASSIGN)

$$\frac{\Gamma; \Omega, y \colon \sigma \vdash e \colon \tau \qquad \Gamma; \Omega, y \colon \tau \vdash s \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega, y \colon \sigma \vdash y := e;\ s \rhd \exists \vec{\jmath}\,.\Omega'}$$

  Indeed,

$$\frac{\Gamma^\star, \Omega^\star, y \colon \sigma^\star \vdash e \colon \tau^\star \qquad \Gamma^\star, \Omega^\star, y \colon \tau^\star \vdash (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, y \colon \sigma^\star \vdash \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{z}} \colon \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.FOR)

$$\frac{\Gamma; \Omega, \vec{x} \colon \vec{\sigma}\,[\mathbf{0}/i] \vdash e \colon \mathbf{nat}(n) \quad \Gamma, y \colon \mathbf{nat}(i); \vec{x} \colon \vec{\sigma} \vdash s \rhd \vec{x} \colon \vec{\sigma}\,[\mathbf{s}(i)/i] \quad \Gamma; \Omega, \vec{x} \colon \vec{\sigma}\,[n/i] \vdash s' \rhd \exists \vec{\jmath}\,.\Omega'}{\Gamma; \Omega, \vec{x} \colon \vec{\sigma}\,[\mathbf{0}/i] \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ s' \rhd \exists \vec{\jmath}\,.\Omega'}$$

with $i \notin \mathrm{FV}(\Gamma)$. Indeed,

$$\frac{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[\mathbf{0}/i] \vdash e^\star : \mathbf{nat}(n) \quad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[\mathbf{0}/i] \vdash \vec{x} : (\vec{\sigma}^\star[\mathbf{0}/i]) \quad \Gamma^\star, y : \mathbf{nat}(i), \vec{x} : \vec{\sigma}^\star \vdash (s)^\star_{\vec{x}} : (\vec{\sigma}^\star[\mathbf{s}(i)/i])}{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{rec}(e^\star, \vec{x}\,, \lambda y.\lambda \vec{x}\,.(s)^\star_{\vec{x}}) : (\vec{\sigma}^\star[n/i])}$$

since $i \notin \mathrm{FV}(\Gamma^\star)$, and then

$$\frac{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{rec}(e^\star, \vec{x}\,, \lambda y.\lambda \vec{x}\,.(s)^\star_{\vec{x}}) : (\vec{\sigma}^\star[n/i]) \qquad \Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[n/i] \vdash (s')^\star_{\vec{z}} : \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, \vec{x} : \vec{\sigma}^\star[\mathbf{0}/i] \vdash \mathbf{let}\ \vec{x} = \mathbf{rec}(e^\star, \vec{x}\,, \lambda y.\lambda \vec{x}\,.(s)^\star_{\vec{x}})\ \mathbf{in}\ (s')^\star_{\vec{z}} : \exists \vec{\jmath}\,.(\Omega'^\star)_{\vec{z}}}$$

- (T.PROC)

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\top} \vdash s \rhd \exists \vec{\jmath}\,.\vec{z} : \vec{\tau}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{z})\,\{s\}_{\vec{z}} : \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\tau})}$$

with $\vec{\imath} \notin \mathrm{FV}(\Gamma)$. Indeed,

$$\frac{\dfrac{\dfrac{\Gamma^\star, \vec{y} : \vec{\sigma}^\star, \vec{z} : \vec{\top} \vdash (s)^\star_{\vec{z}} : \exists \vec{\jmath}\,(\vec{\tau}^\star)}{\Gamma^\star, \vec{y} : \vec{\sigma}^\star \vdash (s)^\star_{\vec{z}}[\vec{()}/\vec{z}] : \exists \vec{\jmath}\,(\vec{\tau}^\star)}}{\Gamma^\star \vdash \lambda \vec{y}\,.(s)^\star_{\vec{z}}[\vec{()}/\vec{z}] : \forall \vec{\imath}\,(\vec{\sigma}^\star \Rightarrow \exists \vec{\jmath}\,(\vec{\tau}^\star))}}{\Gamma^\star, \Omega^\star \vdash \lambda \vec{y}\,.(s)^\star_{\vec{z}}[\vec{()}/\vec{z}] : \forall \vec{\imath}\,(\vec{\sigma}^\star \Rightarrow \exists \vec{\jmath}\,(\vec{\tau}^\star))}$$

since $\vec{\imath} \notin \mathcal{FV}(\Gamma^\star)$.

- (T.CALL)

$$\frac{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p : \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}) \quad \Gamma; \Omega, \vec{r} : \vec{\omega} \vdash \vec{e} : \vec{\tau}[\vec{u}\,/\vec{\imath}\,] \quad \Gamma; \Omega, \vec{r} : \vec{\sigma}[\vec{u}\,/\vec{\imath}\,] \vdash s \rhd \exists \vec{\kappa}\,.\Omega'}{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p(\vec{e}\,; \vec{r}\,)\,; s \rhd \exists \vec{\kappa}\,.\Omega'}$$

with $\vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega')$. Indeed,

$$\frac{\Gamma^\star, \Omega^\star, \vec{r} : \vec{\omega}^\star \vdash p^\star : \forall \vec{\imath}\,(\vec{\tau}^\star \Rightarrow \exists \vec{\jmath}\,(\vec{\sigma}^\star)) \quad \Gamma^\star, \Omega^\star, \vec{r} : \vec{\omega}^\star \vdash \vec{e}^\star : (\vec{\tau}^\star)[\vec{n}\,/\vec{\imath}\,]}{\Gamma^\star, \Omega^\star, \vec{r} : \vec{\omega}^\star \vdash (p^\star\ \vec{e}^\star) : \exists \vec{\jmath}\,(\vec{\sigma}^\star[\vec{n}\,/\vec{\imath}\,])}$$

and then

$$\frac{\Gamma^\star, \Omega^\star, \vec{r} : \vec{\omega}^\star \vdash (p^\star\ \vec{e}^\star) : \exists \vec{\jmath}\,(\vec{\sigma}^\star[\vec{n}\,/\vec{\imath}\,]) \qquad \Gamma^\star, \Omega^\star, \vec{r} : \vec{\sigma}^\star[\vec{n}\,/\vec{\imath}\,] \vdash (s)^\star_{\vec{z}} : \exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}}}{\Gamma^\star, \Omega^\star, \vec{r} : \vec{\omega}^\star \vdash \mathbf{let}\ \vec{r} = p^\star\ \vec{e}^\star\ \mathbf{in}\ (s)^\star_{\vec{z}} : \exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}}}$$

since $\vec{\jmath} \notin \mathcal{FV}(\Gamma^\star, \Omega^\star)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega'^\star)$ and thus $\vec{\jmath} \notin \mathcal{FV}(\exists \vec{\kappa}\,.(\Omega'^\star)_{\vec{z}})$.

□

C.4 Typing preservation by translation from FD to ID

Notation C.6. *The following typing rules are derivable.*

$$\frac{\Gamma; \Omega, \vec{y} : \vec{\top} \vdash s \rhd \Omega'}{\Gamma; \Omega \vdash \mathbf{var}\ \vec{y}\,;\ s \rhd \Omega'}$$

$$\frac{\Gamma; \Omega \vdash \vec{w} : \vec{\tau} \qquad \Gamma; \Omega, \vec{y} : \vec{\tau} \vdash s \rhd \Omega'}{\Gamma; \Omega \vdash \mathbf{var}\ \vec{y} := \vec{w}\,;\ s \rhd \Omega'}$$

$$\frac{\Gamma, \vec{y} : \vec{\tau}; \Omega, \vec{z} : \vec{\tau} \vdash s \rhd \Omega'}{\Gamma; \Omega, \vec{z} : \vec{\tau} \vdash \mathbf{cst}\ \vec{y} = \vec{z}\,;\ s \rhd \Omega'}$$

$$\frac{\Gamma; \Omega, \vec{y} : \vec{\sigma} \vdash \vec{w} : \vec{\tau} \qquad \Gamma; \Omega, \vec{y} : \vec{\tau} \vdash s \rhd \Omega'}{\Gamma; \Omega, \vec{y} : \vec{\sigma} \vdash \vec{y} := \vec{w}\,;\ s \rhd \Omega'}$$

Lemma C.7. *The following typing rule is derivable in* **ID***:*

$$\frac{\Gamma;\Omega\vdash s\rhd\Omega'[\vec{n}/\vec{\jmath}]}{\Gamma;\Omega\vdash s\ \rhd\exists\vec{\jmath}\,.\Omega'}$$

Proof. By induction on the sequence s. □

Lemma C.8. *For all $t\in\mathcal{L}_n$, if $\Gamma\vdash t\colon\tau$ then $\tau=\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)$ for some $\vec{\imath}$, $\sigma_1,\ldots,\sigma_n$.*

Proof. By induction on $t\in\mathcal{L}_n$.

- $t\equiv v\in\mathcal{L}_n$: by definition of $v\in\mathcal{L}_n$, we have $v=(w_1,\ldots,w_n)$, hence the typing derivation of $\Gamma\vdash v\colon\tau$ ends with:

$$\frac{\Gamma\vdash w_1\colon\sigma_1[\vec{m}/\vec{\imath}]\quad\ldots\quad\Gamma\vdash w_n\colon\sigma_n[\vec{m}/\vec{\imath}]}{\Gamma\vdash(w_1,\ldots,w_n)\colon\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)}$$

- $t\equiv\mathbf{let}\ \vec{x}=u'\ \mathbf{in}\ u\in\mathcal{L}_n$ for any u': by definition, we have in all cases $u\in\mathcal{L}_n$. By induction hypothesis, we have $\Gamma\vdash u\colon\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)$ and the typing derivation of t ends with:

$$\frac{\Gamma\vdash u'\colon\exists\vec{\jmath}\,.\vec{\tau}\qquad\Gamma,\vec{x}\colon\vec{\tau}\vdash u\colon\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)}{\Gamma\vdash\mathbf{let}\ \vec{x}=u'\ \mathbf{in}\ u\colon\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)}$$

□

Theorem.

- *Given a term $t\in\mathcal{L}_n$ such that $\Gamma\vdash t\colon\exists\vec{\imath}\,.(\sigma_1\wedge\ldots\wedge\sigma_n)$ in* **FD** *with $\Gamma,\vec{\sigma}\in\Sigma^{\star}_{\mathbf{FD}}$ and a fresh mutable variable tuple $(r_1,\ldots,r_n)$ of any type $\vec{\sigma}\,'\in\Sigma_{\mathbf{ID}}$ we have $\Gamma^{\diamond};\vec{r}\colon\vec{\sigma}\,'\vdash t^{\diamond}_{\vec{r}}\rhd\exists\vec{\imath}\,.(r_1\colon\sigma_1^{\diamond},\ldots,r_n\colon\sigma_n{}^{\diamond})$ in* **ID**.
- *Given a value $v\in\mathcal{V}$ such that $\Gamma\vdash v\colon\sigma$ in* **FD** *with $\Gamma,\sigma\in\Sigma^{\star}_{\mathbf{FD}}$, for any environment Ω we have $\Gamma^{\diamond};\Omega\vdash v^{\diamond}\colon\sigma^{\diamond}$ in* **ID**.

Proof. By mutual induction on $\Gamma\vdash t\colon\vec{\sigma}$ and $\Gamma\vdash v\colon\sigma$, and by case analysis of the translation.

- $x^{\diamond}=x$

  Indeed,

$$\frac{x\colon\sigma\in\Gamma}{\Gamma\vdash x\colon\sigma}$$

$$\frac{x\colon\sigma^{\diamond}\in\Gamma^{\diamond}}{\Gamma^{\diamond};\Omega\vdash x\colon\sigma^{\diamond}}$$

- $(S^n(0))^{\diamond}=\bar{n}$

$$\frac{\dfrac{\Gamma\vdash 0\colon\mathbf{nat}(0)}{\ldots}}{\Gamma\vdash S^n(0)\colon\mathbf{nat}(\mathbf{s}^n(0))}$$

  Indeed,

$$\frac{}{\Gamma^{\diamond};\Omega\vdash\bar{n}\,:\mathbf{nat}(\mathbf{s}^n(0))}$$

- $()^{\diamond}=*$

$$\frac{}{\Gamma\vdash():(n=m)}$$

  Indeed,

$$\frac{}{\Gamma^{\diamond};\Omega\vdash*\colon(n=m)}$$

- $(\lambda\vec{x}\,.t)^{\diamond}=\mathbf{proc}\ (\mathbf{in}\ \vec{x}\,;\mathbf{out}\ \vec{z}\,)\ \{t^{\diamond}_{\vec{z}}\}_{\vec{z}}$ where $\vec{z}=(z_1,\ldots,z_m)$ and $t\in\mathcal{L}_m$.

$$\frac{\dfrac{\Gamma,\vec{x}\colon\vec{\tau}\vdash t\colon\vec{\sigma}[\vec{n}/\vec{\jmath}]}{\Gamma,\vec{x}\colon\vec{\tau}\vdash t\colon\exists\vec{\jmath}\,(\vec{\sigma})}}{\Gamma\vdash\lambda\vec{x}\,.t\colon\forall\vec{\imath}\,(\vec{\tau}\Rightarrow\exists\vec{\jmath}\,(\vec{\sigma}))}$$

With $\vec{\imath} \notin \mathrm{FV}(\Gamma)$. By lemma C.8, $t \in \mathcal{L}_m$ and $\Gamma, \vec{x}\colon\vec{\tau} \vdash t\colon \exists \vec{\jmath}\,(\vec{\sigma})$ implies $\vec{\sigma} = (\sigma_1 \wedge \ldots \wedge \sigma_m)$. By induction hypothesis, $\Gamma, \vec{x}\colon\vec{\tau} \vdash t\colon \vec{\sigma}[\vec{n}/\vec{\jmath}]$ implies $\Gamma^\diamond, \vec{x}\colon\vec{\tau}^\diamond; \vec{z}\colon\vec{\sigma}' \vdash t^\diamond_{\vec{z}} \rhd \vec{z}\colon\vec{\sigma}^\diamond[\vec{n}/\vec{\jmath}]$ for any $\vec{\sigma}'$, hence $\Gamma^\diamond, \vec{x}\colon\vec{\tau}^\diamond; \vec{z}\colon \vec{\top} \vdash t^\diamond_{\vec{z}} \rhd \vec{z}\colon\vec{\sigma}^\diamond[\vec{n}/\vec{\jmath}]$. By Lemma C.7, we $\Gamma^\diamond, \vec{x}\colon\vec{\tau}^\diamond; \vec{z}\colon \vec{\top} \vdash t^\diamond_{\vec{z}} \rhd \exists \vec{\jmath}\,.\vec{z}\colon\vec{\sigma}^\diamond$ is also derivable and then, since $\vec{\imath} \notin \mathrm{FV}(\Gamma)$,

$$\frac{\Gamma^\diamond, \vec{x}\colon\vec{\tau}^\diamond; \vec{z}\colon \vec{\top} \vdash t^\diamond_{\vec{z}} \rhd \exists \vec{\jmath}\,.\vec{z}\colon\vec{\sigma}^\diamond}{\Gamma^\diamond; \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{x}; \mathbf{out}\ \vec{z})\ \{t^\diamond_{\vec{z}}\}_{\vec{z}}\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}^\diamond)}$$

and, for any Ω, $\Gamma^\diamond; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{x}; \mathbf{out}\ \vec{z})\ \{t^\diamond_{\vec{z}}\}_{\vec{z}}\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond; \exists \vec{\jmath}\ \ \mathbf{out}\ \vec{\sigma}^\diamond)$, by weakening (Lemma C.3).

- $(\vec{w})^\diamond_{\vec{r}} = \vec{r} := \vec{w}^\diamond$;

$$\frac{\Gamma \vdash w_1\colon \sigma_1[\vec{n}/\vec{\imath}] \quad \ldots \quad \Gamma \vdash w_m\colon \sigma_m[\vec{n}/\vec{\imath}]}{\Gamma \vdash \vec{w}\colon \exists \vec{\imath}\,.\vec{\sigma}}$$

Indeed, by induction hypothesis, $\Gamma \vdash w_i\colon \sigma_i[\vec{n}/\vec{\imath}]$ implies $\Gamma^\diamond; \Omega \vdash w_i^\diamond\colon \sigma_i^\diamond[\vec{n}/\vec{\imath}]$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash w_i^\diamond\colon \sigma_i^\diamond[\vec{n}/\vec{\imath}]$. Then

$$\frac{\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash \vec{w}^\diamond\colon \vec{\sigma}^\diamond[\vec{n}/\vec{\imath}] \quad \Gamma^\diamond; \vec{r}\colon\vec{\sigma}^\diamond[\vec{n}/\vec{\imath}] \vdash \varepsilon \rhd \vec{r}\colon \exists \vec{\imath}\,.\vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash \vec{r} := \vec{w}^\diamond; \rhd \vec{r}\colon \exists \vec{\imath}\,.\vec{\sigma}^\diamond}$$

- $v^\diamond = v^\diamond$

$$\frac{\Gamma \vdash v\colon \tau[n/i] \quad \Gamma \vdash v'\colon (n = m)}{\Gamma \vdash v\colon \tau[m/i]}$$

Indeed, by induction hypothesis, $\Gamma \vdash v\colon \tau[n/i]$ implies $\Gamma^\diamond; \Omega \vdash v^\diamond\colon \tau^\diamond[n/i]$ and $\Gamma \vdash v'\colon (n = m)$ implies $\Gamma^\diamond; \Omega \vdash v'^\diamond\colon (n = m)$ for any Ω, then

$$\frac{\Gamma^\diamond; \Omega \vdash v^\diamond\colon \tau^\diamond[n/i] \quad \Gamma^\diamond; \Omega \vdash v'^\diamond\colon (n = m)}{\Gamma^\diamond; \Omega \vdash v^\diamond\colon \tau^\diamond[m/i]}$$

- $(t)^\diamond_{\vec{r}} = (t)^\diamond_{\vec{r}}$

$$\frac{\Gamma \vdash t\colon \exists \vec{\jmath}\,.\vec{\sigma}[n/i] \quad \Gamma \vdash v'\colon (n = m)}{\Gamma \vdash t\colon \exists \vec{\jmath}\,.\vec{\sigma}[m/i]}$$

Indeed, by induction hypothesis, $\Gamma \vdash t\colon \vec{\sigma}[n/i]$ implies $\Gamma^\diamond; \vec{r}\colon\vec{\tau}' \vdash (t)^\diamond_{\vec{r}} \rhd \vec{r}\colon \exists \vec{\jmath}\,.\vec{\sigma}^\diamond[n/i]$ for any $\vec{\tau}'$; $\Gamma \vdash v'\colon (n = m)$ implies $\Gamma^\diamond; \Omega \vdash v'^\diamond\colon (n = m)$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon\vec{\tau}' \vdash v'^\diamond\colon (n = m)$; then

$$\frac{\Gamma^\diamond; \vec{r}\colon\vec{\tau}' \vdash (t)^\diamond_{\vec{r}} \rhd \vec{r}\colon \exists \vec{\jmath}\,.\vec{\sigma}^\diamond[n/i] \quad \Gamma^\diamond; \vec{r}\colon\vec{\tau}' \vdash v'^\diamond\colon (n = m)}{\Gamma^\diamond; \vec{r}\colon\vec{\tau}' \vdash (t)^\diamond_{\vec{r}} \rhd \vec{r}\colon \exists \vec{\jmath}\,.\vec{\sigma}^\diamond[m/i]}$$

- $(\mathbf{let}\ y = w\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{cst}\ y = w^\diamond;\ (u)^\diamond_{\vec{r}}$

$$\frac{\Gamma \vdash w\colon \tau \quad \Gamma, y\colon\tau \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}}{\Gamma \vdash \mathbf{let}\ y = w\ \mathbf{in}\ u\colon \exists \vec{\imath}\,.\vec{\sigma}}$$

Indeed, by induction hypothesis:

  - $\Gamma \vdash w\colon \tau$ implies $\Gamma^\diamond; \Omega \vdash w^\diamond\colon \tau^\diamond$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash w^\diamond\colon \tau^\diamond$;
  - $\Gamma, y\colon\tau \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}$ implies $\Gamma^\diamond, y\colon\tau^\diamond; \vec{r}\colon\vec{\sigma}' \vdash u^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon\vec{\sigma}^\diamond$.

Then

$$\frac{\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash w^\diamond\colon \tau^\diamond \quad \Gamma^\diamond, y\colon\tau^\diamond; \vec{r}\colon\vec{\sigma}' \vdash u^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon\vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon\vec{\sigma}' \vdash \mathbf{cst}\ y = w^\diamond;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon\vec{\sigma}^\diamond}$$

- $(\mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ z := w^\diamond;\ \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}}$

$$\frac{\dfrac{\Gamma \vdash \mathbf{succ}\colon \forall x(\mathbf{nat}(x) \Rightarrow \mathbf{nat}(\mathbf{s}(x))) \quad \Gamma \vdash w\colon \mathbf{nat}(n)}{\Gamma \vdash \mathbf{succ}(w)\colon \mathbf{nat}(\mathbf{s}(n))} \qquad \Gamma, y\colon \mathbf{nat}(\mathbf{s}(n))) \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}}{\Gamma \vdash \mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u\colon \exists \vec{\imath}\,.\vec{\sigma}}$$

Indeed, by induction hypothesis:

- $\Gamma \vdash w\colon \mathbf{nat}(n)$ implies $\Gamma^\diamond; \Omega \vdash w^\diamond\colon \mathbf{nat}(n)$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash w^\diamond\colon \mathbf{nat}(n)$;
- $\Gamma, y\colon \mathbf{nat}(\mathbf{s}(n)) \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}$ implies $\Gamma, y\colon \mathbf{nat}(\mathbf{s}(n)); \vec{r}\colon \vec{\sigma}\,' \vdash u \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$, and, by Lemma C.3, $\Gamma^\diamond, y\colon \mathbf{nat}(\mathbf{s}(n)); \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{s}(n)) \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$.

Then

$$\dfrac{\dfrac{\Gamma^\diamond, y\colon \mathbf{nat}(\mathbf{s}(n)); \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{s}(n)) \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{s}(n)) \vdash \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(n) \vdash \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}$$

and

$$\dfrac{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash w^\diamond\colon \mathbf{nat}(n) \quad \Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(n) \vdash \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash \mathbf{var}\ z := w^\diamond;\ \mathbf{inc}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}$$

- $(\mathbf{let}\ y = \mathbf{pred}(w)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ z := w^\diamond;\ \mathbf{dec}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}}$

$$\dfrac{\dfrac{\Gamma \vdash w\colon \mathbf{nat}(n)}{\Gamma \vdash \mathbf{pred}(w)\colon \mathbf{nat}(\mathbf{p}(n))} \qquad \Gamma, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}}{\Gamma \vdash \mathbf{let}\ y = \mathbf{succ}(w)\ \mathbf{in}\ u\colon \exists \vec{\imath}\,.\vec{\sigma}}$$

Indeed, by induction hypothesis:

- $\Gamma \vdash w\colon \mathbf{nat}(n)$ implies $\Gamma^\diamond; \Omega \vdash w^\diamond\colon \mathbf{nat}(n)$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash w^\diamond\colon \mathbf{nat}(n)$;
- $\Gamma, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}$ implies $\Gamma, y\colon \mathbf{nat}(\mathbf{p}(n)); \vec{r}\colon \vec{\sigma}\,' \vdash u \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$, and, by Lemma C.3, $\Gamma^\diamond, y\colon \mathbf{nat}(\mathbf{p}(n)); \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{p}(n)) \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$.

Then

$$\dfrac{\dfrac{\Gamma^\diamond, y\colon \mathbf{nat}(\mathbf{p}(n)); \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{p}(n)) \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(\mathbf{p}(n)) \vdash \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(n) \vdash \mathbf{pred}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}$$

and

$$\dfrac{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash w^\diamond\colon \mathbf{nat}(n) \quad \Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', z\colon \mathbf{nat}(n) \vdash \mathbf{pred}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash \mathbf{var}\ z := w^\diamond;\ \mathbf{pred}(z);\ \mathbf{cst}\ y = z;\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond}$$

- $(\mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda \vec{y}\,.t)\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z} := \vec{w};\ \mathbf{for}\ i := 0\ \mathbf{until}\ w^\diamond\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ t^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$

$$\dfrac{\dfrac{\Gamma \vdash w\colon \mathbf{nat}(n) \quad \Gamma \vdash \vec{w}\colon \vec{\tau}[0/j] \quad \Gamma, i\colon \mathbf{nat}(j), \vec{y}\colon \vec{\tau} \vdash t\colon \vec{\tau}[\mathbf{s}(j)/j]}{\Gamma \vdash \mathbf{rec}(w, \vec{w}, \lambda i.\lambda \vec{y}\,.t)\colon \vec{\tau}[n/j]} \qquad \Gamma, \vec{x}\colon \vec{\tau}[n/j] \vdash u\colon \exists \vec{\imath}\,.\vec{\sigma}}{\Gamma \vdash \mathbf{let}\ \vec{x} = \mathbf{rec}(w, \vec{w}, \lambda i.\lambda \vec{y}\,.t)\ \mathbf{in}\ u\colon \exists \vec{\imath}\,.\vec{\sigma}}$$

with $j \notin \mathrm{FV}(\Gamma)$. Indeed, by induction hypothesis:

- $\Gamma \vdash w\colon \mathbf{nat}(n)$ implies $\Gamma^\diamond; \Omega \vdash w^\diamond\colon \mathbf{nat}(n)$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,', \vec{z}\colon \vec{\tau}^\diamond[0/j] \vdash w^\diamond\colon \mathbf{nat}(n)$;
- $\Gamma \vdash \vec{w}\colon \vec{\tau}[0/j]$ implies $\Gamma^\diamond; \Omega \vdash \vec{w}^\diamond\colon \vec{\tau}^\diamond[0/j]$ for any Ω, hence $\Gamma^\diamond; \vec{r}\colon \vec{\sigma}\,' \vdash \vec{w}^\diamond\colon \vec{\tau}^\diamond[0/j]$;
- $\Gamma, \vec{x}\colon \vec{\tau}[n/j] \vdash u\colon \vec{\sigma}$ implies $\Gamma^\diamond, \vec{x}\colon \vec{\tau}^\diamond[n/j]; \vec{r}\colon \vec{\sigma}\,' \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$, and, by Lemma C.3, $\Gamma^\diamond, \vec{x}\colon \vec{\tau}^\diamond[n/j]; \vec{r}\colon \vec{\sigma}\,', \vec{z}\colon \vec{\tau}^\diamond[n/j] \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath}\,.\vec{r}\colon \vec{\sigma}^\diamond$;
- $\Gamma, i\colon \mathbf{nat}(j), \vec{y}\colon \vec{\tau} \vdash t\colon \vec{\tau}[\mathbf{s}(j)/j]$ implies $\Gamma^\diamond, i\colon \mathbf{nat}(j), \vec{y}\colon \vec{\tau}^\diamond; \vec{z}\colon \vec{\tau}\,' \vdash t^\diamond_{\vec{z}} \rhd \vec{z}\colon \vec{\tau}^\diamond[\mathbf{s}(j)/j]$ for any $\vec{\tau}\,'$, hence $\Gamma^\diamond, i\colon \mathbf{nat}(j), \vec{y}\colon \vec{\tau}^\diamond; \vec{z}\colon \vec{\tau}^\diamond \vdash t^\diamond_{\vec{z}} \rhd \vec{z}\colon \vec{\tau}^\diamond[\mathbf{s}(j)/j]$.

Then

$$\pi_1 = \dfrac{\Gamma^\diamond, i\colon \mathbf{nat}(j), \vec{y}\colon \vec{\tau}^\diamond; \vec{z}\colon \vec{\tau}^\diamond \vdash t^\diamond_{\vec{z}} \rhd \vec{z}\colon \vec{\tau}^\diamond[\mathbf{s}(j)/j]}{\Gamma^\diamond, i\colon \mathbf{nat}(j); \vec{z}\colon \vec{\tau}^\diamond \vdash \mathbf{cst}\ \vec{y} = \vec{z};\ t^\diamond_{\vec{z}} \rhd \vec{z}\colon \vec{\tau}^\diamond[\mathbf{s}(j)/j]}$$

and

$$\pi_2 = \frac{\Gamma^\diamond, \vec{x} : \vec{\tau}^\diamond[n/j]; \vec{r} : \vec{\sigma}', \vec{z} : \vec{\tau}^\diamond[n/j] \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r} : \vec{\sigma}', \vec{z} : \vec{\tau}^\diamond[n/j] \vdash \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}$$

and

$$\pi = \frac{\Gamma^\diamond; \vec{r} : \vec{\sigma}', \vec{z} : \vec{\tau}^\diamond[0/j] \vdash w^\diamond : \mathbf{nat}(n) \qquad \pi_1 \qquad \pi_2}{\Gamma^\diamond; \vec{r} : \vec{\sigma}', \vec{z} : \vec{\tau}^\diamond[0/j] \vdash \mathbf{for}\ i := 0\ \mathbf{until}\ w^\diamond\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ t^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}$$

since $j \notin \mathrm{FV}(\Gamma^\diamond)$, and finally

$$\frac{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,' \vdash \vec{w}^\diamond : \vec{\tau}^\diamond[0/j] \qquad \pi}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,' \vdash \mathbf{var}\ \vec{z} := \vec{w};\ \mathbf{for}\ i := 0\ \mathbf{until}\ w^\diamond\ \{\mathbf{cst}\ \vec{y} = \vec{z};\ t^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}$$

- $(\mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z};\ w^\diamond(\vec{w}^\diamond; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$

$$\frac{\dfrac{\Gamma \vdash w : \forall \vec{\imath}\,(\vec{\tau} \Rightarrow \exists \vec{\jmath}\,(\vec{\tau}\,')) \quad \Gamma \vdash \vec{w} : \vec{\tau}[\vec{n}/\vec{\imath}]}{\Gamma \vdash w\ \vec{w} : \exists \vec{\jmath}\,(\vec{\tau}\,'[\vec{n}/\vec{\imath}])} \qquad \Gamma, \vec{x} : \vec{\tau}\,'[\vec{n}/\vec{\imath}] \vdash u : \exists \vec{\kappa} . \vec{\sigma}}{\Gamma \vdash \mathbf{let}\ \vec{x} = w\ \vec{w}\ \mathbf{in}\ u : \exists \vec{\kappa} . \vec{\sigma}}$$

with $\vec{\jmath} \notin \mathrm{FV}(\Gamma, \vec{\sigma})$. Indeed, by induction hypothesis:

  - $\Gamma \vdash w : \forall \vec{\imath}\,(\vec{\tau} \Rightarrow \exists \vec{\jmath}\,(\vec{\tau}\,'))$ implies $\Gamma^\diamond; \Omega \vdash w^\diamond : \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,'^\diamond)$ for any Ω, hence $\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash w^\diamond : \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,'^\diamond)$;
  - $\Gamma \vdash \vec{w} : \vec{\tau}[\vec{n}/\vec{\imath}]$ implies $\Gamma^\diamond; \Omega \vdash \vec{w}^\diamond : \vec{\tau}^\diamond[\vec{n}/\vec{\imath}]$ for any Ω, hence $\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash \vec{w}^\diamond : \vec{\tau}^\diamond[\vec{n}/\vec{\imath}]$;
  - $\Gamma, \vec{x} : \vec{\tau}\,'[\vec{n}/\vec{\imath}] \vdash u : \exists \vec{\kappa} . \vec{\sigma}$ implies $\Gamma^\diamond, \vec{x} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}]; \vec{r} : \vec{\sigma}\,' \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond$, and, by Lemma C.3, $\Gamma^\diamond, \vec{x} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}]; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}] \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond$.

  Then

$$\pi = \frac{\Gamma^\diamond, \vec{x} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}]; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}] \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}\,'^\diamond[\vec{n}/\vec{\imath}] \vdash \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond}$$

  and

$$\frac{\dfrac{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash w^\diamond : \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}^\diamond; \exists \vec{\jmath}\ \mathbf{out}\ \vec{\tau}\,'^\diamond) \quad \Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash \vec{w}^\diamond : \vec{\tau}^\diamond[\vec{n}/\vec{\imath}] \quad \pi}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash w^\diamond(\vec{w}^\diamond; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond}}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,' \vdash \mathbf{var}\ \vec{z};\ w^\diamond(\vec{w}^\diamond; \vec{z});\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\kappa} . \vec{r} : \vec{\sigma}^\diamond}$$

  since $\vec{\jmath} \notin \mathrm{FV}(\Gamma^\diamond, \vec{\sigma}\,', \vec{\sigma})$.

- $(\mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u)^\diamond_{\vec{r}} = \mathbf{var}\ \vec{z};\ \{(t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}}$

$$\frac{\Gamma \vdash t : \exists \vec{\kappa} . \vec{\tau} \quad \Gamma, \vec{x} : \vec{\tau} \vdash u : \exists \vec{\imath} . \vec{\sigma}}{\Gamma \vdash \mathbf{let}\ \vec{x} = t\ \mathbf{in}\ u : \exists \vec{\imath} . \vec{\sigma}}$$

  Indeed, by induction hypothesis:

  - $\Gamma \vdash t : \exists \vec{\kappa} . \vec{\tau}$ implies $\Gamma^\diamond;\ \vec{z} : \vec{\tau}\,' \vdash (t)^\diamond_{\vec{z}} \rhd \exists \vec{\kappa} . \vec{z} : \vec{\tau}^\diamond$ for any $\vec{\tau}\,'$, hence, by Lemma C.3 $\Gamma^\diamond;\ \vec{r} : \vec{\sigma}\,',\ \vec{z} : \vec{\top} \vdash (t)^\diamond_{\vec{z}} \rhd \exists \vec{\kappa} . \vec{z} : \vec{\tau}^\diamond$;
  - $\Gamma, \vec{x} : \vec{\tau} \vdash u : \exists \vec{\imath} . \vec{\sigma}$ implies $\Gamma^\diamond, \vec{x} : \vec{\tau}^\diamond; \vec{r} : \vec{\sigma}\,' \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond$, and, by Lemma C.3, $\Gamma^\diamond, \vec{x} : \vec{\tau}^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}^\diamond \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond$.

  Then

$$\frac{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash (t)^\diamond_{\vec{z}} \rhd \exists \vec{\kappa} . \vec{z} : \vec{\tau}^\diamond \qquad \dfrac{\Gamma^\diamond, \vec{x} : \vec{\tau}^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}^\diamond \vdash (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\tau}^\diamond \vdash \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}}{\dfrac{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,', \vec{z} : \vec{\top} \vdash \{(t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}{\Gamma^\diamond; \vec{r} : \vec{\sigma}\,' \vdash \mathbf{var}\ \vec{z};\ \{(t)^\diamond_{\vec{z}}\}_{\vec{z}};\ \mathbf{cst}\ \vec{x} = \vec{z};\ (u)^\diamond_{\vec{r}} \rhd \exists \vec{\imath} . \vec{r} : \vec{\sigma}^\diamond}}$$

□

C.5 Expressiveness

Definition C.9. *The translation of a type $\tau \in \Sigma_{\mathbf{FD}^c}$ into a type $\tau^\natural \in \Sigma_{\mathbf{FD}^c}$ is defined by the following rules:*

$$\begin{array}{rcl} \mathbf{nat}(n)^\natural & = & \mathbf{nat}(n) \\ (n = m)^\natural & = & (n = m) \\ (\forall \vec{\imath}\,(\sigma \Rightarrow \tau))^\natural & = & \forall \vec{\imath}\,(\sigma^\natural \Rightarrow \tau^\natural) \\ (\exists \vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_n))^\natural & = & \top \Rightarrow \exists \vec{\imath}\,(\tau_1^\natural \wedge \ldots \wedge \tau_n^\natural) \end{array}$$

$$\bot^\natural = \bot$$

Proposition C.10. *For any functional term t, if $\Gamma \vdash t : \tau$ in $\mathbf{FD}^c$ then $\Gamma^\natural \vdash t^\natural : \tau^\natural$ in $\mathbf{FD}^c$.*

Proof. Straightforward induction on t. □

C.6 CPS translation

Lemma C.11. *The following typing rules are derivable in* **FD***:*

$$\frac{\Gamma \vdash u : \varphi}{\Gamma \vdash \mathbf{val}\ u : \nabla\varphi} \qquad \frac{\Gamma \vdash u : \nabla\varphi \qquad \Gamma, x : \varphi \vdash t : \nabla\psi}{\Gamma \vdash \mathbf{let\ val}\ x = u\ \mathbf{in}\ t : \nabla\psi}$$

Proof. Indeed,

$$\dfrac{\dfrac{\dfrac{}{\Gamma, z : \varphi \Rightarrow o \vdash z : \varphi \Rightarrow o} \qquad \Gamma \vdash u : \varphi}{\Gamma, z : \varphi \Rightarrow o \vdash z\ u : o}}{\Gamma \vdash \lambda z.(z\ u) : (\varphi \Rightarrow o) \Rightarrow o}$$

and,

$$\dfrac{\dfrac{\Gamma \vdash u : (\varphi \Rightarrow o) \Rightarrow o}{\Gamma, z : \psi \Rightarrow o \vdash u : (\varphi \Rightarrow o) \Rightarrow o} \qquad \dfrac{\dfrac{\dfrac{\Gamma, x : \varphi \vdash t : (\psi \Rightarrow o) \Rightarrow o}{\Gamma, z : \psi \Rightarrow o, x : \varphi \vdash t : (\psi \Rightarrow o) \Rightarrow o} \qquad \dfrac{}{\Gamma, z : \psi \Rightarrow o, x : \varphi \vdash z : \psi \Rightarrow o}}{\Gamma, z : \psi \Rightarrow o, x : \varphi \vdash (t\ z) : o}}{\Gamma, z : \psi \Rightarrow o \vdash \lambda x.(t\ z) : \varphi \Rightarrow o}}{\dfrac{\Gamma, z : \psi \Rightarrow o \vdash (u\ \lambda x.(t\ z)) : o}{\Gamma \vdash \lambda z.(u\ \lambda x.(t\ z)) : (\psi \Rightarrow o) \Rightarrow o}}$$

□

Lemma C.12. *Abbreviations callcc and throw are typable in* **FD** *as follows:*

$$\begin{array}{rcl} \mathit{callcc} & : & ((\varphi^\circ \Rightarrow o) \Rightarrow \nabla\varphi^\circ) \Rightarrow \nabla\varphi^\circ \\ \mathit{throw} & : & ((\varphi^\circ \Rightarrow o) \wedge \varphi^\circ) \Rightarrow \nabla\psi^\circ \end{array}$$

Proof. Indeed (with $\Gamma' = \Gamma, h : (\varphi^\circ \Rightarrow o) \Rightarrow ((\varphi^\circ \Rightarrow o) \Rightarrow o), k : \varphi^\circ \Rightarrow o$),

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash h : (\varphi^\circ \Rightarrow o) \Rightarrow ((\varphi^\circ \Rightarrow o) \Rightarrow o)} \qquad \dfrac{}{\Gamma' \vdash k : \varphi^\circ \Rightarrow o}}{\Gamma' \vdash (h\ k) : (\varphi^\circ \Rightarrow o) \Rightarrow o} \qquad \dfrac{}{\Gamma' \vdash k : \varphi^\circ \Rightarrow o}}{\Gamma' \vdash (h\ k\ k) : o}}{\Gamma, h : (\varphi^\circ \Rightarrow o) \Rightarrow ((\varphi^\circ \Rightarrow o) \Rightarrow o) \vdash \lambda k.(h\ k\ k) : (\varphi^\circ \Rightarrow o) \Rightarrow o}}{\Gamma \vdash \lambda h.\lambda k.(h\ k\ k) : ((\varphi^\circ \Rightarrow o) \Rightarrow ((\varphi^\circ \Rightarrow o) \Rightarrow o)) \Rightarrow (\varphi^\circ \Rightarrow o) \Rightarrow o}$$

and,

$$\dfrac{\dfrac{\dfrac{}{\Gamma, k\colon \varphi^\circ \Rightarrow o, a\colon \varphi^\circ, k'\colon \psi^\circ \Rightarrow o \vdash k\colon \varphi^\circ \Rightarrow o} \qquad \dfrac{}{\Gamma, k\colon \varphi^\circ \Rightarrow o, a\colon \varphi^\circ, k'\colon \psi^\circ \Rightarrow o \vdash a\colon \varphi^\circ}}{\Gamma, k\colon \varphi^\circ \Rightarrow o, a\colon \varphi^\circ, k'\colon \psi^\circ \Rightarrow o \vdash k\ a\colon o}}{\dfrac{\Gamma, k\colon \varphi^\circ \Rightarrow o, a\colon \varphi^\circ \vdash \lambda k'.(k\ a)\colon (\psi^\circ \Rightarrow o) \Rightarrow o}{\Gamma \vdash \lambda(k, a).\lambda k'.(k\ a)\colon ((\varphi^\circ \Rightarrow o) \wedge \varphi^\circ) \Rightarrow (\psi^\circ \Rightarrow o) \Rightarrow o}}$$

□

C.7 Labels and jumps

Proposition. *The following typing rules are derivable.*

$$\dfrac{\Gamma, k\colon \neg\vec{\sigma}; \vec{z}\colon \vec{\tau} \vdash s \rhd \vec{z}\colon \vec{\sigma} \qquad \Gamma; \Omega, \vec{z}\colon \vec{\sigma} \vdash s' \rhd \Omega', \vec{z}\colon \vec{\sigma}\,'}{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash k\colon \{s\}_{\vec{z}};\ s' \rhd \Omega', \vec{z}\colon \vec{\sigma}\,'}$$

$$\dfrac{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash k\colon \neg\vec{\sigma} \qquad \Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash \vec{e}\colon \vec{\sigma}}{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash \mathbf{jump}(k, \vec{e}\,)_{\vec{z}} \rhd \Omega', \vec{z}\colon \vec{\omega}\,'}$$

Proof. Indeed, given the type of **callcc** and **throw**, we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, k\colon \neg\vec{\sigma}; \vec{z}\colon \vec{\tau} \vdash s \rhd \vec{z}\colon \vec{\sigma}}{\Gamma, \vec{z}\,'\colon \vec{\tau}, k\colon \neg\vec{\sigma}; \vec{z}\colon \vec{\tau} \vdash s \rhd \vec{z}\colon \vec{\sigma}}}{\Gamma, \vec{z}\,'\colon \vec{\tau}, k\colon \neg\vec{\sigma}; \vec{z}\colon \vec{\top} \vdash \vec{z} := \vec{z}\,';\ s \rhd \vec{z}\colon \vec{\sigma}}}{\Gamma, \vec{z}\,'\colon \vec{\tau}; \Omega, \vec{z}\colon \vec{\tau} \vdash \mathbf{proc}\ (\mathbf{in}\ k; \mathbf{out}\ \vec{z}\,)\ \{\vec{z} := \vec{z}\,';\ s\}_{\vec{z}}\colon \mathbf{proc}\ (\mathbf{in}\ \neg\vec{\sigma}; \mathbf{out}\ \vec{\sigma}\,)} \qquad \Gamma; \Omega, \vec{z}\colon \vec{\sigma} \vdash s' \rhd \Omega', \vec{z}\colon \vec{\sigma}\,'}{\Gamma, \vec{z}\,'\colon \vec{\tau}; \Omega, \vec{z}\colon \vec{\tau} \vdash \mathbf{callcc}(\mathbf{proc}\ (\mathbf{in}\ k; \mathbf{out}\ \vec{z}\,)\ \{\vec{z} := \vec{z}\,';\ s\}_{\vec{z}}; \vec{z}\,);\ s' \rhd \Omega', \vec{z}\colon \vec{\sigma}\,'}}{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash \mathbf{cst}\ \vec{z}\,' = \vec{z};\ \mathbf{callcc}(\mathbf{proc}\ (\mathbf{in}\ k; \mathbf{out}\ \vec{z}\,)\ \{\vec{z} := \vec{z}\,';\ s\}_{\vec{z}}; \vec{z}\,);\ s' \rhd \Omega', \vec{z}\colon \vec{\sigma}\,'}$$

$$\dfrac{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash k\colon \neg\vec{\sigma} \qquad \Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash \vec{e}\colon \vec{\sigma}}{\Gamma; \Omega, \vec{z}\colon \vec{\tau} \vdash \mathbf{throw}(k, \vec{e}\,; \vec{z}\,) \rhd \vec{z}\colon \vec{\omega}\,'}$$

□

Appendix D Examples of imperative programs

In this appendix, we adopt the two following conventions. As is it usual to do in natural deduction derivations, we will discard the typing environment of functional programs and "bracket-close" hypothesis with variables annotations as they are bound by λ-abstractions. Moreover, we will use the substitution rule (in both functional and imperative typing derivations) without explicitly displaying the equations we use, we instead decorate the judgment with their numbers in the definitions given along with the examples.

To begin with, we consider the usual axioms for Peano's arithmetic:

$$\begin{array}{lrcl} (1) & x+0 & = & x \\ (2) & x+\mathbf{s}(i) & = & \mathbf{s}(x+i) \\ (3) & x\times 0 & = & 0 \\ (4) & x\times\mathbf{s}(i) & = & (x\times i)+x \end{array}$$

D.1 Multiplication

D.1.1 Multiplication in FD

Let $\mathcal{D}_s$ be the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{add\colon \forall p(\mathbf{nat}(p)\Rightarrow\forall q(\mathbf{nat}(q)\Rightarrow\mathbf{nat}(p+q)))\quad z\colon\mathbf{nat}(n\times u)}{(add\ z)\colon\forall q(\mathbf{nat}(q)\Rightarrow\mathbf{nat}(n\times u+q))} \qquad x\colon\mathbf{nat}(n)}{(add\ z\ x)\colon\mathbf{nat}((n\times u)+n)}}{(add\ z\ x)\colon\mathbf{nat}(n\times\mathbf{s}(u))}(4)$$

Then

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{0\colon\mathbf{nat}(0)}{0\colon\mathbf{nat}(n\times 0)}(3)}{y\colon\mathbf{nat}(m)\qquad \mathcal{D}_s}}{\mathbf{rec}(y,0,\lambda i.\lambda z.(add\ z\ x))\colon\mathbf{nat}(n\times m)}}{\lambda y.\mathbf{rec}(y,0,\lambda i.\lambda z.(add\ z\ x))\colon\forall m(\mathbf{nat}(m)\Rightarrow\mathbf{nat}(n\times m))}}{\lambda x.\lambda y.\mathbf{rec}(y,0,\lambda i.\lambda z.(add\ z\ x))\colon\forall n(\mathbf{nat}(n)\Rightarrow\forall m(\mathbf{nat}(m)\Rightarrow\mathbf{nat}(n\times m)))}$$

D.1.2 Multiplication in ID

```
cst mult = proc (in X, Y; out Z) {
    Z := 0;

    for I := 0 until Y {
        add(Z, X; Z);
    }_Z;

}_Z;
```

```
−(X: nat(x), Y: nat(y))[Z: ⊤]
|   [Z: nat(x × 0)]      by (3)
|
|   −(I: nat(i))[Z: nat(x × i)]
|   |   [Z: nat(x × s(i))]      by (4)
|   [Z: nat(x × y)]
|
(mult: proc ∀x, y(in nat(x), nat(y); out nat(x × y)))
```

D.2 Factorial

Here follows the equations defining a version of the factorial function:

$$\begin{array}{lrcl} (1) & 0\mathbf{!} & = & \mathbf{s}(0) \\ (2) & \mathbf{s}(n)\mathbf{!} & = & n\mathbf{!}\times\mathbf{s}(n) \end{array}$$

D.2.1 Factorial in FD

Let $\mathcal{D}_s$ be the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathit{mult}\colon \forall n(\mathbf{nat}(n) \Rightarrow \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n \times m)))}{(\mathit{mult}\ z)\colon \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(u! \times m))} \quad z\colon \mathbf{nat}(u!) \qquad \dfrac{i\colon \mathbf{nat}(u)}{\mathbf{s}(i)\colon \mathbf{nat}(\mathbf{s}(u))}}{(\mathit{mult}\ z\ \mathbf{s}(i))\colon \mathbf{nat}(u! \times \mathbf{s}(u))}}{(\mathit{mult}\ z\ \mathbf{s}(i))\colon \mathbf{nat}(\mathbf{s}(u)!)}(2)$$

Then

$$\dfrac{\dfrac{x\colon \mathbf{nat}(p) \qquad \dfrac{1\colon \mathbf{nat}(\mathbf{s}(0))}{1\colon \mathbf{nat}(0!)}(1) \qquad \mathcal{D}_s}{\mathbf{rec}(x, 1, \lambda i.\lambda z.(\mathit{mult}\ z\ \mathbf{s}(i)))\colon \mathbf{nat}(p!)}}{\lambda x.\mathbf{rec}(x, 1, \lambda i.\lambda z.(\mathit{mult}\ z\ \mathbf{s}(i)))\colon \forall n(\mathbf{nat}(p) \Rightarrow \mathbf{nat}(p!))}$$

D.2.2 Factorial in ID

```
cst fact = proc (in X; out Z) {
    Z := 1;

    for I := 0 until X {
        var Y := I;
        inc(Y);
        mult(Z, Y; Z);
    }Z;

}Z;
```

```
−(X: nat(n))[Z: ⊤]
|   [Z: nat(0!)]      by (1)
|
|   −(I: nat(i))[Z: nat(i!)]
|   |   [Y: nat(i)]
|   |   [Y: nat(s(i))]
|   |   [Z: nat(s(i)!)]      by (4)
|   [Z: nat(n!)]
|
(fact: proc ∀n(in nat(n); out nat(n!)))
```

D.3 Ackermann function

Here follows the equations defining a version of the Ackermann function (from [Leivant, 2002]):

$$\begin{array}{lrcl} (1) & \mathbf{a}(0, n) & = & \mathbf{s}(n) \\ (2) & \mathbf{a}(\mathbf{s}(z), 0) & = & \mathbf{s}(\mathbf{s}(0)) \\ (3) & \mathbf{a}(z, \mathbf{a}(\mathbf{s}(z), u)) & = & \mathbf{a}(\mathbf{s}(z), \mathbf{s}(u)) \end{array}$$

D.3.1 Ackermann in FD

Here follows an annotated version of the proof given in [Leivant, 2002].

Let $\mathcal{D}_s$ be the derivation

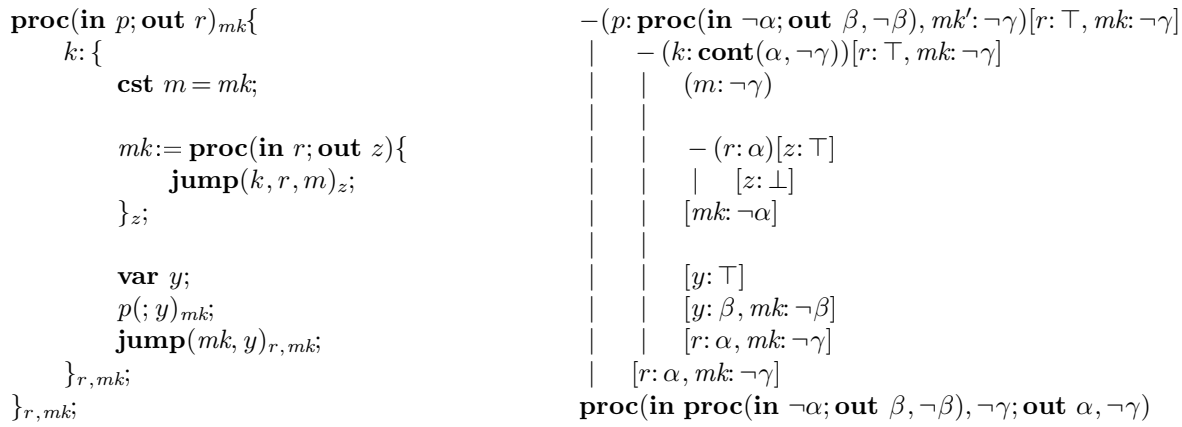
$$\dfrac{\dfrac{y\colon \mathbf{nat}(n) \qquad \dfrac{\dfrac{\dfrac{0\colon \mathbf{nat}(0)}{S(0)\colon \mathbf{nat}(\mathbf{s}(0))}}{S(S(0))\colon \mathbf{nat}(\mathbf{s}(\mathbf{s}(0)))}}{S(S(0))\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(z), 0))}(2) \qquad \dfrac{f\colon \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(z, n))) \quad k\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(z), u))}{(f\ k)\colon \mathbf{nat}(\mathbf{a}(z, \mathbf{a}(\mathbf{s}(z), u)))}}{(f\ k)\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(z), \mathbf{s}(u)))}(3)}{\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k))\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(z), n))}}{\lambda y.\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k))\colon \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(\mathbf{s}(z), n)))}$$

Then

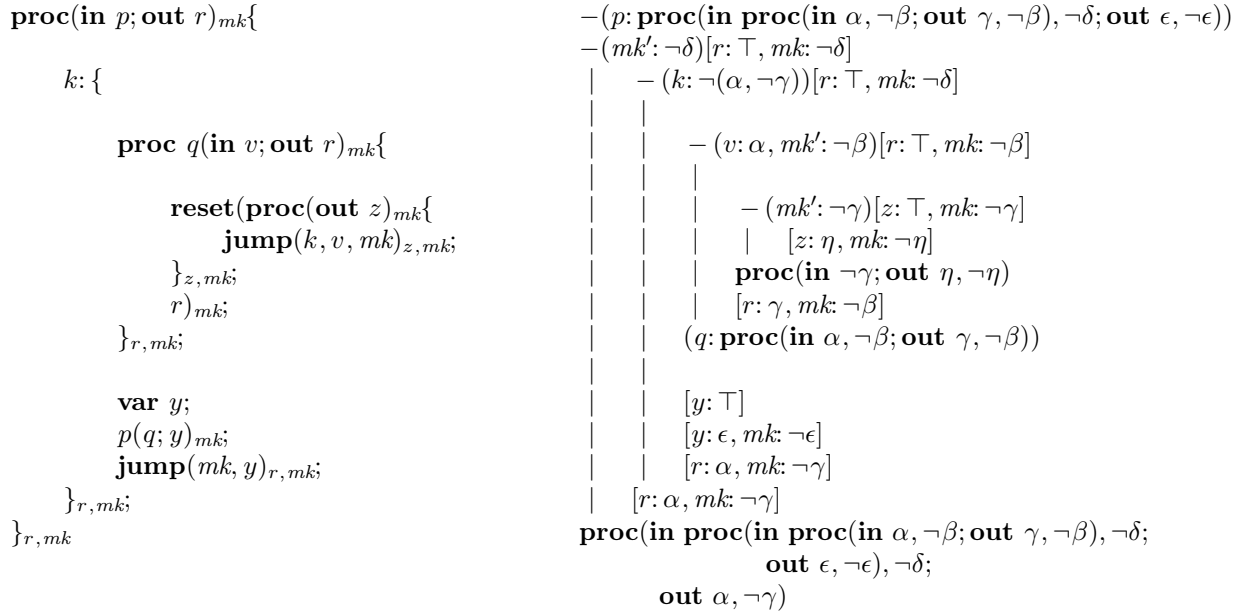
$$\dfrac{\dfrac{x\colon \mathbf{nat}(m) \qquad \dfrac{\dfrac{\dfrac{y\colon \mathbf{nat}(\mathbf{s}(n))}{y\colon \mathbf{nat}(\mathbf{a}(0, n))}(1)}{S(y)\colon \mathbf{nat}(\mathbf{a}(0, n))}}{\lambda y.S(y)\colon \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(0, n)))} \qquad \mathcal{D}_s}{\mathbf{rec}(x, \lambda y.S(y), \lambda i.\lambda f.\lambda y.\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k)))\colon \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(m, n)))}}{\lambda x.\mathbf{rec}(x, \lambda y.S(y), \lambda i.\lambda f.\lambda y.\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k)))\colon \forall m(\mathbf{nat}(m) \Rightarrow \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(m, n))))}$$

D.4 Shift and Reset typing derivations

D.4.1 Reset typing derivations



D.4.2 Shift typing derivation



Appendix E Shift and reset in state-passing style

```
signature CONT = sig

  type void
  type 'a K = 'a → void
```

```
  val callcc: ('a K → 'a) → 'a
  val throw: 'a K → 'a → 'b

end

functor ShiftReset(Cont: CONT) = struct

  open Cont

  val reset: ('a K → 'c * 'c K) * 'd K → 'a * 'd K =
   fn (p,mk') ⇒
      let
        val (r,mk) = ((),mk')
        val (r',mk') = (r,mk)
        val (r,mk) =
            callcc (fn k ⇒
                      let
                        val (r,mk) = (r',mk')
                        val m = mk
                        val mk = fn r ⇒
                                   let val z = throw k (r,m)
                                   in z end
                        val (y,mk) = p(mk)
                        val (r,mk) = throw mk y
                      in (r,mk) end)
      in (r,mk) end

  val shift: (('a * 'b K → 'c * 'b K) * 'd K → 'e * 'e K) * 'd K → 'a * 'c K =
   fn (p,mk') ⇒
      let
        val (r,mk) = ((),mk')
        val (r',mk') = (r,mk)
        val (r,mk) =
            callcc (fn k ⇒
                      let
                        val (r,mk) = (r',mk')
                        val q = fn (v,mk) ⇒
                                  let val (r,mk) =
                                        reset (fn mk ⇒
                                                 let val (z,mk) = throw k (v,mk)
                                                 in (z,mk) end,mk)
                                  in (r,mk) end
                        val (y,mk) = p(q,mk)
                        val (r,mk) = throw mk y
                      in (r,mk) end)
      in (r,mk) end
end
```

Bibliography

[Appel and MacQueen, 1991] Appel, A. W. and MacQueen, D. B. (1991). Standard ML of new jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag.

[Apt, 1981] Apt, K. R. (1981). Ten years of hoare's logic: A survey – part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483.

[Ariola et al., 2007] Ariola, Z. M., Herbelin, H., and Sabry, A. (2007). A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*.

[Atkey, 2006] Atkey, R. (2006). Parameterised notions of computation. In McBride, C. and Uustalu, T., editors, *MSFP 2006: Workshop on mathematically structured functional programming*. Electronic Workshops in Computing, British Computer Society.

[Barbanera and Berardi, 1994] Barbanera, F. and Berardi, S. (1994). Extracting constructive content from classical logic via control-like reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag.

[Benton et al., 1998] Benton, P. N., Bierman, G. M., and de Paiva, V. (1998). Computational types from a logical perspective. *J. Funct. Program*, 8(2):177–193.

[Clarke, 1979] Clarke, E. M. (1979). Programming language constructs for which it is impossible to obtain good hoare axioms. *Journal of the ACM*, 26(1).

[Clint and Hoare, 1972] Clint, M. and Hoare, C. A. R. (1972). Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224.

[Colson and Fredholm, 1998] Colson, L. and Fredholm, D. (1998). System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315.

[Coquand, 1996] Coquand, T. (1996). Computational content of classical logic.

[Cousot, 1990] Cousot, P. (1990). Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–994. Elsevier Science Publishers B.V. (North Holland).

[Crolard et al., 2009] Crolard, T., Polonowski, E., and Valarcher, P. (2009). Extending the loop language with higher-order procedural variables. *Special issue of ACM TOCL on Implicit Computational Complexity*, 10(4):1–37.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic*. North-Holland.

[Damm and Josko, 1983] Damm, W. and Josko, B. (1983). A sound and relatively complete hoare-logic for a language with higher type procedures. *Acta Informatica*, 20(1):59–101.

[Danvy and Filinski, 1989] Danvy, O. and Filinski, A. (1989). A functional abstraction of typed contexts. Technical report, Copenhagen University.

[de Groote, 1995] de Groote, P. (1995). A simple calculus of exception handling. In *Second International Conference on Typed Lambda Calculi and Applications*, LNCS, pages 201–215, Edinburgh, United Kingdom.

[Donahue, 1977] Donahue, J. E. (1977). Locations considered unnecessary. *Acta Inf.*, 8:221–242.

[Felleisen, 1987] Felleisen, M. (1987). *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages*. PhD thesis, Indianapolis, IN, USA.

[Feng et al., 2006] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., and Ni, Z. (2006). Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 401–414, New York, NY, USA. ACM Press.

[Filinski, 1994] Filinski, A. (1994). Representing monads. In *Conference Record of the Twenty-First Annual Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon.

[Filinski, 1999] Filinski, A. (1999). Representing layered monads. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 175–188.

[Flanagan et al., 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1.

[Friedman, 1978] Friedman, H. (1978). Classically and intuitionistically provably recursive functions. *Higher Set Theory*, pages 21–27.

[Gifford and Lucassen, 1986] Gifford, D. and Lucassen, J. (1986). Integrating functional and imperative programming. In *ACM Symposium on Principles of Programming Languages*.

[Girard et al., 1989] Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*, volume 7. Cambridge Tracts in Theorical Comp. Sci.

[Gödel, 1958] Gödel, K. (1958). Über eine bisher noch nicht benützteerweiterung des finiten standpunktes. *Dialectica*, 12:280–287.

[Griffin, 1990] Griffin, T. G. (1990). A formulæ-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Langages*, pages 47–58.

[Harper et al., 1993] Harper, R., Duba, B. F., and MacQueen, D. (1993). Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484.

[Hatcliff and Danvy, 1994] Hatcliff, J. and Danvy, O. (1994). A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA. ACM.

[Hatcliff and Danvy, 1997] Hatcliff, J. and Danvy, O. (1997). Thunks and the lambda-calculus. *J. Funct. Program.*, 7(3):303–319.

[Herbelin, 2005] Herbelin, H. (2005). On the degeneracy of sigma-types in presence of computational classical logic. In Urzyczyn, P., editor, *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Hoare, 1971] Hoare, C. A. R. (1971). Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, volume 188, pages 102–116. Springer.

[Honda et al., 2006] Honda, K., Berger, M., and Yoshida, N. (2006). Descriptive and relative completeness of logics for higher-order functions. *Automata, Languages and Programming*, pages 360–371.

[Honda et al., 2005] Honda, K., Yoshida, N., and Berger, M. (2005). An observationally complete program logic for imperative higher-order functions. *Symposium on Logic in Computer Science, LICS*, 5:270–279.

[Howard, 1969] Howard, W. A. (1969). The formulæ-as-types notion of constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculs and Formalism*, pages 479–490. Academic Press.

[Kelsey et al., 1998] Kelsey, R., Clinger, W., and Rees, J. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[Krivine, 1994] Krivine, J.-L. (1994). Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78.

[Krivine and Parigot, 1990] Krivine, J.-L. and Parigot, M. (1990). Programming with proofs. *J. Inf. Process. Cybern. EIK*, 26(3):149–167.

[Kuroda, 1951] Kuroda, S. (1951). Intuitionistische untersuchungen der formalistischen logik. *Nagoya Math. J*, 2:35–47.

[Landin, 1964] Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

[Landin, 1965a] Landin, P. J. (1965a). A correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101.

[Landin, 1965b] Landin, P. J. (1965b). A correspondence between algol 60 and church's lambda-notations: Part ii. *Commun. ACM*, 8(3):158–167.

[Landin, 1965c] Landin, P. J. (1965c). A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research.

[Leivant, 1990] Leivant, D. (1990). Contracting proofs to programs. In *Logic and Computer Science*, pages 279–327. Academic Press.

[Leivant, 2002] Leivant, D. (2002). Intrinsic reasoning about functional programs i: first order theories. *Annals of Pure and Applied Logic*, 114(1-3):117–153.

[Liang et al., 1995] Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343.

[Meyer and Ritchie, 1976] Meyer, A. R. and Ritchie, D. M. (1976). The complexity of loop programs. In *Proc. ACM Nat. Meeting*.

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML, Revised edition*. MIT Press.

[Moggi, 1990] Moggi, E. (1990). *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1):55–92.

[Murthy, 1990] Murthy, C. R. (1990). *Extracting Constructive Content from Classical proofs*. PhD thesis, Cornell University, Department of Computer Science.

[Murthy, 1991a] Murthy, C. R. (1991a). Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science.

[Murthy, 1991b] Murthy, C. R. (1991b). An evaluation semantics for classical proofs. In *Proc. 6th Annual IEEE Symp. on Logic in Computer Science*, pages 96–107.

[Nanevski et al., 2006] Nanevski, A., Morrisett, G., and Birkedal, L. (2006). Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73. ACM New York, NY, USA.

[Nielsen, 2001] Nielsen, L. R. (2001). A selective cps transformation. *Electronic Notes in Theoretical Computer Science*, 45:311 – 331. MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

[O'Donnell, 1982] O'Donnell, M. J. (1982). A critique of the foundations of hoare style programming logics. *Commun. ACM*, 25(12):927–935.

[Parigot, 1993a] Parigot, M. (1993a). Classical proofs as programs. In *Computationnal logic and theory*, volume 713 of *LNCS*, pages 263–276. Springer-Verlag.

[Parigot, 1993b] Parigot, M. (1993b). Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*.

[Peter, 1968] Peter, R. (1968). *Recursive Functions*. Academic Press.

[Plotkin, 1975] Plotkin, G. (1975). Call-by-name, call-by-value and the lambda-calculus. *TCS*, 1(2):125–159.

[Plotkin, 1981] Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

[Poernomo, 2003] Poernomo, I. (2003). Proofs-as-imperative-programs: Application to synthesis of contracts. *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003; Revised Papers*.

[Poernomo and Crossley, 2003] Poernomo, I. and Crossley, J. N. (2003). The curry-howard isomorphism adapted for imperative program synthesis and reasoning. *Proceedings of the 7th and 8th Asian Logic Conferences. World Scientific*.

[Rehof and Sørensen, 1994] Rehof, N. J. and Sørensen, M. H. (1994). The λ_δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag.

[Reus and Streicher, 2005] Reus, B. and Streicher, T. (2005). About hoare logics for higher-order store. *Automata, Languages and Programming*, pages 1337–1348.

[Talpin and Jouvelot, 1994] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

[Tan and Appel, 2006] Tan, G. and Appel, A. W. (2006). A compositional logic for control flow. *Verification, Model Checking, and Abstract Interpretation*, pages 80–94.

[Thielecke, 1998] Thielecke, H. (1998). An introduction to landin's "a generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–123.

[Wadler, 1994] Wadler, P. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55.

[Xi, 2000] Xi, H. (2000). Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara.